Delft University of Technology
Parallel and Distributed Systems Report Series

A Survey of Parallel Graph Processing Frameworks

Niels Doekemeijer
n.a.doekemeijer@student.tudelft.nl

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl

Report number PDS-2014-003

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Group:
http://www.pds.ewi.tudelft.nl/

N.A. Doekemeijer, A.L. Varbanescu

A Survey of Parallel Graph Processing Frameworks

Abstract

As graph analysis tasks see a significant growth in complexity - as exposed by recent advances in complex networks analysis, information retrieval and data mining, and even logistics - the productivity of deploying such complex graph processing applications becomes a significant bottleneck. Therefore, many programming paradigms, models, frameworks - *graph processing systems* all together - have been proposed to tackle this challenge. In the same time, many data collections have exploded in size, posing huge performance problems. Modern graph processing systems strive to find the best balance between simple, user-friendly and productivity-enhancing front-ends and high-performance back-ends for the analyses they enable.

Since 2004, more than 80 systems have been proposed from both academia and the industry. However, a clear overview of these systems is lacking. Therefore, in this work, we survey scalable frameworks aimed at efficiently processing large-scale graphs and present a taxonomy of over 80 systems. Useful for both users and researchers, we provide an overview of the state of the art techniques and remaining challenges related to graph processing frameworks.

Contents

List of Tables

1 Introduction

Graph theory provides an abstract model for entities and their relationships in the form graphs. Formally, a *graph* (or *network*) consists of a set of *vertices* (entities) and a set of *edges* (links that pair vertices). Many real-world phenomenons can intuitively be cast into this format. Newman [108] characterizes the use of networks in different branches of science and identifies the following four categories:

Social sciences study *social* networks, where vertices represent individuals and edges represent their pattern of contact (such as friendship[119] or kinship[113]). *Information* graphs represent the structure of stored information. Most studied in this category are the World Wide Web[70] and citation graphs for academic papers[34]. *Technological* networks are man-made networks, typically distribution oriented, such as power grids[154] and airline routes[4]. Finally, *biological* graphs model natural phenomenons like neural networks[155] and protein interaction[73].

These graphs can grow very large, very rapidly. For example, the citation graph among US patents between 1963 and 1999 contains over 3.7 million patents and over 16.5 million citation edges [89]. Early 2011, the number of active Facebook[1] users was 721 million with 68.7 billion friendship edges between them [145]. In December 2013, the number of active users had grown to 1.23 billion [43]. The largest publicly available hyperlink graph of the World Wide Web, extracted from a crawl in 2012, contains over 3.5 billion pages and 128.7 billion links [102].

Processing graphs on this scale is nontrivial and thus the topic has received a lot of attention from both academia and industry in the recent years. However, a clear overview is lacking. In this survey, we clarify the intrinsic challenges of building a high performance graph processing framework (Section 2) and illustrate the current research landscape by comparing graph processing frameworks with other systems, such as graph libraries and databases (Section 3).

Our work surveys scalable frameworks aimed at efficiently processing large-scale graphs and presents a taxonomy of over 80 systems. Specifically, systems that allow for offline parallel batch processing are taken into account. We identify multiple distinctive characteristics and categorize systems accordingly. The categories will be introduced (in order of appearance: Platform, Programming Model, Communication Model, Execution Model, Framework Abstraction) before presenting the taxonomy (Section 9).

The taxonomy is useful for both users that want an overview of the available frameworks and researchers that want an overview of the status quo and remaining challenges in the field of graph processing. We conclude this survey by discussing open research challenges (Section 10) and listing our main findings (Section 11).

2 Challenges

Graph problems have some characteristics that make efficient parallelization nontrivial; Section 2.1 discusses these problems inherent to graph computing. The other challenges are more general and related to the Input Data (Section 2.2) and Parallel Computing (Section 2.3).

The goal of a framework is to provide an *abstraction* over most of these challenges to prevent the user from dealing with substantial and repetitive implementation effort. In this way, a framework provides a trade-off between abstraction and expressiveness. Finding a proper balance is a challenge in itself.

2.1 Graph Processing

The term "graph processing" is broad and covers a whole set of analytic applications. Popular research fields that do some form of graph processing are data mining, machine learning, and pattern recognition [98]. Two categories can be distinguished [130]. *Online* graph analytics (or graph *querying*) start computation on a small subset of the graph and require a fast response time. These algorithms, e.g. shortest path, often perform

[1] facebook.com: Online social network website, founded in 2004.

some degree of exploration. *Offline* graph analytics (or *batch processing*) start on the whole graph and perform iterative computations until a convergence criterion is met. These algorithms, e.g. PageRank [114], require high throughput. Our survey focuses on systems explicitly supporting offline graph processing.

Low et al. [92] lists several key properties of machine learning and data mining algorithms (MLDM). The authors show that many important MLDM algorithms perform *iterative* computations (i.e. offline processing) on a large set of parameters, but computation converges *asymmetrically*. This means that a large number of parameters will converge in a just few iterations. It is also noted that these algorithms converge faster in *asynchronous* systems. Where synchronous systems perform updates based on values from the previous iteration, asynchronous systems benefit by always using the most recent values as input (see Section 7).

In general, graph computations are *data-driven* [62, 92, 96]. This, in combination with the *irregular structure* of graphs, leads to a poor *data locality* and a *varying degree of parallelism*.

2.2 Input Data

Laney [88] defines three dimensions of the challenges of data growth: *volume*, *velocity*, and *variety*. Often associated with the term "big data", these challenges are illustrative for the current trends in data acquisition. For example, Meusel et al. [102] note that computing the strongly connected components of the hyperlink graph requires more than one terabyte of memory while using a lazy evaluation technique where successors lists are never stored in memory in uncompressed form. The size of the compressed dataset is around 375 gigabytes.

Data variety comes from annotating and combining datasets. In graph theory, vertices and edges can be annotated with arbitrary properties. For example, edges can be labeled to define a nominal difference (A is a son of B) or can carry weight to define a degree of difference (traveling from A to B takes 60 minutes).

Graph processing systems have to deal with these three dimensions of "big data". Loading the entire graph into the memory of a single machine might be impossible, so efficient *storage* is desired (e.g. aggregate memory of a cluster). The variety of data and velocity of changes make this extra challenging.

Constantly crawling the web results in a *stream* of mutations. However, a single graph mutation does not have to imply a completely different result. Algorithms like PageRank can benefit from incremental computation based on a (partial) previous result [59]. Processing data streams also requires high *performance* for high-throughput computation.

2.3 Parallel Computing

Parallel computing, where multiple processing elements are used concurrently, offers challenges as well. Because computation is data-driven (Section 2.1) and partitioning is an NP-complete problem (i.e. minimum cut into equal-sized subsets [47]), it is difficult to realize independent tasks. Tasks have to *coordinate* for proper *synchronization* and data *consistency*. That is why the overhead of *communication* in a distributed system and Non Uniform Memory Access (*NUMA*) effects should be taken into account.

As partitioning is hard and graph computations have a varying degree of parallelism (Section 2.1), *load balancing* is another challenge in the parallelization of graph processing.

Distributed computing, where multiple machines are used concurrently, adds the problem of *reliability*. The odds of machine-failure are non-negligible when a large number of machines are used. Ideally, the system should be able to transparently detect failures and recover computation [147].

Parallelization might be implemented in the computation itself, such as concurrent processing of vertices [99], but also in the execution of jobs. Concurrent job processing of the same graph is a challenge, especially when graph mutations are allowed [159].

3 Research Landscape

Graph processing is a popular topic among researchers because of its potential in numerous disciplines (Section 1) and the specific set of computing challenges (Section 2). This section clarifies the term "graph processing framework" by relating it to other work in the field, specifically Graph Algorithms, Graph Libraries, Graph Databases, General-Purpose Data Processing, and Domain-Specific Languages.

3.1 Graph Algorithms

A lot of work goes into optimizing specific graph algorithms targeting specific platforms ([60, 67, 71, 118]). Guo et al.[54, 55] present five classes of algorithms used in practice: statistics, traversal, connected components, community detection, and evolution. Representative examples of such algorithms that are often used as building blocks — and receive extra attention because of it — are: diameter estimation [1, 23, 79], Breadth-First Search (BFS) [67, 85, 97, 101], Maximal Independent Set (MIS) [3, 37, 94], spectral clustering [29, 45, 127, 170], and preferential attachment graph modeling [14, 15, 89], respectively.

Low-level implementations of such an algorithm allows for very for specific architectural optimization, but are subject to substantial implementation effort. A lot of this effort is repeated for each new algorithm, i.e. loading into memory, graph representation (data structures), and fault tolerance are nontrivial tasks. Our survey does not take (library) implementations of algorithms into account.

Frameworks provide generic functionality through abstraction, such that the user does not have to deal with most of the challenges related to graph processing (Section 2). Ideally, the abstraction should not come at the expense of performance or expressiveness.

3.2 Graph Libraries

Often used in the same sense, and closely related to the concept of framework, is the term library. Johnson and Foote [75] differentiate between the two with the notion of "inversion of control". Methods defined by the user will be called by the framework itself, which means that it serves as "extensible skeleton".

Where frameworks also take care of the control flow, libraries only offer a collection of objects and functions, but leave coordination to the user. Graph libraries such as Parallel BGL[53] and Combinatorial BLAS[21] offer algorithms as building blocks and (distributed) data structures for graph representation.

Unlike libraries, parallel graph frameworks offer a programming model (Section 5) where things like parallelism, consistency, synchronization, load balancing, and optimization are (mostly) implicit and the user can focus on implementing the actual application (Section 8).

3.3 Graph Databases

Graph DataBase Management Systems (DBMSs) such as Neo4j[107] and Titan[9] are DBMSs optimized for graph structures. DBMSs are persistent storage systems that can contain, represent, and query data [5, 32]. In contrast to relational DBMSs, entity relations are part of the model and no (expensive) table join is necessary to access adjacent elements [149].

Systems usually offers a domain-specific language such as SPARQL, Cypher, or Gremlin for text-based data queries (online graph analytics) [65], but do not allow for batch processing (offline graph analytics) [100].

We only take graph databases that allow for offline parallel graph analytics into account ([31, 117, 130]). For a survey of graph database models, the reader is referred to [5] and for performance oriented surveys the reader is referred to [32, 65, 100].

