Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges

SAFIOLLAH HEIDARI, The University of Melbourne
YOGESH SIMMHAN, Indian Institute of Science
RODRIGO N. CALHEIROS, The University of Melbourne
RAJKUMAR BUYYA, The University of Melbourne

The world is becoming a more conjunct place and the number of data sources such as social networks, online transactions, web search engines, and mobile devices is increasing even more than had been predicted. A large percentage of this growing dataset exists in the form of linked data, more generally, graphs, and of unprecedented sizes. While today's data from social networks contain hundreds of millions of nodes connected by billions of edges, inter-connected data from globally distributed sensors that forms the Internet of Things can cause this to grow exponentially larger. Although analyzing these large graphs is critical for the companies and governments that own them, big data tools designed for text and tuple analysis such as MapReduce cannot process them efficiently. So, graph distributed processing abstractions and systems are developed to design iterative graph algorithms and process large graphs with better performance and scalability. These graph frameworks propose novel methods or extend previous methods for processing graph data. In this article, we propose a taxonomy of graph processing systems and map existing systems to this classification. This captures the diversity in programming and computation models, runtime aspects of partitioning and communication, both for in-memory and distributed frameworks. Our effort helps to highlight key distinctions in architectural approaches, and identifies gaps for future research in scalable graph systems.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Theory of computation** → **Distributed computing models**; *Design and analysis of algorithms*; *Distributed algorithms*; • **Computing methodologies** → **Parallel computing methodologies**; **Distributed computing methodologies**; • **Mathematics of computing** → *Graph algorithms*;

Additional Key Words and Phrases: Big data, graph processing, large-scale graphs, parallel processing, distributed systems

ACM Reference format:
Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3, Article 60 (June 2018), 53 pages.
https://doi.org/10.1145/3199523

This article was partially supported by the Australian Research Council (ARC) Future Fellowship project.

Authors' addresses: S. Heidari, R. N. Calheiros, and R. Buyya, The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia; emails: sheidari@student.unimelb.edu.au, rodrigo.calheiros@gmail.com, rbuyya@unimelb.edu.au; Y. Simmhan, Department of Computational and Data Sciences, Indian Institute of Science (IISc), Bangalore, India; email: simmhan@cds.iisc.ac.in.

https://doi.org/10.1145/3199523

1 INTRODUCTION

The growing popularity of technologies such as Internet of Things (IoT), mobile devices, smartphones, and social networks has led toward the emergence of "big data." Such applications produce not just gigabytes or terabytes of data, but soon petabytes of data that need to be actively processed. Such large volumes of data gathered from billions of connected people and devices around the world is causing unprecedented challenges in terms of how data can be stored, retrieved, and managed; how data security, integrity, availability, and sharing can be ensured; how massive datasets can be mined; and how they can benefit from new computing paradigms such as cloud computing for data analysis (Pettey 2011; Dias de Assuncao et al. 2015).

According to the National Research Council of the US National Academies (Committee on the Analysis on Massive Data 2013), graph processing is among the seven major computational methods of huge data analysis. Graph computations are used in business analytics, social network analytics, image processing, hardware design, and deep learning to an increasing extent. Wide-spread techniques for processing large graphs had, until recently, been limited to shared memory (Hong et al. 2012; Bader and Madduri 2008) and high-performance computing systems (he Graph 500 List 2010; Karypis and Kumar 1995; Harshvardhan et al. 2013). Although distributed approaches have been proposed for processing big graphs since 2001 (Huberman 2001), graph processing systems for commodity clusters and clouds have become particularly popular after Google introduced its Pregel (Malewicz et al. 2010) vertex-centric graph processing system in 2010. Since then, several distributed graph processing frameworks with diverse programming models and features have been proposed to facilitate operations on large graphs. Each of these frameworks has specific characteristics with its own strengths and weaknesses.

The aim of this article is to provide a taxonomy of scalable graph processing systems and frameworks. It identifies strengths and weaknesses in the field and proposes future directions. First, it proposes a comprehensive taxonomy of programming abstractions and runtime features offered by graph processing systems, and maps the existing systems to this taxonomy. Second, it utilizes a top-down approach for investigating graph processing frameworks and their components along with examples to support them. Third, the article identifies gaps in existing systems that need further investigation, and discusses these open problems and future research directions in detail. In summary, this survey gives readers an overarching picture about what graph processing is, what improvements have been gained through recent frameworks, different programming and runtime techniques that have been used, and the applications that benefit from them. It emphasizes scalable graph processing platforms for shared-memory and distributed processing that fall within the ambit of big data processing platforms. It also contrasts them against graph frameworks for supercomputing systems, as evidenced through the Graph500 benchmark.[1] On the other hand, existing works such as McCune et al. (2015) and Doekemeijer and Varbanescu (2014) only focus on surveying and they have limited focus on key elements of graph processing.

The rest of the article is organized as follows: Section 2 includes a definition of graphs and graph processing systems, contrasts graph processing from other big data processing methods, outlines the lifecycle of a typical graph processing system, and gives examples of real graph-based applications and algorithms. Contemporary graph processing frameworks and architectures are explained in Section 3, along with distributed coordination and computational models. Section 4 categorizes existing frameworks based on partitioning, communication models, in-memory execution, fault tolerance, and scheduling. Graph databases are reviewed in Section 5. A taxonomy and discussion on challenges is also presented for each section. A gap analysis and open challenges,

[1]Graph 500 benchmark, http://www.graph500.org/.

Table 1. Graphlike Application and Environments

Application	Item (Vertices of the Graph)	Connection (Edges of the Graph)
Social network	Members	Friendships
Computer network	Computers	Network connectivity
Web content	Web pages	Hyperlinks
Transportation	Cities	Roads
Electrical circuit	Devices	Wires
Commerce	Customers, goods	Purchase transactions
Factory	Machines	Production lines
Supply chain	Providers	Distances
Telecommunication	Mobile Phones	Phone calls

with a perspective on future directions are discussed in Sections 6 and 7, respectively. Finally, we conclude the article in Section 8.

2 BACKGROUND

A Graph G = (V, E), consists of a set of vertices, V = {v1, v2, . . . , vn} and a set of edges, E = {e1, e2, . . . , em} that indicate pairwise relationships, E = V × V. If (vi, vj) ∈ E, then vi and vj are neighbors (Sedgewick and Wayne 2011). The edges may be directed or undirected. So, V and E are the two defining characteristics of a graph, which most graph processing frameworks implement. Frameworks typically support a single attribute value associated with the vertex and edge (e.g., label, weight). In addition, some of the platforms also support a set of named and/or typed attributes for their vertices and edges as part of their data model.

Pairwise relationships between entities play an important role in various types of computational applications. These relationships that are implied by different connections (edges) between items (vertices) give rise to domain questions to draw value from the data, such as: Is it possible to identify transitive relationships between items by following the connections? How many items are connected to a typical item? What is the shortest distance between these items? Which groups of items are similar to each other? How important is an item relative to others? Various graphlike applications and environments are mentioned in Table 1 (Sedgewick and Wayne 2011). As can be seen in Table 1, many applications process data that naturally fits into a graph data model. Several of these applications from social networks, eCommerce, and telecom domains handle large graph datasets, which need to be processed and mined to draw disparate business intelligence, ranging from the interests of people about products for targeted advertising to tracing call logs for cyber-security. Processing large graphs poses some intrinsic challenges due to the nature of graphs themselves. These characteristics make graph processing ill-suited to existing data-processing approaches, and usually inhibit efficient parallelism (Pellegrini 2011). According to Lumsdaine et al. (2007), their properties are noted below:

(1) *Data-driven computations*: Graph computations are usually entirely data-driven. Graphs are made up of sets of vertices and edges that dictate the computations performed by every graph algorithm.
(2) *Irregular problems*: Graph problems are highly irregular due to the non-uniform edge degree distribution and topological asymmetry rather than being uniformly predictable problems, which can be optimally partitioned for concurrent computation.

(3) *Poor locality*: The inherent irregular characteristics of graphs leads to poor locality during computation, which is in conflict with locality-based optimizations supported by many existing processors, making it difficult to achieve high performance for graph algorithms.
(4) *High data access to computation ratio*: A large portion of graph processing is usually dedicated to data access in graph algorithms. Therefore, waiting for memory or disk fetches is the most time-consuming phase relative to the actual computation on a vertex of edge itself.

To streamline the processing of big data, MapReduce, a distributed programming framework for processing large datasets with parallel algorithms, was introduced by Google in 2004 (Dean and Ghemawat 2004). MapReduce has two significant advantages: (1) The programmer has a simple and familiar interface using Map and Reduce functions, inspired by functional programming concepts (Hudak 1989) and (2) the application is automatically parallelized when defined using Map and Reduce methods, without the programmer needing to know how data will be distributed, grouped, and replicated, and how the tasks are scheduled.

Although MapReduce addresses many deficiencies in traditional parallel and distributed computing approaches, it has several limitations that make it less efficient for processing large graphs (Cohen 2009; Afrati et al. 2012; Grabowski et al. 2013): (1) MapReduce is limited to a two-phased computational model that is not naturally suited for graph algorithms that run over many iterations. (2) In common MapReduce implementations, the input graph and its state are not retained in main memory across even in these two phases, let alone across iterations, and consequently requires repetitive disk I/O. (3) MapReduce's tuple-based approach that is unaware of the linked nature of graph datasets is poorly suited to design many graph applications. (4) Graph operations using MapReduce have poor I/O efficiency—because of frequent checkpoints on completed tasks and data replication—which is a bottleneck for many graph algorithms (Lee et al. 2011).

Apache Hadoop (Apache Software Foundation 2011) is a popular open-source implementation of the MapReduce programming model. Besides flexible batch-processing applications that can be built using Hadoop, it is also the basis for NoSQL querying platforms such as Pig (Apache Software Foundation 2008) and Hive (Apache Software Foundation Contributors 2011) to work with large datasets. In addition, various high-level languages such as SCOPE (Zhou et al. 2012), Sphere (Gu et al. 2010) and Swazal (Pike et al. 2005) are available for MapReduce-like systems. Platforms like Apache Spark (Matei Zaharia 2012) have extended the programming model of MapReduce, and offer incremental batch and in-memory computation with better performance. Further, Hadoop's distributed file system storage mechanism (HDFS) as well as a Map-only model of Hadoop is used as the storage and distributed scheduling mechanism in many graph processing frameworks we discuss.