3.4 General-Purpose Data Processing

MapReduce[35], Spark[164], and epiC[74] are examples of general-purpose distributed data processing frameworks. The frameworks provide a certain level of abstraction (for fault tolerance, coordination, parallelism, etc.), but are not optimized for working with graph data and graph algorithms.

Representing graphs and algorithms might still require substantial implementation effort [104], which results in specialized graph processing frameworks being built on top of general-purpose frameworks ([78, 80, 109, 158]). Google, the organization that introduced MapReduce in 2004, recognized that existing general-purpose systems were ill-suited for graph processing and introduced Pregel[99] in 2010, a framework for graph processing that increases performance and usability compared to general systems.

3.5 Domain-Specific Languages

A Domain-Specific Language (DSL) is a programming language specialized for a domain such as graph processing. Programming interfaces to frameworks are often implemented in a general-purpose language and at a fairly low abstraction level, which means that implementation of nontrivial algorithms can be inconvenient [68, 111]. Graph DSLs such as OptiML[139], Green-Marl[66], and general-purpose data processing DSLs such as Pig Latin[111] and Hive[142], offer a more natural programming interface for users familiar with the domain, but not with programming in general.

DSLs are usually the front-end of a framework and compile to a lower-level execution framework. For example, Green-Marl targets GPS (a Graph Processing System [124]) and Pig Latin targets Apache Hadoop[33], an open source implementation of MapReduce. We only take the execution frameworks into account for this survey.

4 Platform

The target platform is one of the major distinctions between frameworks. As a user, selecting the best suitable framework depends on the available platform. As a researcher, these are assumptions that can influence design. Will execution be performed on Shared Memory Systems (i.e. a single machine) or Distributed Systems? Will the graphs surely fit into aggregate memory or should the framework have External Memory Support? Is the environment homogeneous or are there differences in computational strengths (i.e. Heterogeneous Environments)? Finally, should representation of input and computational data be restricted for the benefit of performance optimization (Data Representation)?

4.1 Shared Memory Systems

Shared memory allows for efficient inter-process communication, as multiple tasks have access to the same memory. A shared memory platform does not imply a single processing unit. The challenges in graph processing (Section 2.1) — poor data locality, in particular — make shared memory architectures well suited for graph processing.

However, shared memory architectures usually support a limited amount of physical memory (Red Hat Linux Version 7 supports 3TB of memory, with a theoretical limit of 64TB [121]). Scaling up refers to adding resources (e.g. CPUs or memory) [103].

From 2010 onward, there has been an increase of frameworks that work on a single machine (shared memory). Signal/Collect[138], GraphLab[93], and GraphChi[87] were the first frameworks to prefer a shared memory environment over distributed architectures to enable graph processing on consumer computers.

Although scalability is limited (i.e. limited by the capabilities of the current generation of hardware), communication is efficient, cost is lower, and debugging is straightforward. The user does not have to deal with

managing a cluster, and the framework does not have to deal with quirks that come with distributed computing, such as fault tolerance.

4.2 Distributed Systems

A distributed system consists of multiple processing units where each unit has its own private memory. Data is partitioned over the separate nodes and explicit communication (e.g. message passing) is required to synchronize computation. Scaling out refers to adding more processing units to the system [103], and with cloud computing this type of scaling is available through Infrastructure as a Service (IaaS) [8].

General-purpose data processing frameworks such as MapReduce[35] (2004) and Dryad[72] (2007), and later graph processing frameworks such as PEGASUS[80] (2009) and Pregel[99] (2010), started exploiting the distributed architectures of commodity clusters to enable efficient processing of large volumes of data.

Compared to shared memory systems, distributed systems are less dependent on hardware evolution for scaling, but communication between machines quickly becomes a performance bottleneck in graph applications.

4.3 External Memory Support

External memory support (or out-of-core computation support) refers to the ability to work with data that is too large to fit in main memory. Some graph processing systems are able to accommodate graphs that exceed the size of aggregate memory by utilizing (slower) external memory.

For distributed systems, general-purpose frameworks typically support transparent spilling of data to disk [2, 35, 72, 74, 105, 115]. For example, MapReduce works in phases where data is streamed to and from disk. Map and Reduce results are immediately written to a distributed file system. This disk overhead makes MapReduce less suited for iterative applications [19, 39], which is why the majority of distributed graph processing frameworks keep the graph in aggregate memory.

On shared memory systems, however, a lack of sufficient main memory is more probable because of the practical limits (Section 4.1). In 2012, GraphChi[87] was the first single machine framework to utilize external memory; by storing the graph in an optimized format on disk, the framework can still process graphs that do not fit into main memory.

4.4 Heterogeneous Environments

A heterogeneous environment refers to an environment where not every processing unit is equally powerful. For a single machine, a processing system may optimize for specific hardware, rather than assuming a generic back end. For example, a Solid-State Drive (SSD) has different performance characteristics than a Hard Disk Drive (HDD). Instead of assuming a black box for persistent storage, RASP[161] and FlashGraph[169] optimize for SSD storage, which handles simultaneous non-sequential requests much better than HDD storage.

Similarly, a graphical processing unit (GPU) can be used to offload (part of) the computation. GPUs are an integral part of mainstream computing systems and are highly parallel processors, but markedly different from commodity processors [112]. For example, TOTEM[49] processes high-degree vertices on the CPU and offloads the low-degree vertices to the GPU. Other frameworks offload computation for the whole graph to the GPU ([46, 83, 171, 172]).

Hardware and network topology does not have to be uniform in a cluster. Some machines may have a newer generation of hardware or some machines may be better connected than others. Surfer[29] recognizes these heterogeneity challenges for cloud computing and tries to partition the graph based on available bandwidth between machines.

4.5 Data Representation

Graphs come in many different shapes and forms (Section 2.2). General-purpose frameworks leave loading input data to the user. This allows users to work with any desired graph, although substantial and repetitive implementation effort might be required for each new application. Pregel[99] introduced graph templates to let programmers use custom data types for vertices and edges in combination with a simplified loading scheme.

Graph processing systems generally restrict input to specific graph types. An undirected edge can be represented using two directed edges, which is why most graph frameworks work with directed edges. Few have explicit support for undirected graphs to reduce memory requirements ([25, 53, 130, 137, 169]). Hyperedges, i.e. edges that join more than two vertices, are not explicitly supported by any framework. Representing a graph using a matrix restricts the number of edges between vertices to one at most, which means that multigraphs cannot be represented.

A lot of graph analytic algorithms use the graph structure as an independent, immutable topology. For example, in computations for PageRank, there is a value assigned to each vertex, but the actual structure of the input graph remains static. Although graph mutation is crucial for some algorithms, such as graph coarsening and graph sparsification [143], facilitating the feature increases difficulty as mutation conflicts can arise [99] and memory management must be dynamic [48].

5 Programming Model

A higher-level programming interface should make it easier for the user to implement graph applications. Depending on the level of abstraction, the framework can implicitly take care of challenges such as parallelization, fault tolerance, and optimization (Section 8).

Three categories of programming models are discussed: General-Purpose programming models, Vertex-Centric, and Graph-Centric models.

5.1 General-Purpose

MapReduce[35] is a general-purpose framework which offers a programming model of the same name. Inspired by the functional programming paradigm, the *map* and *reduce* tasks work with immutable key/value pairs of data. The output from the *map* tasks will be the input for the *reduce* tasks (grouped by key). Programs are written in a serial fashion and will be automatically parallelized. Tasks are independent so simply rescheduling failed tasks masks failures.

A generalization of this model is the Directed Acyclic Graph (DAG) model. In this model, an application is represented as a DAG, where vertices represent tasks and edges represent the flow of data. For MapReduce, the DAG is simple with two vertices (*map* and *reduce*) and one edge between them. Most frameworks that work with DAGs offer basic tasks (map, reduce, join, sort, etc.), but allow users to implement custom operations as well ([2, 17, 164]).

DAGs cannot express iterative applications, because of the acyclic restriction. This can be worked around by adding an *iterate* task that executes a DAG until a convergence condition is met ([2, 20, 39, 166]). CIEL[105] allows users to spawn tasks dynamically, resulting in a dynamic DAG. epiC[74] provides a lower-level actor-based model on which models like MapReduce and DAG are implemented.

Less general, but still multi-purpose, is a matrix-centric abstraction model. Graphs are represented using an edge table (sparse matrix) and a vertex table (vector), and algorithms are formulated as a series of generalized matrix-vector multiplications ([21, 58, 78, 80, 148]).

5.2 Vertex-Centric

Malewicz et al. [99] (Pregel) introduced the “think like a vertex” paradigm, in which computation centers around

a single vertex. This graph-oriented paradigm limits the scope of each computation to create a fine-grained parallel system. Malewicz et al. [99] originally limit the direct scope of a vertex's *Compute* function to its outgoing edges, but other compositions have been experimented with as well; Hoque and Gupta [69] (LFGraph) only expose the incoming edges, while Low et al. [93] (GraphLab) give access the direct neighborhood (incoming and outgoing edges, and the vertices they connect to).

Load balancing vertex-centric computations can be difficult for graphs with an unbalanced degree distribution — a phenomenon present in many real-world graphs [14]. Processing individual edges, a finer computation granularity, works around this problem. Gonzalez et al. [51] (PowerGraph) adjust Pregel's vertex-centric programming model to work with edge computations. Computation is split into three phases: *Gather*, *Apply*, *Scatter*. Both the *Gather* and *Scatter* operations are applied on edge-level, rather than vertex-level. The *Apply* function is similar to Pregel's *Compute*, but only has access to the "combined" edge value (accumulated in the *Gather* phase). *Scatter* updates the edge values based on the new vertex value. GRE[160] and X-Stream[122] have similar models, but move the Scatter phase to the beginning of an iteration.

Similar to vertex-centric in terms of scope limitation, but fundamentally different in mindset, is the visitor-based programming model. In a vertex-centric model, all vertices are scheduled concurrently, whereas a visitor-based model schedules vertices based on an underlying search pattern. Users define callbacks for exploratory events, such as vertex discovery or edge traversal in a depth-first search [16]. This pattern is adopted mostly by parallel graph libraries ([16, 53, 61, 64]) that have taken over this approach from traditional libraries (i.e. the Boost Graph Library [136]). However, Buluc and Gilbert [21] note that this approach is inherently difficult to scale, because of the overhead in coordination.