While a number of systems such as PEGASUS (Kang et al. 2009) have brought innovative approaches for processing and mining peta-scale graphs, those systems are based on the MapReduce model and suffer from the above limitations. As a consequence, iterative graph processing systems started to emerge in 2010 with Google's Pregel (Malewicz et al. 2010), a graph processing framework that uses Valiant's Bulk Synchronous Parallel (BSP) processing model (Valiant 1990) for its computation. Pregel was the first system that promoted a "think like a vertex" notion for processing large graphs, similar to MapReduce that operates on <key,value> pairs to process large data volumes. These and other contemporary graph processing systems are discussed further in this survey.

2.1 Overall Scheme of Graph Processing

In general, a typical graph processing systems execute a graph algorithm over a graph dataset across different logical phases, as shown in Figure 1.

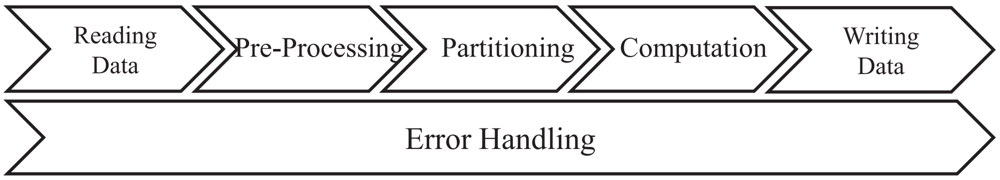


Fig. 1. Graph processing phases.

(1) *Read/write input/output datasets*: The first step is reading the graph data from a source dataset, which can be either on disk or in memory. In the last phase, the processed data should be written back again, either to disk or memory. Graph processing systems typically do not have a custom persistence layer optimized for reading and writing graph datasets, and tend to use the standard file system such as HDFS. Hence, they can present a bottleneck when reading and writing large graph datasets. Many studies even ignore read/write time when they measure the execution time for evaluation. Instead, they try to improve other aspects of the systems like efficient in-memory data structures for computation.
(2) *Pre-processing*: In some approaches, the graph data will be partitioned before being passed to the graph processing system to decrease the overall burden and runtime of the system. The main advantage of this approach is that the programmer does not need to worry about the complexity of the partitioning and it is a one-time cost that is paid upfront. Also, partitioning mechanism and computation mechanism can be two different modules that work independently and thus can be designed and implemented separately. The main drawback for this approach is that it works well only for static partitioning strategies, not dynamic partitioning or repartitioning.
(3) *Partitioning*: In this phase, partitioning will be done dynamically within the graph processing system and not as a separate module. Both partitioning and computation phases can collaborate to choose the best partitioning method at each step, so, dynamic partitioning and repartitioning can be implemented in the processing system. Although programming such a system is more complicated than implementing two independent modules, it provides more runtime flexibility and can be well suited to support diverse graph algorithms.
(4) *Computation*: Different graph processing systems have different computation approaches. This programming model and runtime is at the heart of the whole framework and there have been many proposals for efficient computation methods to decrease the graph application's runtime. More details about this phase, with a taxonomy on various computational models, are presented in Section 3.4.
(5) *Error handling*: This fault-tolerant and failure recovery phase will be applied to the system either during the computation phase or after the computation phase is completed. There are various techniques that can be used here, such as check-pointing or restarting applications. Typically, the time taken for error handling is not considered in experimental results due to the overheads it causes and some frameworks even avoid considering this capability. However, given the use of large commodity clusters that are prone to failures and long-running big data applications, fault tolerance is essential for graph processing frameworks used in an operational setting.

2.2 Large Graph-Oriented Applications

As noted in Table 1, there are many fields and applications that generate and provide big data in the form of graphs. With improvements in computer hardware and processing models such as cloud computing and emerging concepts like IoT, an even greater growth in datasets is expected. Here, we present some typical applications and environments where large graph data are generated and used.

Social networks. Social networks and applications have grown exceedingly popular during the past decade and are constantly adding features to make effective use of the data they collect and to grow their customer network. Social networks are an important source of big graph data, and even big data in general, with large amounts of data created every day (Commission 2010). People are sharing their personal activities with their friends and the whole world, talking about their beliefs, sharing photos and videos, and posting their interests and health information (Cha et al. 2007; Stelzner 2015). In the year 2014, in each minute, 2,460,000 content posts were shared on Facebook, 3,472 photos were pinned on Pinterest, 72 hours of new videos were uploaded to YouTube, 2,78,000 tweets were shared on Twitter, and 20 million photos were viewed on Flicker. These rates continue to grow, and form just a part of the whole big data social network landscape (Gunelius 2014).

Social networks are native generators and consumers of graph datasets, with an additional temporal dimension added to them. “Users” form the vertices of a huge social graph while “friendship” connections between them form the edges of the graph. Connections can be probabilistic and node’s states change over time. Each node or edge can contain different values and information about a member’s personal details, his/her interests, friends, groups and people, his/her followers, the pages that are visited, locations, business information, and so on with many other meta-data about his/her history of activities. All these form a digital trail for every user that needs to be processed and analyzed by social network providers.

From the user’s point of view, the network provider needs to suggest relevant pages or communities for them to follow based on their interests and offer meaningful service offerings (Akbari et al. 2013; Bagci and Karagoz 2015). The providers themselves benefit from leading users through targeted advertising to paid services. Although popular, social network sites are still in their infancy as they figure out how to monetize this massive dataset they have access to and make their business model sustainable. New methods and mechanisms are emerging in the area of analyzing social network data on distributed systems, clusters and clouds (Leimbach et al. 2014).

Computer networks and the Internet. Every machine in a computer network, including clients, servers, routers and switches, is a node of a network graph and physical or network connections between these machines form the edges of the network graph. When various networks from all over the world are connected together to provide different services, it forms the internet, which is an extremely large graph (Chen et al. 2005). Computer networks need to be analyzed to discover whether there might be intruders, resource wasters, low efficiency, dead paths, and also to gain statistical reports about the states of the network (Ammann et al. 2002). This is particularly the case as a bulk of the network traffic moves toward rich content such as streaming video and multiplayer gaming. These types of graphs should be processed in real-time as their state changes, and need a fast response, say to configure switches to allocate bandwidth to traffic, or detect malware and denial of service attacks. Network delays lead to customer dissatisfaction or worse, outages can cripple the functioning of modern society.

Smart utilities. Many large graph datasets are owned by public utility and service providers such as city and rail roads, and power and water grids. Take city road datasets as an example. Logistics companies need to find the shortest path between cities and streets to decrease their fuel consumption and ensure timely delivery of their goods, governments need to plan maintenance and provision emergency services in case of power disruptions or natural disasters, and people need to find the most convenient means for travel between different locations (Lochert et al. 2005).

Table 2. Graph Algorithms Categorization

Traversal	Breadth first search (BFS) Single source shortest path (SSSP)
Graph Analysis	Diameter Density Degree distribution
Components	Connected components Bridges Triangle counting
Communities	Max-flow min-cut K-means, semi clustering
Centrality Measures	PageRank Degree centrality Betweenness centrality
Pattern Matching	Path/subgraph matching
Graph Anonymization	K-degree anonym K-neighborhood anonym
Other Operations	Structural equivalence, similarity, ranking, and so on

Further, with a wider deployment of IoT and city services getting smarter (Paul 2013), the ability to monitor and collect real-time information about these physical infrastructure networks will grow, and graph analytics will be essential for ensuring the smartness of these utilities. In fact, IoT will be a natural extension and an exponential expansion of the internet. Graph applications can be used to drive real-time management of power grid operations with back-to-grid intermittent renewables like solar and wind, pumping operations for water networks, signaling of traffic lights based on current flow patterns, and even scheduling of public transit on-demand.

There are many other examples such as telecommunication (Marburger and Westfechtel 2010), web search engines (Page et al. 1998), environmental analysis (Committee on the Analysis on Massive Data 2013), astronomy (Szalay 2011), mobile computing (Tian et al. 2002), machine learning (Zha et al. 2009), and so on where large graph data is required to be processed and, as we mentioned before, traditional approaches are not suitable.

2.3 Algorithms in Graph Processing Studies and Experiments

We discuss algorithms that are commonly used in most large graph processing studies and experiments. These algorithms are not essentially graph-designed algorithms in terms of the level of parallelism but they have been used for experiments in papers. So, the categorization presented in the table below provides a hint for researchers. However, several works have been done on designing parallel versions of these algorithms to fit them into the graph processing domain (Leiserson and Schardl 2010; Maleki et al. 2016). Table 2 shows the taxonomy of algorithms according to Dominguez-Sal et al. (2010), which is used in a number of works:

(1) *Graph traversal algorithms*: These algorithms travel through all the vertices in a graph according to a specific procedure to check or update the vertices' values (Sedgewick and Wayne 2011). Among the most common algorithms in this type are breadth-first-search (BFS) and depth-first-search (Kozen 1992). Both of these algorithms traverse the graph tree to find a particular node, or visit every node in a specific order. Single-source shortest path (SSSP) is used to find the shortest path between a particular node and any arbitrary

node of the graph that might be based on the minimum cost or weight (Roy 2014). Dijkstra's algorithm and the Bellman-Ford algorithm are popular algorithms in this category (Bannister and Eppstein 2012; Dijkstra 1959).

(2) *Graph analysis algorithms*: These algorithms peruse the topology of the graph to specify graph objects and analyze its complexity. These graph statistics and topological measures are extensively used in protein interplay analysis and social network analysis (Britton et al. 2006; Lau 2012).
(3) *Components*: Connected component algorithms find subgraphs in which a path exists between any two nodes in the subgraph and none are connected to nodes in other subgraphs (Hirschberg et al. 1979). So, each vertex only belongs to one connected component of the graph. Weakly connected components (WCCs) work on undirected graphs, while strongly connected components are relevant to directed graphs. Another component identification problem is counting triangles.
(4) *Communities*: A community is a set of vertices in which each vertex in the community is closer to other vertices of the same community than any other vertices of the graph. Various topological and attribute measures can be used to define the closeness and quality of communities, and k-means clustering and semi-clustering are popular algorithms in this category (Jain 2008; Boykov and Kolmogorov 2004).
(5) *Centrality measures*: The aim of these algorithms is to give an approximate indication of the importance of a vertex in its community according to how well it is connected to the network. The most used algorithm of this type is PageRank (Page et al. 1998), an algorithm that is used by Google search engine to rank websites. Betweenness centrality is another common metric (Madduri et al. 2009).
(6) *Pattern matching*: These algorithms are used to recognize the presence of input patterns in the graph, which can be an exact or approximate recognition (Sun et al. 2012).
(7) *Graph anonymization*: These algorithms are used to create a new graph based on an original graph where the latter emulates specific topological or attribute properties of the original one. This prevents any possible intruders to re-identify the network (Wu et al. 2010).
(8) *Other operations*: There are also other algorithms, such as random walk algorithms, where we choose a vertex randomly from neighbors of a vertex to start or continue the process from there and try to converge in a probabilistic point (Fouss et al. 2007).

In another categorization used by Han et al. (2013) and Kang et al. (2011), algorithms have been categorized based on the types of graph queries that result in two classes of algorithms (other types of query classification can be found in Sarwat et al. (2013) and Jamadagni and Simmhan (2016)):

(1) *Global queries*: These queries need to traverse the whole graph. So, algorithms such as diameter estimation, PageRank, connected components, random walk with restart (RWR), degree distribution, and the like are in this group.
(2) *Targeted queries*: These queries only need to access part of the graph, not all the graph. Kang et al. (2011) have formulated seven types of queries including neighborhood (1-step and n-step), induced subgraph, egonet (1-step and n-step), k-core, and cross-edges.

Although there are many algorithms that can be implemented on a graph processing system, there are some challenges that these algorithms face. First, according to Lumsdaine et al. (2007), many graph systems have limited memory that can be exclusively allocated to the processing algorithm, in addition to other processes and threads that simultaneously use and access the memory. Graph algorithms, in particular, those that operate in a shared-memory system, can exceed

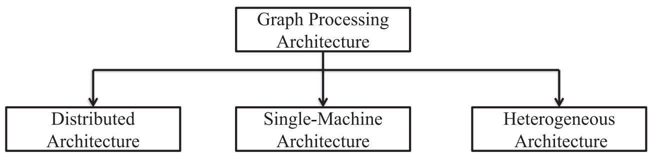

Graph Processing Architecture
Distributed Architecture
Single-Machine Architecture
Heterogeneous Architecture

Fig. 2. Graph processing architectures.

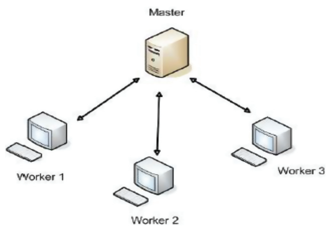


Fig. 3. Master-workers architecture.

available physical memory for large graphs processed on single machines (Murphy and Kogge 2007). Recent graph processing systems address this by using a distributed computing paradigm. In addition, utilizing external memory algorithms is another approach to reach out of memory (core) to have access to more space. Second, the level of granularity in an algorithm can influence the level of parallelism it can exploit, especially those with linear runtime. So, a more fine-grained level of parallelism results in better scaling of such algorithms (Hendrickson and Berry 2008). Third, algorithms should deal with diverse workloads and need to reassign tasks to processors when the visited nodes in a graph algorithm have spatial locality in the global memory. Finally, graph processing systems and algorithms should deal with the additional degree of parallelism exposed by submitting multiple concurrent queries when working on a large graph, or algorithms that operate over dynamic graphs. However, most of the systems that we review operate one graph algorithm or query over a single (large) graph.

3 GRAPH PROGRAMMING MODELS

We present different dimensions of graph programming and computation models, and classify and analyze prominent literature on graph frameworks based on these categories. A comprehensive list of graph processing systems based on this taxonomy is tabulated in Table 3.

3.1 Graph Processing System Architectures

Graph processing systems can be categorized into three types of architecture models as depicted in Figure 2.

3.1.1 Distributed Architecture. A distributed system includes several processing units (host) and each host has access to only its own private memory. Each partition of the graph is typically assigned to one host to be processed while the hosts interact with each other by explicit or implicit message passing (Strandmark and Kahl 2011). Such systems are meant to weakly scale by supporting larger graphs as more hosts are added to the system. From a cloud computing point of view, these map to an infrastructure as a service (Buyya et al. 2009) architecture, where the hosts are virtual machines (VMs). Distributed graph processing systems utilize master-slaves (workers) architecture, as shown in Figure 3, where there is one master that is responsible for managing the whole system, assigning partitions to workers, managing fault-tolerance, coordinating the operations of the workers, and so on; and there are multiple workers that are responsible for performing computation on the partitions.

Although the programmer has to adapt their algorithms and applications to suit the abstractions provided by the distributed graph processing systems, such systems ease the scaling of the applications on distributed environments, without the challenges of races and deadlocks that are associated with distributed computing (Coulouris et al. 2012). In contrast, shared-memory frameworks that have been developed for single machines are easier to program but are limited by their ability to hold only parts of large graphs in memory (Shun and Blelloch 2013).

Google Pregel is a distributed vertex-centric framework that uses a master-worker architecture on multiple hosts of a cluster. GraphLab, developed in Carnegie Mellon University and later supported by GraphLab Inc., was developed for single machine processing (Low et al. 2010), but evolved into a distributed one (Low et al. 2012). There are other Pregel-like systems such as GPS (Salihoglu and Widom 2013), Mizan (Khayyat et al. 2013), and GoFFish (Simmhan et al. 2014), and non-Pregel-like systems such as Presto (Venkataraman et al. 2013), Trinity (Shao et al. 2013), and Surfer (Chen et al. 2010), which have been developed as distributed graph processing systems. Even frameworks such as GraphX (Xin et al. 2013) are built on top of a Spark distributed dataflow system. All of these systems use multi-node clusters or cloud VMs for their execution environment. However, as of yet, none of these exploit the elasticity property of clouds, and rather treat captive VMs as a commodity cluster.

Besides the aforementioned graph frameworks, there are several graph processing libraries developed for high performance computing clusters. Boost graph library (BGL) (Siek et al. 2002) is a generic graph processing library that provides generic interfaces to the graph's structure and common operations, but hides the details of its implementation. This allows graph algorithms using BGL to have interoperable implementations on shared-memory and parallel computing platforms. Graph500 (he Graph 500 List 2010) is a graph processing benchmark by which various metrics of supercomputers such as communication performance, memory size for graph storage, and the performance of random access to memory are measured. It contrasts with Top500[2] which is designed for compute-intensive applications. Although there have been many other attempts for providing parallel graph frameworks for high performance computing including several libraries such as MPI (El-Rewini and Abd-El-Barr 2005), PVM (Geist et al. 1994), BLAS (Lawson et al. 1979), JUNG (O'Madadhain et al. 2003), and LEAD (Mehlhorn and Näher 1995), none of them provide the required flexibility for a general-purpose graph processing platform (Gregor and Lumsdaine 2005).

3.1.2 *Shared-Memory Architecture.*

Prior to the recent growth in distributed graph processing systems, there have been several works on processing large scale graphs on a single machine. A single machine consists of one processing unit (host), which can have one or more CPU cores, and physical memory that ranges from a few to hundreds of gigabytes that is shared across all the cores.

In 2012, Microsoft researchers conducted a study (Rowstron et al. 2012) on whether using Hadoop on a cluster for analyzing big data is the right approach for data analytics. They concluded that for many data processing tasks, a single machine with large memory is more efficient than using clusters. They also investigated the cost aspect of using a single machine in big data processing and mentioned that "… for workers that are processing multi gigabytes rather than terabytes + scale, a big memory server may well provide better performance per dollar than a cluster" (Lorica 2013).

Shared memory frameworks are inherently limited in the amount of memory and CPU cores present in that single machine (Doekemeijer and Varbanescu 2014). The main challenge is that single hosts often have limited physical memory whereas processing large, real-world graphs can require a significant amount of memory to retain them fully in memory for many graph applications, or keeping and managing a subset of the graph out-of-memory. Novel techniques to address this limitation have been proposed.

In Strata Startup Showcase 2013, SiSense, which is a business intelligence solutions provider company, won the audience award with a software system called Prism that can exploit a terabyte

[2]Top500. n.d. Home Page. Retrieved June 25, 2016, from https://www.top500.org.

of data on a single machine with only 8GB of RAM (Lorica 2014). It relies on disk for storage, transfers data to memory when needed, and benefits from L1/L2/L3 caches of the CPU. It utilizes a column store and an interface that allows scalability to a hundred terabytes. Among major IT companies, for instance, Twitter uses Cassovary[3], an open-source graph processing system that has been developed to handle graphs that fit in the memory of a single machine. It has been claimed that Cassovary is a viable system for "most practical graphs" because of using a space efficient data structure. WTF (who to follow) (Gupta et al. 2013) is a recommendation algorithm that is used by Twitter to suggest users with common interests and connections that is implemented on Cassovary.

GraphChi (Kyrola et al. 2012) is a vertex-centric graph processing framework that proposes a parallel sliding window (PSW) method for leveraging external memory (disk) and is suited for sparse graphs. PSW needs a small number of sequential disk-block transmissions, letting it perform well on both SSD (solid state drive) and HDD (hard disk drive). Besides, GraphChi can process an ongoing in-flow of graph updates while performing advanced graph mining algorithms simultaneously, like Kineograph (Cheng et al. 2012). GraphChi uses space-efficient data structures such as a degree file that is created at the end of processing to save in-degree or out-degree for each vertex as a flat array. It also uses dynamic selective scheduling that lets *update function* and *graph amendments* to enlist vertices to be updated. It was extended later as a graph management system called GraphChi-DB (Kyrola and Guestrin 2014) and tried to address some of these challenges.

Many other graph processing systems have been developed based on single machines. Signal/Collect (Stutz et al. 2010), for example, is a vertex-centric framework made to improve the semantic web computational performance. In this model, signals will be sent along edges where they will be collected in vertices. The advantage of this model is that it provides flexibility for synchronous, asynchronous, and prioritized execution. Other systems such as RASP (Yoneki et al. 2014) and X-stream (Roy et al. 2013), which provide an edge-centric framework, FlashGraph (Zheng et al. 2015), Galois (Nguyen et al. 2013), TOTEM (Gharaibeh et al. 2013), BPP (Najeebullah et al. 2014b), and others also make processing graphs possible on single machines using various computational models and processing systems.