5.3 Graph-Centric

The vertex-centric programming model is a high-level, graph-oriented, scalable abstraction. Some algorithms map naturally to its interface (e.g. PageRank [99]), but others are considerably more difficult to represent (e.g. betweenness centrality [68]). Restructuring traditional graph algorithms to be vertex-centric can be nontrivial and will result in non intuitive code. Green-Marl[66] offers a graph-centric model and lets the compiler handle optimization and parallelization. Compared to vertex-centric, a graph-centric model incorporates the notion of a (sub)graph.

Tian et al. [143] argue that "think like a graph" is more desired than "think like a vertex". Such a model, as implemented by Giraph++[143] and GoFFish[137], allows for more coarse-grained parallelism. Rather than limiting computation scope to a single vertex, these systems perform computations on graph partitions. The authors argue that this approach increases data propagation speed significantly and makes it easier to port traditional algorithms.

6 Communication Model

Communication is an important element in graph analysis, as tasks have to communicate for coordination and data synchronization (Section 2.3). We distinguish between three models for communication: Dataflow, Message-Based, and Shared Memory. Finally, as will be discussed in Section 6.4, a distinction can be made in the direction of communication.

6.1 Dataflow

Dataflow refers to a communication model where state (i.e. data) flows through the system towards the next phase of computation. A DAG programming model (Section 5.1) represents this inherently; data moves along edges towards the next task. Load balancing and fault-tolerance are relatively straightforward as tasks can be executed on every processing unit. A task can even be preemptively scheduled on multiple machines [35].

However, in graph computations, the actual structure of the graph is mostly static and transferring this data causes a lot of communication overhead, especially over multiple iterations [39, 115, 166]. Frameworks like Twister[39], MR-MPI[115], and iMapReduce[166] differentiate between state data and static data, while Haloop[19] makes the scheduler loop-aware to minimize the excess of data movements between machines.

6.2 Message-Based

In a message-based communication model, state is local and messages flow through the system to update external entities. Pregel[99] first introduced the message passing concept for graph processing with its Vertex-Centric programming model. Here, each vertex is allowed to send messages to other vertices in the graph.

To make sure all new data is available for computation, a global synchronization step is required between iterations. This sequence of parallel computation, communication and synchronization is referred to as Bulk Synchronous Parallel (BSP) processing [146]. Fidel et al. [44] optimize for algorithms where the number of incoming messages is known a priori. If the exact number is known beforehand, each vertex can serve as a synchronization point.

To remove the synchronization step, update methods would have to be able to work with partial data. Zhang et al. [167] (Maiter) introduce a Delta-Based Accumulative Iterative Computation (DAIC) vertex-centric programming model that works with delta-based updates. However, this model can only represent algorithms in which messages are processed in an accumulative fashion.

6.3 Shared Memory

Communication through shared memory allows multiple processing units to access and mutate the same data. In Distributed Systems, the framework has to take care of transparent synchronization between workers to offer virtual shared memory. Race conditions are imminent when concurrent jobs can both read and write to the same memory space, so data consistency has to be taken care of explicitly.

In graph processing systems, virtual shared memory is realized through the use of ghost vertices [53, 93]. One worker is assigned ownership of the vertex, while other partitions work with immutable copies. Consistency can be assured by keeping ghosts immutable during an iteration [69], with (distributed) write locks [92], or with an accumulator [53, 116]. Restricting computation scope to adjacent neighbors limits the extra memory required for ghost data [51, 69, 92, 137].

6.4 Flow Model

Exchange of information between vertices can flow in two directions. In a *push* style flow, information flows from a vertex to its neighbors. For example, in a Single-Source Shortest Path (SSSP) algorithm, the active vertex notifies its neighbors of its new path length after an update [109]. In a *pull* style flow, information flows in the reverse direction, and the active vertex updates its own shortest path length by proactively reading the lengths of its neighbors' paths.

In a pull mode flow, consistency is inherently guaranteed because the active vertex only updates itself. The downside is that a vertex is uninformed of neighbor updates, so there might be an overhead in checking for changes. In contrast, a push mode flow requires locks for every neighbor update [59]. Some algorithms cannot be expressed without support for pull-based communication (e.g. betweenness centrality), while for other algorithms it is an optimization (e.g. PageRank) [50].

Dataflow and Message-Based communication naturally map to a push-based communication flow, while Shared Memory maps to a pull-based flow. However, no communication method is inherent to one specific flow model. Active message pushing can be prevented by caching messages between iterations [167] and neighboring vertices can be locked in a shared-memory computation [92].

Some frameworks restrict the information flow to one direction (Pregel is push-based, while LFGraph[69] is pull-based), but multiple frameworks facilitate both communication models ([31, 51, 59, 92]).

7 Execution Model

A high level Programming Model separates the computation model from the execution model. The execution model is partly determined by the computation scheduling order. The distinction between Synchronous Execution and Asynchronous Execution stems from the difference in time for when an updated value becomes visible to the rest of the graph. Scheduling can be done in batch, incrementally, or prioritized.

7.1 Synchronous Execution

Synchronous execution of a graph algorithm can be depicted as a sequence of iterations, delimited by a global barrier. Each iteration performs updates based on values from the last iteration (in parallel), and updated values are only exchanged between iterations. This regular communication interval makes it suitable for algorithms that perform lightweight computation and intensive communication, as communication bandwidth can be utilized better [156].

Synchronous execution prefers a large number of updates in each iteration so that the overhead of a global barrier is minimized [156], which makes it unsuitable for graph applications where computation converges asymmetrically.

It is also unsuitable for applications that require coordination between adjacent vertices. For example, in a greedy graph coloring algorithm, all vertices would synchronously change to the same color [51, 141, 156].

Push-based communication models (i.e. Dataflow and Message-Based) are synchronous by default, unless computation is performed on partial data (such as with accumulative message handling [167]). Extra memory is required to buffer updates between iterations.

7.2 Asynchronous Execution

Asynchronous execution lets updates be performed on the most recent data. Synchronization is performed as soon as possible, rather than through a global barrier, resulting in an irregular communication interval. This makes makes it suitable for applications that perform imbalanced computation and little communication [156].

In contrast to synchronous execution, development and debugging is more difficult because of the nondeterministic nature of computation [51]. In return, convergence is faster for many algorithms when updates are performed on most recent values (belief propagation, for example [52]). Some frameworks enforce a certain degree of determinism ([51, 87]) to make sure that multiple executions have the same result (note that this does not have to be equal to the result of a synchronous execution).

A pull-based communication model (i.e. Shared Memory) easily allows usage of most recent data. However, this does risk computation on unchanged data, resulting in useless work [168]. Because reads and writes may be intertwined, it also requires a system to introduce read-locks for data consistency [92].

When computation is centered around a group of vertices (i.e. a Graph-Centric programming model), data propagation within the group is asynchronous. Frameworks that make a distinction between local and remote vertices can benefit from local asynchronous computation while synchronization of remote values is still performed in synchronous iterations ([48, 141, 143]).

Xie et al. [156] note that the performance of the two execution modes varies significantly with different platforms, applications, input graphs, and execution stages and using an inappropriate execution mode may result in performance loss. The authors introduce PowerSwitch, a framework that transparently switches between modes during execution based on heuristic predictions.

Some systems are limited to one execution mode ([87, 99]), while others let the user choose during implementation ([92, 130]).

7.3 Scheduling

In parallel computation, all jobs are ideally performed concurrently. However, most programming models adopt a fine-grained parallelization scheme where the number of jobs exceeds the number of processing units. For example, in a Vertex-Centric model, each individual vertex is scheduled for computation, which means that each processing unit is assigned multiple jobs. The scheduling model determines the order in which jobs are processed.

In a *bulk* iteration model, all data is scheduled for processing in each iteration. Jobs are executed in arbitrary order and no discrimination is made by the importance of a calculation. This is common for Dataflow frameworks, where state flows through the system ([19, 39, 40, 166]). A convergence condition determines when to quit iterating; for example, when the result hasn't changed (fixed-point iteration), or when the number of iterations has exceeded a limit.

Many iterative graph applications, however, converge asymmetrically (Section 2.1). The majority of vertices require only a single update for PageRank [92]; rescheduling already converged vertices for execution adds considerable overhead. *Incremental* scheduling processes only a subset of the data (the "active" set of vertices). Vertices can schedule themselves and others for future computation, implicitly (with a push-based Flow Model, e.g. Stratosphere[2] or Pregel[99]) or explicitly (e.g. GraphLab[92, 93]). Convergence is reached when no active vertices are left.

Zhang et al. [168] show that *prioritized* scheduling can result in faster convergence for several graph applications, such as SSSP and PageRank. Note that this scheduling method implies incremental scheduling and Asynchronous Execution. Here, vertices are explicitly scheduled for execution in combination with a user-defined priority level. Jobs are processed in descending order of priority, such that more influential computations are executed first.

8 Framework Abstraction

As discussed in Sections 3 and 5, a framework serves as an "extensible skeleton" that prevents users from dealing with substantial and repetitive implementation effort. This section discusses several features that can be taken care of through framework abstraction: Parallelization, Partitioning, Load Balancing, Fault Tolerance, and Optimization.

8.1 Parallelization

We only take parallel graph processing systems into account for this survey. The parallelization aspect can be done *implicitly* by the framework (i.e. the user writes a sequential program) or *explicitly*. In contrast to libraries, frameworks work with the notion of "inversion of control", which means that user code is called by the framework rather than the other way around [75]. This implies that the framework takes care of job scheduling (Section 7.3) and concurrent job execution.

However, some aspects of a correct parallel implementation might still be the responsibility of the user. Several systems offer multiple data consistency models from which the required model is manually chosen for each application ([51, 92, 93, 109]). Some systems require the user to work with atomic operations or accumulation functions ([116, 134]), while others leave consistency fully to the user ([48, 74, 83]). Krepska et al. [86] (HipG) work with customized synchronizers to explicitly coordinate barriers. When the user is required to consider the parallel aspect of an application, we refer to explicit parallelization. Otherwise, parallelization is implicit.

8.2 Partitioning

Partitioning refers to way in which data is divided between workers. Traditionally, a good data distribution optimizes for equal processing time while minimizing communication between workers. However, realizing a minimum graph cut with equal-sized subsets is an NP-complete problem [47]. As a problem in itself, graph partitioning has received a lot academic attention. For a general overview of recent work, the reader is referred to Buluc et al. [22]. For more parallel computing oriented surveys, the reader is referred to Hendrickson and Kolda [63], and Schloegel et al. [128].