Graph processing on a single machine would be easier to program and execute than on distributed systems if the entire graph fits within the local resources on that machine. This is because of efficient communication, simpler debugging, and easier execution management on a single machine. But this limits their scalability beyond a certain graph size. New approaches for processing graphs on single machines are targeting flash and SSDs (Yamato 2015; Koo et al. 2015) whose speeds are matching main memory and offer advantages for graph processing (Zheng et al. 2015; Nilakant et al. 2014; Yoneki et al. 2014).

3.1.3 Heterogeneous Architecture. In a heterogeneous environment, not every processing unit is equally powerful (Guo et al. 2015). This may be a single machine and additional on-board accelerators and specialized devices, or it can also consist of distributed, non-homogeneous systems. Because of this, we considered them as a separate group in this taxonomy. For example, processing systems such as RASP (Yoneki et al. 2014) and FlashGraph (Zheng et al. 2015) have tried to optimize the storage part of the system by using SSD, which is much faster and more reliable than traditional hard drives (Geier 2015). Many graph processing systems have proposed utilizing graphic processing units (GPU) alongside CPU for computation (Zhang et al. 2015). Medusa (Zhong and He 2013a), for instance, was developed to make processing graphs using GPUs easier. Medusa is a programming framework that enables users to write C/C++ APIs to promote the capabilities

[3]Twitter. 2012. Cassovary: A Big Graph-Processing Library. Retrieved July 24, 2015, from https://blog.twitter.com/2012/cassovary-big-graph-processing-library.

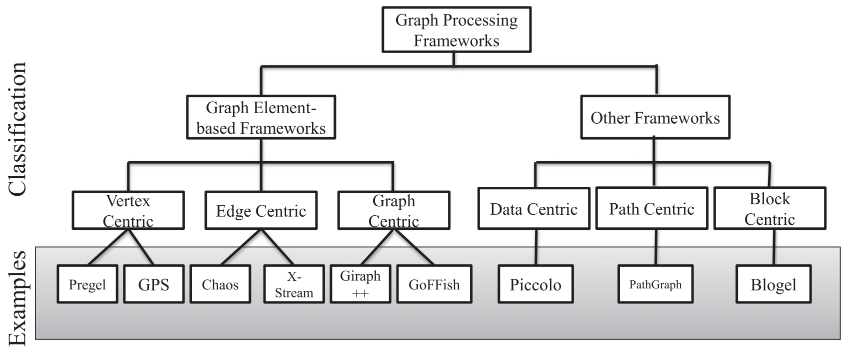


Fig. 4. Taxonomy of programming models used by graph processing frameworks.

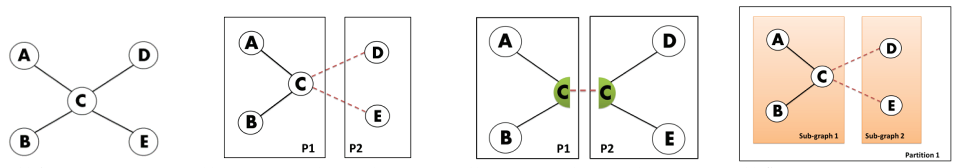
of GPUs to execute the APIs in parallel. Its extended version also can be run on multiple GPUs within a single machine. Gharaibeh et al. (2013) developed a system called TOTEM that assigns the low-degree vertices to the GPU and operates high-degree vertices processing on the CPU. On the other hand, systems like CuSha (Khorasani et al. 2014) compute the entire graph on GPU. Another possibility is to exploit non-uniform VM sizes on Clouds for a distributed, heterogeneous architecture, which has been less explored.

Recently some research works have started exploring the use of field-programmable gate arrays (FPGAs). FPGA is an integrated circuit made of matrix of configurable logic blocks and their programmable connections that can be configured by the user after being manufactured. GraphGen (Nurvitadhi et al. 2014) is a generic vertex-centric FPGA-based graph processing framework. It has been designed to get vertex-centric specifications and create FPGA implementations for targeted platforms. The problem with GraphGen is that it keeps the entire graph inside the on-board DRAM that limits the scalability of the system remarkably. FPGP (Dai et al. 2016) is another framework that enables interval-shared vertex-centric processing on FPGA. FPGP has also been used to analyze the performance bottleneck of other processing frameworks on FPGA. Although it has been shown that FPGP does not perform as good as CPU-based single server frameworks, it shows the mechanism of FPGA-based generic graph processing systems well. Overall, FPGA is still a new research area in graph processing context compared to CPU and GPU based systems, but it is getting more attention (Engelhardt and So 2016; Ma et al. 2017).

3.2 Graph Processing Frameworks

Graph processing frameworks enable graphs to be processed on different infrastructures such as clusters and clouds. Here, we restrict ourselves to distributed memory systems that are designed for commodity, rather than high-performance computing or supercomputing clusters. The programming abstraction for each framework is designed either based on a graph topology element, such as vertices and edges, or other alternative approaches. Figure 4 depicts the taxonomy of graph processing frameworks according to main characteristics of the graph and other alternatives. We discuss these further below.

3.2.1 Vertex-Centric (Edge-Cut) Frameworks. Vertex-centric programming is the most mature distributed graph processing abstraction and several frameworks have been implemented using this concept (Doekemeijer and Varbanescu 2014). A vertex-centric system partitions the graph based on its vertices, and distributes the vertices across different partitions, either by hashing them without regard to their connectivity (Malewicz et al. 2010; Apache Software Foundation 2012) or by trying to reduce the edge cuts across partitions (Salihoglu and Widom 2013). Edges



(a) A sample graph (b) Vertex-centric (edge-cut) approach (c) Edge-centric (vertex-cut) approach (d) Component-centric approach

Fig. 5. Graph element-based approaches for graph processing frameworks.

that connect vertices lying in two different partitions either form remote edges that are shared by both partitions or owned by the partition with the source vertex.

In the vertex-centric programming abstraction introduced by Google's Pregel (Malewicz et al. 2010), computation centers around a single vertex—its state and its outgoing edges—and interactions between vertices are through explicit messages passing between them. This gives a fine-grained degree of vertex-level data parallelism that can be exploited for concurrent execution. Pregel's execution follows a BSP model, where vertex computation and inter-vertex messaging are interleaved, and the application iteratively progresses along barrier-synchronized supersteps. The Pregel API allows developers to focus on the vertex-centric graph algorithms while abstracting away communication and coordination details to the runtime. In Pregel, the domain of a vertex's user-defined *compute* function is restricted to the vertex and its outgoing edges, while LFGraph (Hoque and Gupta 2013) considers incoming edges to be restricted. Figure 5(b) shows vertex-centric processing approach for a sample graph shown in Figure 5(a).

A vertex-centric model makes programming of graph processing *intuitive and easy*, similar to the advantages of Map-Reduce for tuple-centric programming. Parallelization is done *automatically*, and *race conditions* on distributed execution are avoided. Primitives like *combiners* and *aggregators* are available for application-level message optimizations and global state exchange. The model also allows for *graph mutations*, where the structure of the graph can be changed as part of the execution (useful, e.g., when iteratively coarsening the graph for partitioning, clustering, or coloring).

However, Pregel has several shortcomings: (1) While the vertex-centric model exposes parallelism at the level of individual vertices, which can be computed in negligible time, massive graphs can impose coordination overheads on this degree of parallelization that may outweigh the benefits (Tian et al. 2013); (2) the number of barrier-synchronized supersteps taken for traversal algorithm can be proportional to the diameter of the graph with the number of message exchanges required between partitions also being high, proportional to the number of edges (Simmhan et al. 2014); (3) mapping shared memory graph algorithms to this model is not trivial and requires new vertex-centric algorithms to be developed (Simmhan et al. 2014); and (4) using a vertex-centric programming model without regard to the graph partitioning and data layout on disk can lead to punitive I/O initialization and runtime performance (Simmhan et al. 2014). These shortcomings have been addressed in some other vertex-centric frameworks such as GoFFish (Simmhan et al. 2014) and GPS (Salihoglu and Widom 2013).

Apache Giraph (Apache Software Foundation 2012), is a popular open-source implementation of Pregel. Giraph uses Map-only Hadoop jobs to schedule and coordinate the vertex-centric workers and uses HDFS for storing and accessing graph datasets. It is developed in Java and has a large community of developers and users such as Facebook (Jackson 2013; Salihoglu et al. 2015). Giraph has a faster input loading time compared to Pregel because of using byte array for graph storage. On the other hand, this method is not efficient for graph mutations, which lead to decentralized edges when removing an edge. Giraph inherits the benefits and deficiencies of the Pregel vertex-centric programming model. Its performance and scalability is algorithm and graph dependent, and works very fast, e.g., on stationary algorithms like PageRank but not as fast on traversal algorithms like single source shortest path (SSSP) (Roy 2014) and WCCs (Salihoglu and Widom 2014), particularly for graphs with a large diameter. However, the ease of use of this framework and the community support has made it a popular platform over which to develop other Pregel-like systems with feature enhancements to the vertex-centric concept.

Other distributed platforms like Apache Hama and GraphX also offer a vertex-centric programming model, with features comparable to Giraph. GraphX, developed on top of Apache Spark, determines *transformation on graphs* where every operating action produces a new graph. This framework uses a programming abstraction called Resilient Distributed Graph (RDG) interface, which builds upon Spark's in-memory storage abstraction—Resilient Distributed Datasets (RDD). The graph in GraphX includes the *directed adjacency structure* along with *user defined attributes* connected to each node and edge, and both are encoded as RDGs. Using RDG, the implementation of frameworks such as Pregel and PowerGraph on Spark needs less efforts.

Pregelix (Bu et al. 2014) is a vertex-centric framework that tries to model Pregel as an iterative dataflow on top of the Hyracks (Borkar et al. 2011) parallel dataflow engine. Pregelix has been developed to address three main challenges in distributed Pregel-like systems: (1) Many Pregel-like systems have limitations to support out-of-core vertex-storage; (2) existing Pregel-like systems have specific strategies and implementations for communication, node storage, message delivery, and so on. Therefore, a user cannot choose between different implementation strategies based on what is better for a particular algorithm, dataset or cluster. Pregelix improves physical flexibility and scalability of the processing system to address this challenge; and, finally, (3) Pregelix tries to leverage current data-parallel platforms to streamline the implementation of Pregel-like systems.