In graph processing frameworks, we distinguish between partitioning techniques based on three classifications. First, does the input require a *preprocessing* step or does the algorithm work *on-the-fly*? On-the-fly (or *streaming*) graph partitioning reduces overhead by limiting computation to a single pass, while a preprocessing step is impeded less by time and other resource restrictions.

Several frameworks that optimize for out-of-core processing require a preprocessing step to convert input into an optimized storage format [58, 83, 87, 106, 163]. TOTEM[48] partitioning uses the degree distribution of the input graph, which requires an extra pass over the data. Chen et al. [29] (Surfer) process not the input data, but the network architecture to enable optimized partitioning.

Although not required, on-the-fly algorithms can still benefit considerably from a preprocessing step [22]. Salihoglu and Widom [124] (GPS) report an improvement in run time by up to 2.5x using this approach. Preprocessed data can be used for multiple runs.

Generally, the surveyed frameworks work with a simple streaming partitioning method to minimize the overhead of data loading. Typical algorithms for this purpose are random, range, and round-robin partitioning [36]. A simple abstraction, i.e. $hash(key)$ **mod** R, where R is the number of partitions and *hash* is a user-defined function, allows for a certain degree of customization ([35, 99]). Using the identity function for *hash* results in a round-robin partitioning, while using a cryptographic hash function results in a more random partitioning.

The second distinction in partitioning techniques for graph processing is made by the support for *dynamic* repartitioning during execution, rather than using the same *static* distribution from start. Runtime behavior of an algorithm can be unpredictable, so using an adaptive partitioning method can improve performance. As there is extra communication required for reassigning data, it's only beneficial for applications with more than a few iterations [12, 82, 124]. Note that dynamic repartitioning is implicit for load balanced Dataflow frameworks that do not separate static and state data.

The final distinction can be made by looking at where in the graph is cut. An *edge-cut* evenly assigns vertices to partitions with a minimal number of crossing edges, while a *vertex-cut* evenly assigns edges to partitions with a minimal number of crossing vertices. For many real-world graphs, where degree distribution follows a power law [14], a vertex-cut leads to a more balanced partitioning [51]. However, computations need to be expressed on edge-level to allow for efficient parallel computation [51, 122, 160].

8.3 Load Balancing

In parallel computing, it is key to evenly distribute workload between available workers. Stragglers are tasks that take significantly longer to finish than the rest, and prevent further execution because of it. They arise when workload is unbalanced, through improper partitioning, multi-tenancy or hardware variability, for example.

Although tightly related to Partitioning, load balancing can also be realized by other (transparent) means. When data is shared or replicated between workers, computation is not restricted to one processor. In load balancing through *work stealing*, the first available worker takes care of the computation. This approach is generally used in General-Purpose Data Processing systems, such as iHadoop[40] and iMapReduce[166], but also in shared-memory frameworks, such as Grace[117], MapGraph[117], and FlashGraph[169]. In stateless systems, preemptively scheduling duplicate tasks can help mitigate stragglers [35, 72].

When no adaptive load balancing is performed, the initial data distribution determines the workload balance. Dynamic repartitioning can also be regarded as an adaptive load balancing technique. We will distinguish

between *static* and *dynamic* load balancing techniques. For an extensive comparison between both, the reader is referred to Kameda et al. [77].

8.4 Fault Tolerance

Reliability becomes a challenge in a system with multiple machines (Section 2.3), as the chance of machine-failure is non-negligible [147]. Failing machines can be detected by means of a simple timeout mechanism, either by the whole group or by a central coordinating process. We observe that most of the surveyed frameworks aimed at working in a distributed environment adopt the master-worker paradigm, where a single master process is responsible for global coordination and a group of workers perform computation.

Treaster [144] differentiates between recovery techniques for the central and parallel components in an application. The most common means for both are replication and rollback, respectively. For fault tolerance through replication, a second machine acts as a copy of the main machine and takes over in the event of failure. Fault recovery with a rollback mechanism assumes the presence of stable storage (where data will be accessible even if the machine fails). Rollback protocols can be based on checkpoints or message logs [41]. Most surveyed frameworks work with a distributed checkpointing protocol that saves state to a distributed file system (e.g. Hadoop Distributed File System [135]), but pay little attention to failure of the master machine.

Low et al. [92] note that a strong emphasis on fault tolerance in graph processing systems is questionable, because the optimal interval between checkpoints (balancing cost of fault tolerance against cost of a job restart) usually exceeds total application running times. Wang et al. [152] (Imitator) introduce a replication-based protocol for workers based on already present ghost vertices.

We will distinguish between frameworks through the impact of machine failure. Transparently *rescheduling* work from a failed machine has little recovery impact, but might introduce overhead to facilitate ([2, 35, 152, 159]). Checkpoints introduce less overhead during computation, but a *rollback* possibly requires redoing some computations ([39, 92, 99, 116]). When fault recovery is not explicitly supported, the system requires computation to *restart* from scratch [56, 59, 86, 115]).

8.5 Optimization

Many graph applications contain similar computation patterns (Section 2.1). By providing a Programming Model that explicitly supports such patterns, frameworks can optimize their implementation.

The most notable example for such a pattern is *iterative* computation. Early general-purpose data processing frameworks (i.e. MapReduce[35] and Dryad[72]) did not have explicit support for computation until convergence, which results in a significant overhead of manual iterative scheduling (hence the emergence of iterative MapReduce variants, such as HaLoop[19], Twister[39], and iMapReduce[166]). All surveyed graph processing frameworks support this feature.

Another feature that most graph frameworks explicitly support is an *aggregation* mechanism (e.g. *aggregators* in Pregel[99], *sync operation* in GraphLab[92, 93], *accumulators* in Spark[164]). Such a mechanism combines values from each independent computation and globally publishes the aggregate before the next iteration. This can be used for global coordination and to gather statistics. Note that a similar result can be achieved by adding a custom aggregation vertex with undirected edges to all other vertices. However, broadcasting the same value to all workers can be optimized by the framework [150]. Low et al. [93] (GraphLab) apply this feature in an asynchronous setting, where aggregations run continuously in the background.

Some graph algorithms are a combination of parallel and sequential computations [56, 124]. Multiple frameworks support *global execution*, i.e. the explicit expression of sequential computations between parallel iterations ([56, 124, 134, 173]). This prevents the sacrifice of a globally coordinated parallel iteration where only one computations takes place.

9 Taxonomy

Table 1 presents our taxonomy of the surveyed parallel graph processing systems. As described in Section 4, we distinguish between **S**hared and **D**istributed memory systems. Similarly, programming models (Section 5) are categorized into D**A**G, **M**atrix, **V**ertex-centric, **G**raph-centric, and V**I**sitor models. Master-**W**orker and **D**ecentralized are the categories for coordination. For the other features listed in Sections 4 to 8, a distinction is made between explicit, user choice, and implicit support. Systems are ordered by year of first appearance and grouped into four categories: general-purpose, library, distributed architecture, and shared memory architecture.

With MapReduce in 2004, Dean and Ghemawat [35] started a trend of distributed general-purpose data processing frameworks. Around the same time, Gregor and Lumsdaine [53] and Hielscher and Gottschling [64] made an effort to parallelize the Boost Graph Library (BGL) [136]. However, it wasn't until 2009 that frameworks offered a more suitable interface for arbitrary graph applications; PEGASUS[80] introduced a more graph-oriented programming model (generalized matrix-vector multiplication), while Stratosphere[2] added support for iterative computation in a general-purpose DAG programming environment (Section 5.1). In the following years, general-purpose frameworks further optimized MapReduce ([19, 39, 40, 115, 165, 166]) and other DAG models ([17, 74, 104, 105, 164]) to work better for a broader range of applications (such as graph processing).

Malewicz et al. [99] introduced Pregel in 2010, a truly graph-specific framework with a vertex-centric programming model. As source code was not disclosed, multiple open source clones originated (Phoebus[140], JPregel[24], Bagel[6], GoldenOrb[120], Giraph[7], Pregelix, BC-BSP[13]). Giraph, the most notable clone, is currently used by Facebook. Giraph ([124, 131, 141, 143, 172]) has since evolved with help of academic attention. Rather than taking a drastically different approach, most research focuses on the value of complementary features such as a slightly different programming model (Section 5.2), asynchronous execution (Section 7.2), or dynamic repartitioning (Section 8.2).

Inspired by Pregel, a surge of the number of single machine frameworks occurred in 2012. Recent research in this direction tries to exploit performance characteristics of modern hardware (Section 4.4)) such as SSDs ([161, 169]) and GPUs ([25, 46, 83]). Although programming models are similar, implementation is much more focused at efficient storage, rather than communication in distributed systems.

We make several observations from our survey. Firstly, graph libraries aside, all groups have been a hot topic of research. Secondly, the increased generality of general-purpose systems has resulted in several front ends for graph-oriented applications ([78, 80, 109, 158]), which emphasizes that research in these directions overlap to a certain extent. Thirdly, there does not appear to be a consensus between framework builders regarding the "optimal" selection of features. Implicit parallelization, stream partitioning, and optimization for iterative computation and global aggregation are the only basic features (Section 8) implemented by most frameworks. The decisions for push/pull flow (Section 6.4) and synchronous/asynchronous execution (Section 7) are most often left to the user. Among graph processing frameworks, the vertex-centric programming model (Section 5.2) is very popular. Finally, a lot of systems are alike from a top-level point of view, but differ in implementation. Implementation greatly influences performance, but also other aspects, such as usability and stability. Measuring these aspects of a framework requires a more hands-on comparison.

		Platform				Computational Model						Framework Abstraction											Implementation			
Year	System	Architecture	Out-of-core	Heterogeneous	Mutable Graph	Programming	Push Flow	Pull Flow	Synchronous	Asynchronous	Incremental	Prioritized	Parallelization	Repartitioning	Vertex-cut	Edge-cut	Stream Partitioning	Load Balancing	Checkpoint Recovery	Reschedule Failures	Iteration	Aggregation	Global Execution	Coordination	Message-based	Dataflow	Shared Memory
2004	MapReduce/Hadoop[33, 35]	D	●		●	A	●		●				●	∗ ∗			●	●	●	●				W		●	
2007	Dryad/DryadLINQ[72, 162]	D	●		●	A	●		●				●	∗ ∗			●	●	●	?				W		●	
2009	Stratosphere[2, 153]	D	●		●	A	●		●		●		●				●	●		●	●			W		●	
2010	HaLoop[19]	D	●			A	●		●				●	∗			●	●	●	●	●			W		●	
2010	Twister[39]	D				A	●		●				●				●		●		●	●		W		●	
2010	Hama[129]	D				VMA	●		●		●		●			●	●		●		●	●		W	●		
2010	Spark[164]	D			●	A	○	○	●				●				●	●		●	●	●		W		●	
2010	Piccolo[116]	D			●	V		●	○	○							●	●	●		●	●	●	W			●
2011	MR-MPI[115]	D	●		●	A	●		●				●				●				●	●		W	●		
2011	Hyracks[17]	D			●	A	●		●				●	∗			●	●	●			●		W		●	
2011	iHadoop[40]	D				A	●		●				●				●	●		●	●			W		●	
2011	iMapReduce[166]	D	○			A	●		●				●	∗ ∗			●	●	●		●			W		●	
2011	PrIter[165]	D	○			A	●			●	●	●	●	∗ ∗			●	●	●		●			W		●	
2011	CIEL/SkyWriting[105]	D	●		●	A	●		○	○	●		●	∗ ∗			●	●		●	●			W		●	
2013	Naiad[104]	D			●	VA	●		○	○	●		●	● ∗	○	○	●	●	●		●	●	●	W		●	
2014	epiC[74]	D	●		●	A	○	○	○	○	●		●	∗			●	●		●	●	●	●	W	●		
2004	ParGraph[64]	D				GI		?	●		∗					●	●				●	●		D			●
2005	Parallel BGL[53]	D				GI		?	○	○	∗					●	●				●	●		D			●
2007	MTGL[16]	S				GI		?	●		∗										●			W			∗
2009	Thrust Graph Library[84]	S		?	●	GI		?	●		∗										●			W			∗
2009	STINGER[11, 38]	S			●	G															●			W			∗
2011	CombBLAS (KDT)[21, 95]	D				M	●		●				●		●		●				●			W		●	
2012	STAPL Graph Library[61]	SD			●	GVI		●	○	○	○	●	●	○		●	●	∗			●			D			●
2013	MMAP[123]	S	●			G		●													●			W			●
2009	PEGASUS[80, 81]	D	●			M	●		●				●		○	○	○	?	∗	∗	●			W		●	
2010	Pregel[99]	D			●	V	●		●		●		●			●	●		●		●	●		W	●		
2010	Surfer[29, 30]	D		●	●	G	●		●				●			●			●		●	●		W	●		
2010	Phoebus[140]	D			●	V	●		●		●		●			●	●		●		●	●		W	●		
2010	JPregel[24]	D			●	V	●		●		●		●			●	●		●		●	●		W	●		
2011	Bagel[6]	D			●	V	●		●		●		●			●	●	?		∗	●	●		W		∗	
2011	GoldenOrb[120]	D	●		●	V	●		●		●		●			●	●	?	∗	∗	●	●		W		∗	
2011	GBASE[78]	D	●			M	●		●				●		●			?	∗	∗	●			W		∗	
2011	HipG[86]	D				V		●	○	○	●					●	●				●	●	●	D			●
2011	DisNet[90]	D				V	●		●				●				●		●	●				W		●	
2011	Menthor[56]	D			●	V	●		●				●				●				●	●		D	●		
2012	Trinity[130]	D				G	○	○	○	○			?	?		●		∗	●		●	●		D	●		●
2012	Maiter[167, 168]	D				V	●		○	○	●	●	●			●	●		●		●			W	●		
2012	Kineograph[31]	D				V	○	○	●		●		●			●	●		●		●	●		D	●		
2012	Mizan[82]	D			●	V	●		●		●		●	●		●	●	∗	●		●	●		D	●		
2012	GraphGPU[132, 133]	D		●		M	●		●				●			?					●			W		●	
2012	Giraph[7]	D	●		●	V	●		●		●		●			●	●	?	∗	∗	●	●	●	W		∗	
2012	Distributed GraphLab[92]	D				V	○	○	○	○	●	●	●			●			●		●	●		W			●
2012	PowerGraph[51]	D				V	○	○	○	○	●		●		●		●		●		●	●		W			●
2013	GPS[124]	D			●	V	●		●		●		●	○		●	○	∗	●		●	●	●	W	●		
2013	X-Pregel (ScaleGraph)[12]	D			●	V	●		●		●		●	●		●	●	∗			●	●		W	●		
2013	Giraph++[143]	D			●	GV	○	○	○	○	●		●			●	●	?	●		●	●	●	W	●		●
2013	Giraphx[141]	D			●	V		●		●	●		●			●	●	?	●		●	●	●	W			●
2013	G2[172]	D		●	●	V	●		●				●		●				●		●		●	W	●		
2013	Titan-Hadoop/Faunus[10]	D	●		●	V	●		●		●		●			●	●	∗		∗	●	●	●	W		●	
2013	Presto[148]	D				M	●		●				●	●		●	●	∗	●		●	●	●	W		●	
2013	GraphX[158]	D				G	●		●				●		●		●	∗	●		●	●	●	W		●	
2013	GRE[160]	D				V	●		●		●		●		●		●		●		●			W	●		
2013	LFGraph[69]	D				V		●	●				●			●	●				●			W			●
2013	Pregelix[18]	D	●		●	V	●		●		●		●			●	●	?	∗	∗	●	●	●	W		∗	
2013	PAGE[131]	D	●		●	V	●		●		●		●			●	●	?	∗	∗	●	●	●	W		∗	
2013	BC-BSP[13]	D	●		●	V	●		●		●		●			●	●		●		●	●		W	●		
2013	PowerLyra[28]	D				V		●	●		●		●		●				●		●	●		W			●
2013	PowerSwitch[156]	D				V		●	●	●	●		●		●		●		●		●	●		W			●
2014	GoFFish[137]	D				G		●		●	●		●			●	●				●			W	●		●
2014	Cyclops[27]	D				V		●	●		●		●			●	●		●		●	●		W	●		●
2014	Imitator[152]	D				V	●		●		●		●			●	●			●	●	●		W	●		
2014	GraphHP[26]	D				V	●			●	●		●			●	●		●		●	●		W	●		
2014	Chronos[59]	D				V	○	○	●				●			●	●				●			W	●		●
2014	Seraph[159]	D			●	V	●		●				●			?	●			●	●			W	●		
2010	Signal/Collect[138]	S				V	●		○	○	○	●	●								●			W	●		
2010	GraphLab[93]	S				V		●	○	○	●	●	●								●	●		W			●
2012	GraphChi[87]	S	●		●	V		●		●	●		●			●					●	●		W			●
2012	Grace[117]	S				V	?			●	●		●			●	●	●			●	?		W	●		
2012	TOTEM[48]	S		●		G	●			●						●					●	●	●	W	●		●
2013	Medusa[171]	S		?	●	V	●		●		●		●			●					●		●	W	●		
2013	RASP[161]	S	●	?		V		●		●	●		●			●					●			W			●
2013	X-Stream[122]	S	●		●	V	●		●				●		●		●				●	●		W	●		
2013	GRACE @ Cornell[151, 157]	S				GV	●		○	○	●	●	●			●	●				●		●	W	●		
2013	Galois[109]	S	●		●	V	○	○		●	●	●	●			●	●				●		●	W			●
2013	Ligra[134]	S				G	●			●	●										●	●	●	W			●
2013	TurboGraph[58]	S	●			M	●		●				●			●					●			W		∗	
2014	PathGraph[163]	S	●			V		●		●			●		?						●			W			●
2014	BPP[106]	S	●		●	V		●		●	●		●			●					●	●		W			●
2014	FlashGraph[169]	S	●	?		V	○	○		●	●		●			●	●	●			●			W	●		●
2014	GraphGen[110]	S		?		V		●	●		●		●								●			W			●
2014	CuSHa[83]	S		?		V		●		●						●					●			W			●
2014	MPGraph/MapGraph[46]	S		?		V		●	●		●		●					●			●			W			●
2014	GasCL[25]	S		●		V	●		●		●		●								●	●		W	●		

Table 1: Taxonomy of parallel graph processing frameworks
● explicit support — ○ explicit support, user choice — ∗ implicit support

10 Discussion

Several questions remain unanswered and pose interesting future research directions. For example, *how effective are the currently proposed programming models*? Elser and Montresor [42] note that vertex-centric programming interfaces (from Hama[129], GraphLab[92, 93], Giraph[7]) are a better fit for graph problems than the general-purpose interfaces from Hadoop[33] (MapReduce) and Stratosphere[2]. However, it is unclear what the performance implications are for arbitrary algorithms, as efficient implementation is still up to the user. Lin and Schatz [91] present design patterns for efficient graph algorithms in MapReduce that achieve an improvement in running time by 69% for PageRank.

Salihoglu and Widom [126] argue that efficient implementation of graph algorithms on Pregel-like systems is still surprisingly challenging. Nguyen et al. [109] show that similar programming models (GraphLab, PowerGraph[51], Ligra[134]) perform differently for varying algorithms and input graphs. The authors argue that this is inherent to the models' restrictiveness and not their implementation. A high-level domain-specific language (Section 3.5) such as Green-Marl[66] is promising in this regard, as it allows frameworks to offer optimized primitives, rather than relying on the user for optimal implementation. HelP[125] takes a similar approach by exposing a set of high-level primitives for graph processing on top of GraphX[158].

For most of our surveyed systems, it is not clear what kind of algorithms can be expressed efficiently. Algorithms like PageRank and SSSP map naturally to Vertex-Centric programming models and are often used for performance comparisons in literature, but it would be interesting to see more challenging algorithms taken into account as well (such as strongly connected components or minimum spanning forest [126]).

Proper benchmarks could be used to answer the above and other performance-related questions, e.g. *are standalone frameworks required or can general-purpose frameworks be extended?*, *when is a single machine sufficient?*, *to what extent can heterogeneous architectures be exploited?*. Little is known about how systems relate in terms of performance. Several attempts at a comparative performance evaluation have been made ([42, 55, 57, 76]), but only with a very small subset of the frameworks, due to the considerable amount of effort required. In general, the outcome is that there is no "best" framework (although MapReduce frameworks generally perform worse for graph applications). Giving insight into these performance characteristics would make framework selection easier.