3.2.2 Edge-Centric (Vertex-Cut) Frameworks. In edge-centric frameworks, edges are the primary unit of computation and partitioning, and vertices that are attached to edges lying in different partitions are replicated and shared between those partitions. It means that each edge of the graph will be assigned to one partition, but each vertex might exist in more than one partition. Figure 5(c) depicts this approach. While edge-based partitioning is more costly, this model shows better graph processing performance compared to vertex centric approaches (Rahimian et al. 2014). However, programming an edge-centric system is more difficult than vertex-centric systems (Yuan et al. 2014). It is also important to create edge-balanced partitions in this method to load balance the computation across workers, just as vertex balancing is important for vertex-centric frameworks. Decreasing the vertex cuts has been investigated in some research (Feige et al. 2005; Beseri Sevim et al. 2012; Liu et al. 2006).

Catalyurek and Aykanat (1996) and Devine et al. (2006) have suggested a vertex-cut method for distributed graph placement in hyper graph partitioning, where the edge-centric problem can be solved by converting each edge into a vertex and vice versa. The motivation for developing vertex-cut frameworks is that systems such as Pregel and GraphLab (Low et al. 2010) are effective for flat graphs but have shortcomings with graphs that follow a *power law edge degree distribution* due to low quality partitioning and vertices with high edge degrees. Real-world graphs such as social networks are such power-law graphs where a small set of vertices have high edge degrees that connect to a large part of the graph, e.g., *celebrities* in social networks. Partitioning and representing power-law graphs in a distributed environment is also difficult (Leskovec et al. 2008; Abou-Rjeili and Karypis 2006).

X-Stream (Roy et al. 2013) is a well-known edge-centric system that processes out-of-core and in-memory graphs using a gather-scatter approach (Section 3.4). It bases this approach on the intuition that storage media such as SSDs, main memory, and magnetic disk perform significantly better with a sequential access to data than random access. The authors have implemented different algorithms on their system and observe that many of them can work on edge-centric mode. It can even return results from unsorted edge lists. However, it causes overheads when new edges are added to the graph. X-Stream is not suitable for very large graphs that do not fit onto the SSD, it wastes a remarkable amount of bandwidth for certain algorithms, and, finally, X-Stream is not suitable for graphs and algorithms that require many iterations (Yuan et al. 2014).

Chaos (Roy et al. 2015) is another edge-centric framework that is created based on X-Stream's streaming model. It introduces a scalable distributed framework that can be scaled from secondary storage to several hosts on a cluster. Unlike some other graph processing systems, Chaos does not strive to attain locality and load balance and claims that network bandwidth in a small cluster surpasses storage bandwidth. Instead, it is designed to partition the graph for sequential access on storage. So, it spreads data uniformly at random on the cluster's machines, which are not necessarily sorted edge-lists. Chaos also utilizes a gather-apply-scatter (GAS) computation model (Section 3.4) by which the edge-centric characteristic of its model is proven by iterating over edges and getting updated in the gather and scatter stages.

3.2.3 Component-Centric Frameworks. Component-centric approaches have been recently introduced, where components are collections of vertices and or edges that are coarser than a single vertex or edge. Tian et al. (2013) from IBM introduced "think like a graph" instead of "think like a vertex" (Tian et al. 2013) abstraction after observing shortcomings in vertex-centric and edge-centric methods of graph processing. In their partition-centric view, they divide the whole graph into partitions and assign those partitions to machines for being processed. A *partition*, which is a collection of vertices and edges in the graph, forms the unit of computing. Figure 5(d) shows this approach. Giraph++, based on Apache Giraph (Apache Software Foundation 2012), implements this model and uses this coarse-grained parallelism. In contrast to a vertex-centric model that hides partitioning and component connectivity details from users, Giraph++ exposes the partition's structure to the users to allow optimizations. So, the performance of the system depends on the partitioning strategy that is used and how effectively users exploit the access to the coarse components in their execution. On the other hand, communication within a partition is by direct memory access, which is faster than passing messages between each single vertex in a vertex-centric model. This results in fewer network messages passing and lower time of execution per iteration (superstep), with a reduction in the number of iterations needed for convergence. It also benefits from local asynchrony in the computation, which means that vertices in the same partition can exchange their state and perform consequent computations to the extent possible in the same iteration.

Simmhan et al. developed GoFFish (Simmhan et al. 2014), which has a subgraph-centric computation model to merge both the scalability and flexibility of the vertex-centric programming approach with the extensibility of shared-memory subgraph computation. A *subgraph* (WCC) is the unit of computation. A partition may contain one or more subgraphs, whereas each subgraph only belongs to one partition of the graph. Vertices in the same subgraph have a local path between each other, so existent shared memory graph algorithms can directly be exerted to each subgraph. This gives a programming and algorithm design advantage over partition-centric frameworks like Giraph++ that offer no guarantee on connectivity between vertices in a partition, while retaining the advantages of fewer iterations and shared-memory access of those frameworks. Subgraphs, or vertices that span subgraphs, communicate by passing messages, similar to a vertex-centric model.

GoFFish consists of two major components: (1) a distributed graph oriented file system, *GoFS*, which partitions, stores, and provides access to graph datasets in a cluster across hosts, and (2) a subgraph-centric programming framework, *Gopher*, which executes applications designed using the subgraph-centric abstraction using the Floe (Simmhan and Kumbhare 2013) dataflow engine on top of GoFS. However, subgraph-centric programming algorithms are vulnerable to imbalances in the number of subgraphs per iteration as well as non-uniformity in their sizes. The time complexity per iteration also can be larger since it often runs the single machine graph algorithm on each subgraph, even as it often takes much fewer iterations. The benefits are also more pronounced for graphs with large diameter, where algorithms tend to be several times faster than a vertex-centric equivalent, rather than small-diameter power-law graphs. GoFFish supports applications that operate on single property-graphs as well as on time-series graphs (Simmhan et al. 2014).

3.2.4 Other Graph Frameworks. In addition to the aforementioned programming abstractions, other alternatives have been developed as well. Several data-centric models offer a declarative dataflow interface to users to access and process data without needing to explicitly define communication mechanisms. For example, MapReduce provides a dataflow programming model that is popular for processing bulk on-disk data, but not for in-memory computations across multiple iterations, and applications do not have online access to the intermediate states. Piccolo (Power and Li 2010), was developed at New York University as a data-centric programming method for writing parallel in-memory applications in several machines. It uses a key-value interface with a user-defined accumulator function that automatically combines concurrent updates on the same key. Like many other dataflow models such as Pig, Hive, Dryad (Isard et al. 2007; Yu et al. 2008), Flume Java (Chambers et al. 2010), and Swazal, developers in Piccolo operate at a higher level of dataflow programming abstraction but need to know the framework and system behavior well to leverage its scalability for different applications. For example, the programmer has to *a priori* specify the number of partitions while creating a table. Further, these are tuple-oriented data flow models rather than graph specific ones.

Yuan et al. (2014) introduces PathGraph, which aims to leverage memory and disk locality on both out-of-core and in-memory graphs using a path-centric approach. Their path-centric abstraction utilizes a set of tree-based partitions to model the graph and benefits from a path-centric computation instead of a vertex or edge centric computation. It means that the graph will be partitioned into paths including two forward and reverse edge traversal trees for each partition. It applies iterative computation per traversal tree partition in parallel, and then merges partitions by examining border vertices. Two functions, *gather* and *scatter* (Section 3.4), are used to traverse each tree by a user-defined algorithm. In addition to the computation tier, PathGraph has a path-centric storage tier to better the local accessibility for the computation. The storage structure is based on a tree partition and uses vertex-based indexing for tree-based edge chunks. The system outperforms the vertex-centric GraphChi (Kyrola et al. 2012) and edge-centric X-Stream frameworks.

Frameworks such as Blogel (Yan et al. 2014) adopt blocks as units of computation. Blogel introduces the concept of "think like a block" rather than "think like a vertex" and argues that existing systems do not address three main characteristics of real-world graph including (1) skewed degree distribution, (2) large diameter, and (3) high density. Considering these characteristics, the basic idea in Blogel is to put a high degree vertex with all its neighbors in one block and assign the whole block to one host. It also uses three computing modes (B, V, VB-mode), depending on the algorithm, along with two different partitioners (graph Voronoi diagram (GVD) partitioner and 2D partitioner).

3.3 Distributed Coordination

Figure 6 shows various distributed coordination in existing graph processing systems.

3.3.1 Synchronous. When a graph algorithm executes synchronously, it means that concurrent workers process their share of the work iteratively, over a sequence of *globally coordinated* and well-defined iterations. Synchronization may be applied to vertex-centric, edge-centric, and

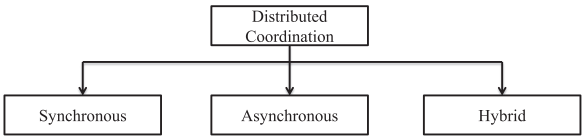


Fig. 6. Distributed coordination.

component-centric models, and both on distributed and single machine systems. For example, Pregel-like systems call their barrier synchronized iterations a superstep, and workers coordinate their computation and communication phases in each superstep—everyone completes a superstep before starting the next. Initially, the master assigns partitions to the workers in the first iteration; the workers update their set of vertices based on the assigned partitions and wait for a global barrier, which tells them all workers are ready with their partition (Malewicz et al. 2010). Subsequent supersteps indicate actual computation based on the application logic. Updated vertices in each partition send messages to (typically) vertices in neighboring partitions between iterations. Within an iteration, vertices can only access information about their local vertex's state and messages received from the previous iteration. Such a synchronized execution is possible even in a shared-memory system, across workers (threads, processes) on a single server.

These regular intervening periods make the system appropriate for algorithms where sizeable computation and communication can take place within each iteration since there is an overhead associated with the coordination. The bulk messaging at iteration boundaries can utilize the bandwidth efficiently if there is heavy communication between partitions (Ediger and Bader 2013; Xie et al. 2015). It is easy to program, debug, and deploy such systems, without concerns of distributed race conditions and deadlocks. Another advantage of synchronous processing is that the outcome of each superstep is known immediately and provides real-time response of incremental application progress and easier error recovery in case of superstep boundaries. Synchronous execution is also suitable for balanced workloads that are computed symmetrically, with all workers having adequate work, so that the overhead of the global barrier and idle time for faster workers waiting for slower workers to synchronize is reduced (Xie et al. 2015). These advantages make synchronous execution very popular such that several graph-processing systems like Pregel, GPS, Kineograph, Mizan, GasCL (Che 2014), and Medusa (Zhong and He 2013a) use this model. Some like GoFFish (Simmhan et al. 2015) have two levels of such synchronized supersteps, an outer loop over different graphs in the context of time-series graphs, and an inner loop as supersteps over a single graph from the outer loop.