More insight in performance characteristics for frameworks, graphs, and algorithms should also enable predictions for resource requirements. Answers to questions like *what architecture is best suited for this application?* or *how many machines are needed to process this graph?* are currently unknown, but can benefit a large number of users.

Next to the challenges in evaluating built systems, there are still challenges in designing new frameworks as well. Typically, graph databases do not support offline batch processing of graphs (Section 3.3), but *does there have to be a distinction between graph data management and graph analysis*? Performance of algorithms greatly depend on proper input partitioning, which is one of the key elements of a DBMS.

Can multiple applications work on a shared graph structure? When graph structure remains largely static during computation, it is unnecessarily expensive to load the input data more than once. Xue et al. [159] (Seraph) decouple graph structure from job data to allow multiple concurrent jobs.

Finally, *can graph frameworks facilitate efficient analysis on data streams?* We notice that a lot of frameworks focus on the volume aspect of big data, but ignore the challenge in velocity (Section 2.2). A lot of data is constantly changing and continuous bulk analysis of the entire input causes significant overhead, especially when the majority of the result does not change. Kineograph[31] and Chronos[59] allow for incremental processing of snapshots. On top of that, Chronos enables processing of historical data, which is another interesting concept for analysis.

11 Conclusion

Graphs are a powerful abstraction mechanism for representing relationships between data entities. Popular in many branches of science and industry, graphs keep growing in size and efficiently processing them remains challenging.

In this survey, we have showed that frameworks have to deal with challenges inherent to graph applications on top of general challenges in big data and parallel computing. Graph computations are typically data-driven, while graphs have an irregular structure and size that can exceed the memory of a single machine, resulting in poor data locality and nontrivial parallelization.

We have presented a taxonomy that classifies over 80 parallel graph processing systems based on distinctive features related to the Platform, Programming Model, Communication Model, Execution Model, and Framework Abstraction. We concluded that most literature focuses on the value of complementary features, rather than taking drastically different approaches. A lot of systems are alike from a top-level point of view, but differ in implementation. There does not appear to be a "best" combination of features.

Proper benchmarks, an open research challenge, can be valuable in comparing frameworks based on other aspects, such as performance, usability, and stability. Currently, it is not clear what kind of algorithms can be expressed efficiently by most frameworks. Performance characteristics differ based on architecture, programming model, algorithm implementation, and input data. High-level domain-specific languages and framework optimized primitives have potential in this regard. Performance prediction, another open challenge, should ease choosing the proper execution platform.

Finally, there are still open research challenges for framework design. From the initially listed challenges in graph processing, the velocity of big data is not taken into account most often. In this respect, frameworks might use a shared graph structure for multiple applications to reduce memory overhead or facilitate the processing of data streams.

References

[1] Aingworth, D., Chekuri, C., Indyk, P., and Motwani, R. 1999. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*. 6

[2] Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., and Warneke, D. 2014. The Stratosphere platform for big data analytics. *The VLDB Journal*. 8, 9, 13, 15, 16, 17, 18

[3] Alon, N., Babai, L., and Itai, A. 1986. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms 7,* 4 (Dec.), 567–583. 6

[4] Amaral, L. A., Scala, A., Barthelemy, M., and Stanley, H. E. 2000. Classes of small-world networks. *Proceedings of the National Academy of Sciences of the United States of America 97,* 21 (Oct.), 11149–52. 4

[5] Angles, R. 2012. A Comparison of Current Graph Database Models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pp. 171–177. IEEE. 6

[6] Apache Software Foundation 2011. Bagel. https://spark.apache.org/docs/latest/bagel-programming-guide.html. 16, 17

[7] Apache Software Foundation 2012. Giraph. https://giraph.apache.org/. 16, 17, 18

[8] Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., and Rabkin, A. 2010. A view of cloud computing. *Communications of the ACM 53,* 4 (April), 50. 8

[9] Aurelius 2012. Titan. https://thinkaurelius.github.io/titan/. 6

[10] Aurelius 2013. Faunus. https://github.com/thinkaurelius. 17

[11] Bader, D. A., Amos-binks, A., Berry, J., Chavarr, D., Hastings, C., and Poulos, S. C. 2009. STINGER : Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. Technical report. 17

[12] Bao, N. and Suzumura, T. 2013. Towards highly scalable pregel-based graph processing platform with x10. In *WWW '13 Companion Proceedings of the 22nd international conference on World Wide Web companion*, pp. 501–508. 14, 17

[13] Bao, Y., Zhigang, W., Yu, G., Ge, Y., Fangling, L., Hongxu, Z., Bairen, C., Chao, D., and Leitao, G. 2013. BC-BSP: A BSP-Based Parallel Iterative Processing System for Big Data on Cloud Architecture. In B. Hong, X. Meng, L. Chen, W. Winiwarter, and W. Song (Eds.), *Database Systems for ...*, Volume 7827 of *Lecture Notes in Computer Science*, pp. 31–45. Berlin, Heidelberg: Springer Berlin Heidelberg. 16, 17

[14] Barabási, A. 1999. Emergence of Scaling in Random Networks. *Science 286,* 5439 (Oct.), 509–512. 6, 10, 14

[15] Batagelj, V. and Brandes, U. 2005. Efficient generation of large random networks. *Physical Review E 71,* 3 (March), 036113. 6

[16] Berry, J. W., Hendrickson, B., Kahan, S., and Konecny, P. 2007. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–14. IEEE. 10, 17

[17] Borkar, V., Carey, M., Grover, R., Onose, N., and Vernica, R. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*, pp. 1151–1162. IEEE. 9, 16, 17

[18] Bu, Y. 2013. Pregelix. In *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*, New York, New York, USA, pp. 1–2. ACM Press. 17

[19] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. 2010. HaLoop. *Proceedings of the VLDB Endowment 3,* 1-2 (Sept.), 285–296. 8, 11, 13, 15, 16, 17

[20] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. 2012. The HaLoop approach to large-scale iterative data analysis. *The VLDB Journal 21,* 2 (March), 169–190. 9

[21] Buluc, A. and Gilbert, J. R. 2011. The Combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications 25,* 4 (May), 496–509. 6, 9, 10, 17

[22] Buluc, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C. 2013. Recent Advances in Graph Partitioning. 14

[23] Cardoso, J. C. S., Baquero, C., and Almeida, P. S. 2009. Probabilistic Estimation of Network Size and Diameter. *2009 Fourth Latin-American Symposium on Dependable Computing*, 33–40. 6

[24] Chandrasekhar, M. and Prakasam, K. 2010. JPregel. https://kowshik.github.io/JPregel/. 16, 17

[25] Che, S. 2014. GasCL: A Vertex-Centric Graph Model for GPUs. Technical report. 9, 16, 17

[26] Chen, Q., Bai, S., Li, Z., Gou, Z., Suo, B., and Pan, W. 2014. GraphHP: A Hybrid Platform for Iterative Graph Processing. Technical report. 17

[27] Chen, R., Ding, X., Wang, P., Chen, H., Zang, B., and Guan, H. 2014. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, New York, New York, USA, pp. 215–226. ACM Press. 17

[28] Chen, R., Shi, J., Chen, Y., and Guan, H. 2013. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. Technical report. 17

[29] Chen, R., Weng, X., He, B., and Yang, M. 2010. Large graph processing in the cloud. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, New York, New York, USA, pp. 1123. ACM Press. 6, 8, 14, 17

[30] Chen, R., Yang, M., Weng, X., Choi, B., He, B., and Li, X. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, New York, New York, USA, pp. 1–13. ACM Press. 17

[31] Cheng, R., Chen, E., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., and Zhao, F. 2012. Kineograph. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, New York, New York, USA, pp. 85. ACM Press. 6, 12, 17, 18

[32] Ciglan, M., Averbuch, A., and Hluchy, L. 2012. Benchmarking Traversal Operations over Graph Databases. *2012 IEEE 28th International Conference on Data Engineering Workshops*, 186–189. 6

[33] Cutting, D., Cafarella, M., and Apache Software Foundation 2005. Hadoop. https://hadoop.apache.org/. 7, 17, 18

[34] de Solla Price, D. J. 1965. Networks of Scientific Papers. *Science 149,* 3683 (July), 510–515. 4

[35] Dean, J. and Ghemawat, S. 2008. MapReduce. *Communications of the ACM 51,* 1 (Jan.), 107. 7, 8, 9, 10, 14, 15, 16, 17

[36] DeWitt, D. and Gray, J. 1992. Parallel database systems: the future of high performance database systems. *Communications of the ACM 35,* 6 (June), 85–98. 14

[37] Du, N., Wu, B., Xu, L., Wang, B., and Pei, X. 2006. A Parallel Algorithm for Enumerating All Maximal Cliques in Complex Network. *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, 320–324. 6

[38] Ediger, D., McColl, R., Riedy, J., and Bader, D. A. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5. IEEE. 17

[39] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. 2010. Twister. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, New York, New York, USA, pp. 810. ACM Press. 8, 9, 11, 13, 15, 16, 17

[40] Elnikety, E., Elsayed, T., and Ramadan, H. E. 2011. iHadoop: Asynchronous Iterations for MapReduce. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pp. 81–90. IEEE. 13, 14, 16, 17

[41] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34,* 3 (Sept.), 375–408. 15

[42] Elser, B. and Montresor, A. 2013. An evaluation study of BigData frameworks for graph processing. *2013 IEEE International Conference on Big Data*, 60–67. 18

[43] Facebook Inc. 2014. Facebook Reports Second Quarter 2014 Results. 4

[44] Fidel, A., Amato, N. M., and Rauchwerger, L. 2014. KLA. In *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14*, New York, New York, USA, pp. 27–38. ACM Press. 11

[45] Fortunato, S. 2010. Community detection in graphs. *Physics Reports 486,* 3-5 (Feb.), 75–174. 6

[46] Fu, Z., Personick, M., and Thompson, B. 2014. MapGraph. In *Proceedings of Workshop on GRAph Data management Experiences and Systems - GRADES'14*, New York, New York, USA, pp. 1–6. ACM Press. 8, 16, 17

[47] Garey, M. R., Johnson, D. S., and Stockmeyer, L. 1974. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC '74*, New York, New York, USA, pp. 47–63. ACM Press. 5, 14

[48] Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., and Ripeanu, M. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, New York, New York, USA, pp. 345. ACM Press. 9, 12, 13, 14, 17