A synchronous execution model has some disadvantages as well that should be considered while designing or choosing a processing system for graphs. First, this model is not suitable for unbalanced workloads in which computation converges asymmetrically (Suri and Vassilvitskii 2011). Likewise, if the distributed machines are not homogenous, the performance of the hardware may also cause some partitions to operate slowly. In such cases, it is possible to have stragglers when all partitions have been computed on workers except one slower worker which has not finished and hence delays all workers in the superstep. So, the runtime in this model is completely dependent on the slowest machine in each iteration (Salihoglu and Widom 2013). Some of these shortcomings have been identified and addressed through elastic load balancing of partitions across workers (Dindokar and Simmhan 2016). Another drawback is that the intermediate processing updates between supersteps, in the form of messages or state, has to be retained in memory and this causes additional memory pressure (Redekopp et al. 2013). A third disadvantage is that a synchronous execution model is ill-suited for applications and algorithms that need coordination between adjacent vertices (Doekemeijer and Varbanescu 2014). For example, in a graph coloring algorithm in which vertices try to choose a different color from their neighbors, two adjacent vertices might pick conflicting colors frequently (Gonzalez et al. 2012; Tasci and Demirbas 2013) and the algorithm will converge slowly. Lastly, based on the drawbacks mentioned above, the cost for the systems that use synchronous model of execution is higher because the throughput must always remain high and running time would be longer (Zhang et al. 2014).

3.3.2 Asynchronous. An asynchronous execution model does not have any global barrier and a subsequent phase of execution will be started on a worker immediately after its current iteration finishes its computation (Xie et al. 2015). Hence, some of the challenges of load balancing and long tail computation in the synchronous model are addressed by asynchronous computation, where workers do not have to wait for the slowest worker to start their subsequent iteration. This approach is useful when the workload is imbalanced and convergence can occur faster than synchronous approach. Therefore, we can say that an asynchronous model is the preferred model when computation across workers is heavily skewed and there is little communication that can benefit from bulk operations (Xie et al. 2015). In other words, this model is preferable for CPU-based algorithms while synchronous model would perform much better on I/O-bound algorithms. Another advantage for this model is that it can use dynamic scheduling to implement prioritized computation to execute more units of computation before others, to obtain better performance (Zhang et al. 2012). Normally, asynchronous execution provides more flexibility than synchronous execution by utilizing dynamic workloads, which makes it outperform synchronous methods in many cases; however, the exact comparison between these two models depends on various properties of the input graph, platforms that the system has been deployed on, execution stages, and applications (Xie et al. 2015). Finally, using asynchronous approach provides a non-blocking process because resources could be free and become available for the next iteration, whereas in a synchronous approach, they are blocked until the global barrier declares the end of superstep, which leads to a competition for resources at the beginning of next superstep.

As before, there are disadvantages to this model as well. The key disadvantage is that programming asynchronous processing systems is more difficult than synchronous systems. The programmer should deal with irregular communication intervals, unpredictable response time, complex error handling, and more complicated scheduling issues. For example, for error recovery in such a system, many factors have to be considered: which machine has faced a fault, in which iteration of a particular worker the error happened, which resources caused the errors, should new resources be allocated to the computation or it should only be rearranged, and so on. This also results in more complex debugging and deployment, and careful programming to avoid deadlocks. In case of a pull-based communication model (Section 4.2), which is usually implemented in an asynchronous manner, many redundant communications may happen because there are several intertwined reads and writes while adjacent vertices values do not change (Han et al. 2014; Zhang et al. 2012). On the other hand, regardless of these drawbacks, many single machine systems have preferred an asynchronous execution approach since the shared memory makes it easier to asynchronously use the latest data without waiting for a barrier.

3.3.3 Hybrid. There are many systems that use only synchronous execution mode; for example, Pregel, GoldenOrb (Cao 2011), GBASE (Kang et al. 2011), Chronos (Han et al. 2014), and GraphX (Xin et al. 2013), while many other systems utilize an asynchronous mode like GiraphX (Tasci and Demirbas 2013), GraphHP (Chen et al. 2014), Ligra (Shun and Blelloch 2013), RASP (Yoneki et al. 2014), and GraphChi. But recently a new approach called hybrid execution model has been implemented in a few systems that tries to take advantage of both asynchronous and synchronous approaches or incorporate them with new additional solutions. Such graph processing systems have been developed to improve system performance by overcoming the shortcomings of existing methods, and use both synchronous and asynchronous models of coordination to benefit from their relative strengths.

GRACE (Wang et al. 2013), for instance, is a single machine framework that combines synchronous programming with asynchronous execution features. It actually separates execution policies from application logic. In an asynchronous execution, a processing sequence of vertices can be intelligently ordered by dynamic scheduling to remarkably speed up the convergence of computation. GRACE uses the BSP computational model and message passing communication model as two primary paradigms of a synchronous model. It helps GRACE to improve its automatic scalability by applying prioritized execution of vertices and receiving messages selectively outside of the last iteration. Various workloads like topic-sensitive PageRank, social community detection, and image restoration have been used in GRACE and it shows comparable running time to other asynchronous systems such as GraphLab with even better scalability.

Another hybrid approach distinguishes between local vertices that are within a partition and remote vertices connecting across partitions. These types of systems exploit both local asynchronous computation when they still need global barrier for synchronization of remote vertices values (Chen et al. 2014). In the local computation phase, messages will be passed very fast across local vertices using shared memory. In the next phase (synchronous phase), remote nodes (boundary nodes) that are connected by edges across the partitions, will be updated by exchanging messages. In fact, component-centric frameworks such as Giraph++ and GoFFish allow users to exploit this. Others like Giraph Unchained (Han and Daudjee 2015) also allow incremental forward progress within a future superstep based on partial messages that are received, even before the previous superstep completes. These straddle synchronous and asynchronous models.

Apart from these methods, a novel approach has been introduced by Xie et al. (2015) in the PowerSwitch system, which sequentially switches between synchronous and asynchronous execution mode, according to a heuristic prediction. This is because some properties of the processing might change as it progresses. For example, processing SSSP algorithm begins with just a few vertices active, which means that few messages are passed; this is suitable for an asynchronous model. But as the process goes further, the number of vertices involved in the computation will increase, which means that the number of messages passing among them increase as well, and this is suitable for a synchronous execution model. PowerSwitch can effectively predict the proper heuristic for each step and it can switch between the two modes if required.

3.4 Computational Models

Performing computation is at the heart of a graph processing system where data (vertex or edge) will be processed and updated. Computational models that are used in existing graph processing systems can be divided into two major groups: (1) two-phase models, and (2) three-phase models. Figure 7 shows the classification of these two models with examples from each group. The computational model of some systems is a combination of these methods with other approaches.

3.4.1 Two-Phase Computational Models. There are usually two functions that are applied on data (vertices or edges) in a two-phase computation model. Signal-collect approach is the first two-phase programming model for large scale graph processing on the semantic web within a system with the same name (signal/collect) (Stutz et al. 2010). Computation in the vertices are completed by collecting the signals that are coming from edges and performing some processing on them using the vertex's state and then sending (signaling) their adjacent vertices in the compute graph. Signal/collect has been implemented for working with both synchronous and asynchronous execution models. In both models, some parameters should be set to determine when the computation should be stopped: *signal_threshold* and *collect_threshold* parameters in which a minimum level of

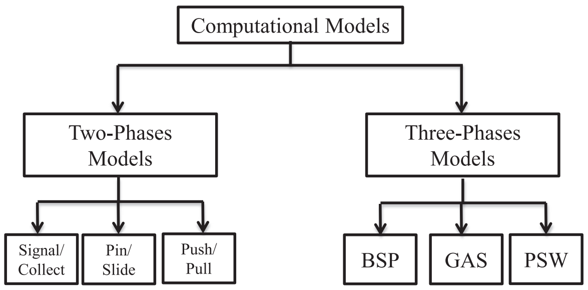


Fig. 7. Classification of computational models in graph processing systems.

"importance" will be set for the execution, and a *num_iterations* parameter, which is the number of iterations in synchronous mode, and *num_ops,* that is, the number of executed operations in asynchronous mode. All these parameters should be set by the user.

Pin-and-slide was first introduced by TurboGraph (Han et al. 2013) for parallel execution of large datasets on a single machine. A pin-and-slide mechanism consists of a graph dataset, a buffer pool, and two different threads, as explained in Section 3.1, callback threads, and execution threads. When the buffer manager is being sent an asynchronous input/output by a callback thread, it sends the demand to the FlashSSD via the operating system, after which the control of execution goes back to the calling thread immediately. The main goal in this system is to reach all related adjacency lists efficiently. To achieve this goal, first, the pages that contain these adjacency lists should be identified. The most important challenge here is pinning large adjacency (LA) pages, which means that a number of smaller adjacency pages must be unpinned first, then the LA page can be pinned. To overcome this challenge, LA pages will be pinned when all related LA pages for a big vicinity list are completely loaded to maximize the buffer exploiting. When execution threads or callback threads terminated the processing of a page, this page will be unpinned and an asynchronous input/output demand will be sent to the FlashSSD by the execution thread. As soon as the processing has been completed, the execution window can be slid by the size of the pinned pages in the buffer (Han et al. 2013).

Nguyen et al. (2013) have used another two-phase approach called push/pull (PP) styles. The value of an active vertex will be pushed (flowed) from that vertex to its neighbors, which is more like scatter phase. The pull style occurs when the data from an active vertex's neighbors flow into that vertex, which is more like a gather phase. In an algorithm like SSSP, the push-style is applied to the active vertex neighbors by updating the destination label of the siding nodes of the active vertex by doing a relaxation with them; and the pull-style function updates the destination label of the active vertex by doing relaxation with all neighbors of that node. The pull mode also needs less synchronization because there is just one writer for each active vertex. KineoGraph (Cheng et al. 2012) is another system that uses this model for computation.