[49] Gharaibeh, A., Costa, L. B., Santos-Neto, E., and Ripeanu, M. 2013. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 851–862. IEEE. 8

[50] Gharaibeh, A., Santos-Neto, E., Costa, L. B., and Ripeanu, M. 2013. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. 11

[51] Gonzalez, J., Low, Y., Gu, H., Bickson, D., and Guestrin, C. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 17–30. 10, 11, 12, 13, 14, 17, 18

[52] Gonzalez, J., Low, Y., and Guestrin, C. 2009. Residual splash for optimally parallelizing belief propagation. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Volume 5, pp. 177–184. 12

[53] Gregor, D. and Lumsdaine, A. 2005. The parallel BGL: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, pp. 1–18. 6, 9, 10, 11, 16, 17

[54] Guo, Y. and Biczak, M. 2013. Towards benchmarking graph-processing platforms. Technical report. 6

[55] Guo, Y., Biczak, M., Varbanescu, A. L., Iosup, A., Martella, C., and Willke, T. L. 2014. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 395–404. IEEE. 6, 18

[56] Haller, P. and Miller, H. 2011. Parallelizing Machine LearningFunctionally. In *Scala Workshop*, Stanford CA, USA. 15, 17

[57] Han, M., Daudjee, K., Ammar, K., and Ozsu, M. 2014. An Experimental Comparison of Pregel-like Graph Processing Systems. In *Proceedings of the VLDB Endowment*, Number i, pp. 1047–1058. 18

[58] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. 2013. TurboGraph. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, New York, New York, USA, pp. 77. ACM Press. 9, 14, 17

[59] Hant, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. 2014. Chronos. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, New York, New York, USA, pp. 1–14. ACM Press. 5, 11, 12, 15, 17, 18

[60] Harish, P. and Narayanan, P. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna (Eds.), *High performance computingHiPC 2007*, Volume 4873 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 197–208. Springer Berlin Heidelberg. 6

[61] Harshvardhan, Fidel, A., Amato, N. M., and Rauchwerger, L. 2012. The STAPL Parallel Graph Library. In H. Kasahara and K. Kimura (Eds.), *Languages and Compilers for Parallel Computing*, Volume 7760 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg. 10, 17

[62] Hendrickson, B. and Berry, J. W. 2008. Graph Analysis with High-Performance Computing. *Computing in Science & Engineering 10,* 2 (March), 14–19. 5

[63] Hendrickson, B. and Kolda, T. G. 2000. Graph partitioning models for parallel computing. *Parallel Computing 26,* 12 (Nov.), 1519–1534. 14

[64] Hielscher, F. and Gottschling, P. 2004. ParGraph. http://pargraph.sourceforge.net/. 10, 16, 17

[65] Holzschuher, F. and Peinl, R. 2013. Performance of graph query languages. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops on - EDBT '13*, New York, New York, USA, pp. 195. ACM Press. 6

[66] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. 2012. Green-Marl. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, New York, New York, USA, pp. 349. ACM Press. 7, 10, 18

[67] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, New York, New York, USA, pp. 267. ACM Press. 6

[68] Hong, S., Salihoglu, S., Widom, J., and Olukotun, K. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '14*, New York, New York, USA, pp. 208–218. ACM Press. 7, 10

[69] Hoque, I. and Gupta, I. 2013. LFGraph: Simple and Fast Distributed Graph Analytics. Technical report. 10, 11, 12, 17

[70] Huberman, B. A. 2001. *The Laws of the Web: Patterns in the Ecology of Information*. MIT Press Cambridge. 4

[71] Ibarra, L. and Richards, D. 1993. Efficient parallel graph algorithms based on open ear decomposition. *Parallel Computing 19,* 8 (Aug.), 873–886. 6

[72] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 - EuroSys '07*, New York, New York, USA, pp. 59. ACM Press. 8, 14, 15, 17

[73] Jeong, H., Mason, S. P., Barabási, A. L., and Oltvai, Z. N. 2001. Lethality and centrality in protein networks. *Nature 411,* 6833 (May), 41–2. 4

[74] Jiang, D., Chen, G., and Ooi, B. 2014. epiC: an Extensible and Scalable System for Processing Big Data. In *Proceedings of the VLDB Endowment*, Volume 7. 7, 8, 9, 13, 16, 17

[75] Johnson, R. E. and Foote, B. 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming 1,* 2, 22–35. 6, 13

[76] Kajdanowicz, T., Kazienko, P., and Indyk, W. 2014. Parallel processing of large graphs. *Future Generation Computer Systems 32*, 324–337. 18

[77] Kameda, H., Li, J., Kim, C., and Zhang, Y. 2011. *Optimal Load Balancing in Distributed Computer Systems*. Springer Publishing Company, Incorporated. 15

[78] Kang, U., Tong, H., Sun, J., Lin, C.-Y., and Faloutsos, C. 2012. Gbase: an Efficient Analysis Platform for Large Graphs. *The VLDB Journal 21,* 5 (June), 637–650. 7, 9, 16, 17

[79] Kang, U., Tsourakakis, C., and Appel, A. 2008. *HADI: Fast diameter estimation and mining in massive graphs with Hadoop*. 6

[80] Kang, U., Tsourakakis, C. E., and Faloutsos, C. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *2009 Ninth IEEE International Conference on Data Mining*, pp. 229–238. IEEE. 7, 8, 9, 16, 17

[81] Kang, U., Tsourakakis, C. E., and Faloutsos, C. 2011. PEGASUS: Mining peta-scale graphs. *Knowledge and Information Systems 27*, 303–325. 17

[82] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. 2013. Mizan. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, New York, New York, USA, pp. 169. ACM Press. 14, 17

[83] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. 2014. CuSha. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, New York, New York, USA, pp. 239–252. ACM Press. 8, 13, 14, 16, 17

[84] Kojima, K. 2009. Thrust Graph Library. https://code.google.com/p/thrust-graph/. 17

[85] Korf, R. and Schultze, P. 2005. Large-scale parallel breadth-first search. *AAAI*, 1380–1385. 6

[86] Krepska, E., Kielmann, T., Fokkin, W., and Bal, H. 2011. HipG. *ACM SIGOPS Operating Systems Review 45,* 2 (July), 3. 13, 15, 17

[87] Kyrola, A., Blelloch, G., and Guestrin, C. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 31–46. 7, 8, 12, 14, 17

[88] Laney, D. 2001. 3D data management: Controlling data volume, velocity and variety. Technical Report February 2001. 5

[89] Leskovec, J., Kleinberg, J., and Faloutsos, C. 2005. Graphs over time. In *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05*, New York, New York, USA, pp. 177. ACM Press. 4, 6

[90] Lichtenwalter, R. and Chawla, N. V. 2011. DisNet: A Framework for Distributed Graph Computation. In *2011 International Conference on Advances in Social Networks Analysis and Mining*, pp. 263–270. IEEE. 17

[91] Lin, J. and Schatz, M. 2010. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs - MLG '10*, New York, New York, USA, pp. 78–85. ACM Press. 18

[92] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. 2012. Distributed GraphLab. *Proceedings of the VLDB Endowment 5,* 8 (April), 716–727. 5, 11, 12, 13, 15, 17, 18

[93] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. 2010. GraphLab: A New Framework for Parallel Machine Learning. Technical report (June). 7, 10, 11, 13, 15, 17, 18

[94] Luby, M. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing 15,* 4, 1036–1053. 6

[95] Lugowski, A., Alber, D., and Buluç, A. 2012. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In J. Ghosh, H. Liu, I. Davidson, C. Domeniconi, and C. Kamath (Eds.), *Proceedings of the 2012 SIAM International Conference on Data Mining*, Philadelphia, PA. Society for Industrial and Applied Mathematics. 17

[96] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters 17,* 01 (March), 5–20. 5

[97] Luo, L., Wong, M., and Hwu, W.-m. 2010. An effective GPU implementation of breadth-first search. *Proceedings of the 47th Design Automation Conference on - DAC '10*, 52. 6

[98] Mahoney, M. W., Lim, L., and Carlsson, G. E. 2008. Algorithmic and statistical challenges in modern largescale data analysis are the focus of MMDS 2008. *ACM SIGKDD Explorations Newsletter 10,* 2 (Dec.), 57. 4

[99] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. 2010. Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, New York, New York, USA, pp. 135. ACM Press. 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17

[100] McColl, R. C., Ediger, D., Poovey, J., Campbell, D., and Bader, D. A. 2014. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications - PPAA '14*, New York, New York, USA, pp. 11–18. ACM Press. 6

[101] Merrill, D., Garland, M., and Grimshaw, A. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*, New York, New York, USA, pp. 117. ACM Press. 6

[102] Meusel, R., Vigna, S., Lehmberg, O., and Bizer, C. 2014. Graph Structure in the Web Revisited: A Trick of the Heavy Tail. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pp. 427–432. 4, 5

[103] Michael, M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. 2007. Scale-up x Scale-out: A Case Study using Nutch/Lucene. *2007 IEEE International Parallel and Distributed Processing Symposium*, 1–8. 7, 8

[104] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. 2013. Naiad. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, New York, New York, USA, pp. 439–455. ACM Press. 7, 16, 17

[105] Murray, D. G., Smowton, C., and Hand, S. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI'11 Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pp. 113–126. 8, 9, 16, 17

[106] Najeebullah, K., Khan, K. U., Nawaz, W., and Lee, Y.-K. 2014. BPP: Large Graph Storage for Efficient Disk Based Processing. 14, 17

[107] Neo Technology 2007. Neo4j. http://neo4j.com/. 6

[108] Newman, M. E. J. 2003. The Structure and Function of Complex Networks. *SIAM Review 45,* 2 (Jan.), 167–256. 4

[109] Nguyen, D., Lenharth, A., and Pingali, K. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, New York, New York, USA, pp. 456–471. ACM Press. 7, 11, 13, 16, 17, 18

[110] Nurvitadhi, E., Weisz, G., Wang, Y., Hurkat, S., Nguyen, M., Hoe, J. C., Martinez, J. F., and Guestrin, C. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28. IEEE. 17

[111] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. 2008. Pig latin. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, New York, New York, USA, pp. 1099. ACM Press. 7

[112] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. 2008. GPU Computing. *Proceedings of the IEEE 96,* 5 (May), 879–899. 8