3.4.2 Three-Phase Computational Models. BSP is a parallel programming and also the most representative model in this category that has been used in several graph processing systems (Valiant 1990; Malewicz et al. 2010; Salihoglu and Widom 2013; Vaquero et al. 2013; Khayyat et al. 2013). To deal with the scalability challenges of parallelizing tasks across a number of workers, BSP, which utilizes an MPI, was developed. In BSP, as a vertex-centric computational model, each node is able to have two modes of "active" or "inactive." The computation comprises a series of supersteps that come with a synchronization hurdle between them. So, in each superstep: (1) the node that is involved in computation obtains its adjacent nodes' updated values from the last superstep (except the first superstep), (2) then the node will be updated based on the obtained values, and (3) the node forwards its updated value to its neighbors that will be available for them in the next iteration. In each iteration, a vertex may choose to vote to halt, in case it does not receive any messages from its neighbors or it has reached a locally stable state. That means it will not participate in the processing anymore until it receives new messages that convert its state from inactive to active. So, in each iteration, only active vertices will be computed. If there is no active vertex in the graph, then the computation is finished.

Some research has modified the BSP model and introduced new models. For example, temporally iterative BSP (TI-BSP) (Simmhan et al. 2015) is a computational model for time-series graphs on a subgraph-centric model such as GoFFish. It has used BSP as a building block to support the design pattern. TI-BSP is a series of BSP loops (nested supersteps) in which each *outer loop*, as a *timestep*, runs on one graph instance in time. Supersteps using a subgraph programming model form the *inner loop* that operates over a single instance. As a result, the design pattern will be decided based on the order of timestep execution and the messages between them.

There is another BSP model that stands for BiShard Parallel, and has been introduced by the single machine based system, BPP (Najeebullah et al. 2014b), to empower full CPU parallelism for graph processing. This model also has three phases that include (1) loading a sub-graph of the large graph from disk, (2) performing compute operations on the sub-graph and updating the values of edges and vertices, and (3) writing back the modified values on disk. BiShard Parallel performs under an asynchronous execution model and needs more disk space compared to one shard mechanism that was used in GraphChi, because two copies of each edge are managed in this model.

GAS is another three-phase computational model that was introduced by PowerGraph. The data about the adjacent nodes and edges is obtained and collected using a general summation over all adjacent vertices and edges of a vertex in the *gather* phase. The *apply* operation should be defined by the user and must be both associative and commutative, and can vary from a numeric summation to the aggregation of data across all adjacent edges and vertices. The results from gather phase are used to update the central vertice values in the apply phase. Finally, the recent data of the central vertex is used to renew the values on neighboring edges in the *scatter* phase. The critical challenge in this model is that graph parallel abstractions should be able to perform computations with high fan-in and high fan-out where both of them are specified by gather/scatter phases. GAS model is used to develop a runtime system mapping in parallel on GPUs as a graph application called GasCL (Che 2014). This model is like the one that has been used in systems such as Pregel and GraphLab, but in a different way.

GraphChi uses a different computation model called PSW. PSW is an asynchronous computation model that can efficiently process the graph with changeable edge value from disk, with a few number of non-consecutive disk access. PSW performs three phases for processing a graph as follows: it loads a subset of the graph from disk, then applies update operation on vertices and edges, and, eventually, the new updated values will be written on disk. The number of "reads" from disk is exactly equal to the number of "writes" to the disk in this model.

The comparison between two-phase and three-phase models shows that the two-phase model is generally used in frameworks with single machine architectures. Since these systems usually use asynchronous coordination, using the two-phase approach is more efficient as they do not need to wait for the synchronization barrier. On the other hand, three-phase approach is mostly used by distributed frameworks because one or two phases of the model will be affected after the global barrier. Therefore, hosts on such distributed systems have to wait for each other. However, very few distributed frameworks such as GraphLab (Low et al. 2012) and Trinity (Shao et al. 2013) tried to use three-phase computation mode while utilizing asynchronous coordination.

4 RUNTIME ASPECTS OF GRAPH FRAMEWORKS

4.1 Partitioning

Graph partitioning is a method in which graph data is divided into smaller parts with specific properties (Buluç et al. 2016). For example, in a *k-way* partitioning, the graph is partitioned into K smaller parts of equal size while minimizing the edge cuts between partitions. This is an NP-complete problem (Andreev and Racke 2004). Graph partitioning is a fundamental research problem and several reviews have been done on different methods and perspectives of graph partitioning (Buluç et al. 2016; Bader et al. 2013). In a graph processing system, partitioning is applied on the large graph to assign each smaller partition to a worker to be computed. The most important challenge in this context is, *"How do we partition the graph to achieve better cuts while taking load balancing and simplicity of computation into consideration?"*

Many novel heuristics have been proposed for partitioning large graphs. METIS (Karypis and Kumar 1995), for instance, is a popular tool that uses multi-level partitioning. It is able to perform high quality partitioning that decreases the overall communication (edge cuts) and reduce imbalances across partitions. However, METIS is computational costly and high random access needs make it unsuitable for large graphs. ParMETIS is a parallel MPI-based version of METIS that mitigates some of these performance limitations.

There are distributed partitioning algorithms, some of which have been implemented on top of graph processing frameworks as well. Spinner (Martella et al. 2015), for instance, runs on top of Giraph and utilizes an iterative node migration approach using a label propagation algorithm to deal with scalability and changing partitions. It allows Spinner to scale to billion-vertex graphs by avoiding costly synchronization among vertices. Blogel implements a GVD partitioner using a vertex-centric computing method by operating as a multi-source BFS. It partitions the vertices into blocks using multi-source BFS over linear workloads.

Some graph processing systems create additional topological constructs on top of the partitioned graph. In GoFFish, which is a subgraph-centric framework, each partition may have more than one subgraph (WCC), and these subgraphs by definition cannot have an edge between them. So GoFFish has a post-processing stage once the graph is partitioned, in which it identifies all subgraphs within a partition that form the unit of processing during the programming model's execution.

In general, two partition creation strategies can help to improve the runtime performance during distributed graph processing: (1) creating more partitions than workers and allocate more than one partition to each worker and (2) allocating one partition per worker, yet using multiple workers on each processing host (Salihoglu and Widom 2013). We next discuss alternative perspectives toward partitioning to support graph processing systems, also shown in Figure 8.

4.1.1 Static Partitioning vs. Dynamic Partitioning. Several graph processing frameworks utilize static partitioning, which means that they consider the graph and the processing environment to stay unchanged (Schloegel et al. 2001). These systems assume that the I/O bandwidth, latency, processing units, and the graph itself are constant and predictable. So, this method of one-time *a priori* graph partitioning is easy to program and load balancing can be easily achieved, if the assumptions hold and the problem domain does not change (Elsner 2002).

On the other hand, dynamic repartitioning assumes that runtime behavior of an algorithm, the processing environment, and even the graph itself can be variable. They try to repartition the current state of the graph according to the system and algorithm behavior at a given point in time, and assign them to the available workers. This approach has been used for graph databases and a number of graph processing systems (Salihoglu and Widom 2013; Nicoara et al. 2015). Dynamic repartitioning can be applied in-flight when, for instance, workers are waiting for a straggler

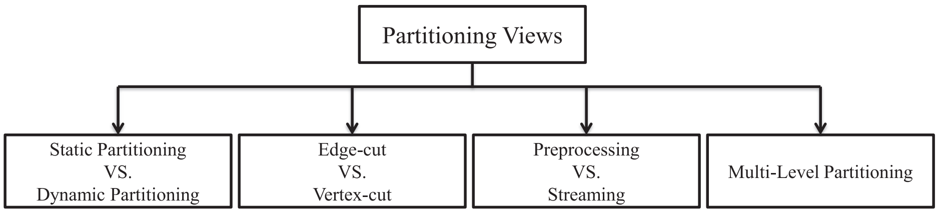

Partitioning Views
Static Partitioning VS. Dynamic Partitioning
Edge-cut VS. Vertex-cut
Preprocessing VS. Streaming
Multi-Level Partitioning

Fig. 8. Partitioning views in graph processing systems.

worker to finish. In this situation, the vertices that have been assigned to the slowest worker can be repartitioned and reassigned to other idle workers to be computed faster and also balances the workload to reduce overall runtime. This does need support for dynamic migration by the graph framework (Salihoglu and Widom 2013). Another reason to use dynamic repartitioning is when the number of active vertices in the graph change due to mutations or when the algorithm is non-stationary, causing vertices to become inactive, and it is suitable for iterative programming models such as Pregel (Xu et al. 2014).

According to GPS (Salihoglu and Widom 2013), three major questions should be answered in a dynamic repartitioning process: (1) Which nodes should be reallocated? (2) When and how to transfer the reallocated nodes to their new workers? (3) How to place the reallocated nodes? These decisions can impose a heavy cost and affect the overall runtime. Some researchers have also shown that dynamic repartitioning does not offer significant performance improvements except under particular conditions. For example, the vertices in a PageRank algorithm are always active, which makes dynamic repartitioning moot due to predictable and stationary load through the entire application's lifetime (Lu et al. 2014). But despite these concerns, systems such as GPS, xDGP (Vaquero et al. 2013), Mizan and XPregel (Thien Bao and Suzumura 2013) have incorporated dynamic repartitioning and migrate the vertices synchronously across the workers, along with their incoming messages.

4.1.2 Edge-Cut Partitioning vs. Vertex-Cut Partitioning. Vertex-centric (edge-cut) frameworks partition the graph by assigning vertices to partitions and cutting some edges across partitions in the process, while minimizing the number of such crossing edges. This is a common and well-supported partitioning approach. On the other hand, edge-centric (vertex-cut) frameworks partition the graph by cutting vertices and assigning edges to each partition. This approach minimizes the number of crossing vertices, which is useful for many real-world graphs that have a power-law degree distribution to balance edges across the partitions well (Gonzalez et al. 2012; Abou-Rjeili and Karypis 2006). As was shown in Figure 10, vertices shared by edges belonging to different partitions would be cut and replicated across all the partition. One copy of the vertex is considered as the master and other copies are ghosts or mirrors. When updated, each ghost vertex sends its locally updated value to the master, and the master vertex applies updates from all ghost vertices to itself and sends the globally updated value back to the ghost vertices. We can see that many messages need to be passed across the network to maintain the cut vertices up to date.