[113] Padgett, J. F. and Ansell, C. K. 1993. Robust Action and the Rise of the Medici, 1400-1434. *American Journal of Sociology 98,* 6 (May), 1259. 4

[114] Page, L., Brin, S., Motwani, R., and Winograd, T. 1999. The PageRank citation ranking: Bringing order to the web. Technical report. 5

[115] Plimpton, S. J. and Devine, K. D. 2011. MapReduce in MPI for Large-scale graph algorithms. *Parallel Computing 37,* 9 (Sept.), 610–632. 8, 11, 15, 16, 17

[116] Power, R. and Li, J. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, pp. 1–14. 11, 13, 15, 17

[117] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haridasan, M. 2012. Managing Large Graphs on Multi-Cores With Graph Awareness. In *USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pp. 4–4. 6, 14, 17

[118] Quinn, M. J. and Deo, N. 1984. Parallel graph algorithms. *ACM Computing Surveys 16,* 3 (Sept.), 319–348. 6

[119] Rapoport, A. and Horvath, W. J. 2007. A study of a large sociogram. *Behavioral Science 6,* 4 (Jan.), 279–291. 4

[120] Ravel 2011. GoldenOrb. https://github.com/jzachr/goldenorb. 16, 17

[121] Red Hat Enterprise Linux 2014. Red Hat Enterprise Linux technology capabilities and limits. https://access.redhat.com/articles/rhel-limits. 7

[122] Roy, A., Mihailovic, I., and Zwaenepoel, W. 2013. X-Stream. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, New York, New York, USA, pp. 472–488. ACM Press. 10, 14, 17

[123] Sabrin, K., Lin, Z., Chau, D., Lee, H., and Kang, U. 2013. MMAP: Mining Billion-Scale Graphs on a PC with Fast, Minimalist Approach via Memory Mapping. Technical Report October 2013. 17

[124] Salihoglu, S. and Widom, J. 2013. GPS. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management - SSDBM*, New York, New York, USA, pp. 1. ACM Press. 7, 14, 15, 16, 17

[125] Salihoglu, S. and Widom, J. 2014a. HelP. In *Proceedings of Workshop on GRAph Data management Experiences and Systems - GRADES'14*, New York, New York, USA, pp. 1–6. ACM Press. 18

[126] Salihoglu, S. and Widom, J. 2014b. Optimizing graph algorithms on pregel-like systems. In *40th International Conference on Very Large Data Bases*, Number c, pp. 577–588. 18

[127] Schaeffer, S. E. 2007. Graph clustering. *Computer Science Review 1,* 1 (Aug.), 27–64. 6

[128] Schloegel, K., Karypis, G., and Kumar, V. 2003. Graph partitioning for high-performance scientific simulations. In *Sourcebook of parallel computing*, pp. 491–541. 14

[129] Seo, S., Yoon, E. J., Kim, J., Jin, S., Kim, J.-s., and Maeng, S. 2010. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 721–726. IEEE. 17, 18

[130] Shao, B., Wang, H., and Li, Y. 2013. Trinity. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, New York, New York, USA, pp. 505. ACM Press. 4, 6, 9, 12, 17

[131] Shao, Y., Yao, J., Cui, B., and Ma, L. 2013. PAGE. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13*, New York, New York, USA, pp. 823–828. ACM Press. 16, 17

[132] Shirahata, K., Sato, H., Suzumura, T., and Matsuoka, S. 2012. A GPU Implementation of Generalized Graph Processing Algorithm GIM-V. In *2012 IEEE International Conference on Cluster Computing Workshops*, pp. 207–212. IEEE. 17

[133] Shirahata, K., Sato, H., Suzumura, T., and Matsuoka, S. 2013. A Scalable Implementation of a MapReduce-based Graph Processing Algorithm for Large-Scale Heterogeneous Supercomputers. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 277–284. IEEE. 17

[134] Shun, J. and Blelloch, G. E. 2013. Ligra. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13*, New York, New York, USA, pp. 135. ACM Press. 13, 15, 17, 18

[135] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. 2010. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 1–10. 15

[136] Siek, J., Lee, L.-Q., and Lumsdaine, A. 2002. *The Boost Graph Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 10, 16

[137] Simmhan, Y., Kumbhare, A., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C., and Prasanna, V. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In F. Silva, I. Dutra, and V. Santos Costa (Eds.), *Euro-Par 2014 Parallel Processing*, Volume 8632 of *Lecture Notes in Computer Science*, Cham, pp. 451–462. Springer International Publishing. 9, 10, 11, 17

[138] Stutz, P., Bernstein, A., and Cohen, W. 2010. Signal/Collect: Graph Algorithms for the (Semantic) Web. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm (Eds.), *The Semantic WebISWC 2010*, Volume 6496 of *Lecture Notes in Computer Science*, pp. 764–780. Berlin, Heidelberg: Springer Berlin Heidelberg. 7, 17

[139] Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., Olukotun, K., Rompf, T., and Odersky, M. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, Number Ml. 7

[140] Suresh, A. 2010. Phoebus. https://github.com/xslogic/phoebus. 16, 17

[141] Tasci, S. and Demirbas, M. 2013. Giraphx: Parallel Yet Serializable Large-Scale Graph Processing. In F. Wolf, B. Mohr, and D. an Mey (Eds.), *Euro-Par 2013 Parallel Processing*, Volume 8097 of *Lecture Notes in Computer Science*, pp. 458–469. Berlin, Heidelberg: Springer Berlin Heidelberg. 12, 16, 17

[142] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. 2009. Hive. In *Proceedings of the VLDB Endowment*, Volume 2, pp. 1626–1629. 7

[143] Tian, Y., Balmin, A., and Corsten, S. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment 7,* 3, 193–204. 9, 10, 12, 16, 17

[144] Treaster, M. 2005. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. *ACM Computing Research Repository (CoRR) 501002*, 1–11. 15

[145] Ugander, J., Karrer, B., Backstrom, L., and Marlow, C. 2011. The Anatomy of the Facebook Social Graph. 4

[146] Valiant, L. G. 1990. A bridging model for parallel computation. *Communications of the ACM 33,* 8 (Aug.), 103–111. 11

[147] van Steen, M. and Tanenbaum, A. S. 2001. *Distributed systems: principles and paradigms*, Volume 40. 5, 15

[148] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A., and Schreiber, R. S. 2013. Presto. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, New York, New York, USA, pp. 197. ACM Press. 9, 17

[149] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. 2010. A comparison of a graph database and a relational database. In *Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10*, New York, New York, USA, pp. 1. ACM Press. 6

[150] Wadsworth, D. M. and Chen, Z. 2008. Performance of MPI broadcast algorithms. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–7. 15

[151] Wang, G., Xie, W., Demers, A., and Gehrke, J. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*. 17

[152] Wang, P., Zhang, K., Chen, R., Chen, H., and Guan, H. 2014. Replication-based Fault-tolerance for Large-scale Graph Processing. In *The Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. 15, 17

[153] Warneke, D. and Kao, O. 2009. Nephele. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers - MTAGS '09*, New York, New York, USA, pp. 1–10. ACM Press. 17

[154] Watts, D. J. and Strogatz, S. H. 1998. Collective dynamics of small-world networks. *Letters to Nature 393,* June, 440–442. 4

[155] White, J. G., Southgate, E., Thomson, J. N., and Brenner, S. 1986. The Structure of the Nervous System of the Nematode Caenorhabditis elegans. *Philosophical Transactions of the Royal Society B: Biological Sciences 314,* 1165 (Nov.), 1–340. 4

[156] Xie, C., Chen, R., Guan, H., Zang, B., and Chen, H. 2013. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. Technical report. 12, 17

[157] Xie, W., Wang, G., Bindel, D., Demers, A., and Gehrke, J. 2013. Fast iterative graph computation with block updates. *Proceedings of the VLDB Endowment 6,* 14 (Sept.), 2014–2025. 17

[158] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. 2013. GraphX. In *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, New York, New York, USA, pp. 1–6. ACM Press. 7, 16, 17, 18

[159] Xue, J., Yang, Z., Qu, Z., Hou, S., and Dai, Y. 2014. Seraph. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, New York, New York, USA, pp. 227–238. ACM Press. 5, 15, 17, 18

[160] Yan, J., Tan, G., and Sun, N. 2013. GRE: A Graph Runtime Engine for Large-Scale Distributed Graph-Parallel Applications. *CoRR*, 12. 10, 14, 17

[161] Yoneki, E. and Roy, A. 2013. Scale-up graph processing. In *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, New York, New York, USA, pp. 1–6. ACM Press. 8, 16, 17

[162] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 1–14. 17

[163] Yuan, P., Zhang, W., Xie, C., Jin, H., Liu, L., and Lee, K. 2014. Fast Iterative Graph Computation: A Path Centric Approach. In *SC 2014*. 14, 17

[164] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. 2010. Spark : Cluster Computing with Working Sets. In *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10. 7, 9, 15, 16, 17

[165] Zhang, Y., Gao, Q., Gao, L., and Wang, C. 2011. PrIter. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, New York, New York, USA, pp. 1–14. ACM Press. 16, 17

[166] Zhang, Y., Gao, Q., Gao, L., and Wang, C. 2012a. iMapReduce: A Distributed Computing Framework for Iterative Computation. *Journal of Grid Computing 10,* 1 (March), 47–68. 9, 11, 13, 14, 15, 16, 17

[167] Zhang, Y., Gao, Q., Gao, L., and Wang, C. 2012b. Maiter: A message-passing distributed framework for accumulative iterative computation. Technical report. 11, 12, 17

[168] Zhang, Y., Gao, Q., Gao, L., and Wang, C. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems 25,* 8 (Aug.), 2091–2100. 12, 13, 17

[169] Zheng, D., Mhembere, D., Burns, R., and Szalay, A. S. 2014. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. 8, 9, 14, 16, 17

[170] Zheng, J., Chen, W., Chen, Y., and Zhang, Y. 2008. Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In *2008 13th Asia-Pacific Computer Systems Architecture Conference*, pp. 1–8. IEEE. 6

[171] Zhong, J. and He, B. 2012. An overview of Medusa. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*, New York, New York, USA, pp. 283. ACM Press. 8, 17

[172] Zhong, J. and He, B. 2013. Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, pp. 9–16. IEEE. 8, 16, 17

[173] Zhong, J. and He, B. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems 25,* 6 (June), 1543–1552. 15