To avoid this, PowerGraph does partitioning based on high-degree vertices of the graph and many systems have adopted such edge-centric partitioning (Kim and Candan 2012; Rahimian et al. 2014; Xin et al. 2013; Jain et al. 2013). There are also a number of additional optimizations that have been done (Rahimian et al. 2014; Kim and Candan 2012). Authors in Feige et al. (2008) and Bourse et al. (2014) have investigated several edge-centric (vertex-cut) approaches with vertex-centric (edge-cut) approaches and found that in many cases the former outperforms the latter. The reason is that because the degree distribution is skewed, balancing the number of vertices in each partition does not guarantee workload balance; therefore, for natural graphs (that have power-law degree distribution), vertex-cut partitioning approach can obtain better workload balance. They also conclude that for any edge-cut, a vertex-cut can be constructed directly which needs strictly less storage and communication.

4.1.3 Pre-Processing vs. Streaming. As seen in Section 2.1, there might be a pre-processing phase before the computation or the main processing starts. In the pre-processing approach, the large graph, which is present on disk, will be partitioned before entering the system. Single-machine frameworks such as GraphChi (Kyrola et al. 2012), TurboGraph (Han et al. 2013), BPP (Najeebullah et al. 2014b), and CuSha (Khorasani et al. 2014) use this method because they do not have enough memory to keep all the processing states in the single system. So, they partition the graph before starting the processing to help cope with large graphs. It also limits the partitioning operation, which can be costly, to a single time. Distributed frameworks like GoFFish do partitioning and subgraph identification in such a pre-processing phase for the same reason.

In streaming partitioning, the graph is partitioned once or as it is loaded into the graph processing system. The graph data enters the system sequentially, say a vertex and its adjacency list at a time, and the vertex is mapped to a partition on the fly. In this model, the order in which the vertices enter the system is important as each placement depends on the previous placements (Stanton and Kliot 2012). Streaming can also benefit from a pre-processing model of partitioning, where specific vertex or edge ordering has been performed. *Random partitioning*, *round-robin*, and *range algorithms* are the three most common algorithms for steaming whereas linear deterministic greedy (LDG) (Stanton and Kliot 2012) and FENNEL (Tsourakakis et al. 2014) are two greedy heuristics that improve the performance and quality of such online partitioning.

4.1.4 Multi-Level Partitioning. Some graph processing frameworks have proposed multi-level approaches for partitioning the graph. In this method, there will be more than one strategy for partitioning that may even be applied to the graph in different times. GridGraph (Zhu et al. 2015), for example, is a single-server block-based framework that uses a two-level hierarchical method to partition a given graph. First, it partitions the graph once at a pre-processing phase in which it divides the graph into 1D-partitioned vertex and 2D-partitioned edge chunks, respectively. Then, at the runtime, it uses a dual sliding windows approach to partition the graph by streaming the edges and applying updates on vertices, which guarantees the locality and improves I/O.

GraphMap (Lee et al. 2015) is a distributed framework that also uses a two-level partitioning mechanism to improve the locality and workload balancing. In the global level, GraphMap utilizes a hash method to partition the graph and assign the partition blocks to the workers and, in the local level, it applies range partitioning to each block partitions of a worker. It has also been designed to use other partitioning techniques such as METIS and ParMETIS both on-disk and in-memory.

Using multi-level partitioning has two edges. It can worsen the performance of the system by prolonging the execution time and unnecessary computations, or it can improve the performance particularly when it is applied as a layered mechanism. For example, apply one partitioning technique to the entire graph and at the same time perform another technique on the partitions on workers, which can be designed asynchronously.

4.2 Communication Models

Graph processing systems use different approaches to communicate among their vertices, edges, and partitions. In this section, we discuss these methods as shown in Figure 9 and enumerate the advantages and disadvantages of each method.

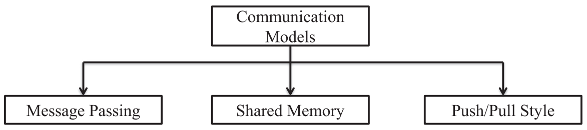


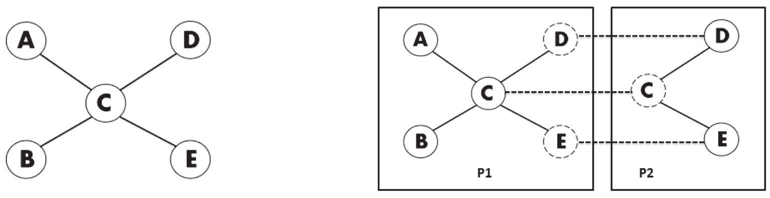
Fig. 9. Communication models in graph processing systems.

4.2.1 Message Passing. Many distributed graph programming models offer explicit or implicit communication between their entities. In the message passing technique, communication is carried out by sending messages explicitly from one entity to another in the graph. The entity can be a vertex, edge or a component in a local or remote partition (Malewicz et al. 2010). Message passing is performed in many graph processing systems using communication libraries. For example, Pregel allows developers to pass messages from a vertex in the graph to another by calling an API. As part of the BSP execution model, messages sent in one iteration are received by the destination vertex in the subsequent iteration using bulk messaging. The receiving vertex updates its state based on incoming messages and sends its modified state to one or more of its neighbors by sending additional messages. Each source vertex maintains a list of its adjacent vertex IDs or outgoing edges IDs. Further, vertices also have queues where incoming messages from its neighbors and outgoing messages to its neighbors are stored between superstep boundaries.

A message passing model of communication is used by many graph processing frameworks and architectures, including vertex-centric, edge-centric, and component-centric frameworks. Programming using synchronous message passing is also intuitive and the complexity is limited to an API to send a message to a destination entity, which is a familiar model for many programmers (Hudak 1989). Although message passing is common in the frameworks with synchronous model of execution, it can be used in asynchronous execution models as well. Asynchronous message passing method is used extensively between vertices in the same partition or subgraph where they do not need to wait for other vertices to send their message in-bulk. Vertices and subgraphs can communicate asynchronously while communication between partitions can be synchronous. However, the asynchronous model of message passing brings more complexity to the programming paradigm because it requires more resources for storing and rebroadcasting data in a system where its components do not execute concurrently (Gehani 1990). On the other hand, buffer management is an important issue that should be considered by message passing implementation. Issues like: How many buffers should each worker have? What should be the size of each buffer? When should a buffer block a sender? And what if the buffer is full but there are new messages coming? This model also has overhead because of the number of message replicas that exist in the network.

The Message Passing Interface (MPI) (El-Rewini and Abd-El-Barr 2005) is a common protocol used in graph processing systems, and is used by systems such as Pregel, GraphLab, Piccolo, and Mizan. Portability and velocity are two significant advantages of using MPI, where creating overhead is its most noticeable disadvantage. Communication can be done by passing actual messages between servers or by serialization. For Java-based systems like Giraph and Hama, protocols such as Thrift (Apache Software Foundation 2008) and ActiveMQ (Apache Software Foundation 2007) can be used for message passing. They utilize remote procedure call (RPC) to communicate seamlessly without the need to change the messages structures. Also protocols such as Avro (Apache Software Foundation 2012), and Protocol Buffer (Google 2008) can be used for serialization by which the data will be serialized to be able to be sent between different platforms.

Systems also propose optimizations on top of these standard messaging libraries to reduce the communication overhead and minimize the runtime of the algorithm by reducing the number and



(a) Sample graph (b) Shared memory model

Fig. 10. Shared memory model with ghost (mirror) vertices.

size of messages that are passed. For example, GasCL has considered two different message buffering strategies: first, messages from the same source are stored together, second, messages for the same destination are stored together. The second approach is more common and when a message is dispatched from the origin, it is instantly put down to the right target node. It also uses a reverse edge index to store the message, which utilizes array offset to facilitate message combining. Giraph++ (Tian et al. 2013) introduces two types of messages in its hybrid model. "Internal messages" that are messages sent from a vertex within a partition to another vertex within the same partition, and "external messages" that are messages sent from the vertices of one partition to another partition. It provides two incoming message buffers for each vertex inside a partition as well: $inbox_{in}$ for internal messages and $inbox_{out}$ for external messages. In GPS (Salihoglu and Widom 2013), instead of sending multiple copies of the same message to multiple vertices in another partition, the system only sends one message to the remote partition and then, in the remote partition, the message will be copied to the vertices that need to receive it. This can dramatically reduce the network traffic.

4.2.2 Shared Memory. Using shared memory for communication is well suited for frameworks running on a single server, but can also be used for distributed systems in place of explicit message passing. In this model, the memory location can be simultaneously accessed by multiple processing modules, including both read and write to that location. Contrary to message passing, the shared memory method avoids extra memory copying and intermediate buffering. As single machine frameworks have limited memory and CPU resources, this shared-memory model that is often natively supported by the operating systems is preferred (Nitzberg and Lo 1991). Locks or semaphores are usually used in this model to prevent race conditions because concurrent tasks can read and write to the same memory (Low et al. 2012; Chen et al. 2008). To maintain memory consistency, shared memory machines provide mechanisms for invoking the appropriate job (sequential consistency) or reordering a collection of jobs to be executed consecutively (relaxed consistency).

In distributed systems, it is also possible to have a distributed shared memory, where changes to memory locations are internally transferred using messages between different machines. From the programmer's points of view, they only perform memory accesses and the development is easier as explicit messages need not be passed. The concept of data "ownership" is lacking as well since anyone can write to that location. On the other hand, data locality cannot be controlled easily, and if many remote workers access a particular memory location, it puts pressure on the processor and memory holding that location and can also lead to higher bus traffic and cache misses.

Virtual shared memory can be realized by using ghost vertices or mirrors, which are the copies of distant adjacent vertices (Low et al. 2010). One machine keeps the main vertex and another machine works on copies of this vertex. Main vertex and ghost copies are shown in Figure 10. By keeping the mirror copies immutable during the computation with distributed write locking or an accumulator, the consistency is maintained (Low et al. 2012; Power and Li 2010). Both GraphLab and PowerGraph use this approach for communication. In particular, this is suitable for edge-centric frameworks because vertices should be cut in these frameworks and the partitioning is done based on edge-grouping. So, vertices can be easily cut (Figure 10). Trinity (Shao et al. 2013) is another memory-based graph processing system (Section 4.3) that uses ghost vertices for communication. The Trinity specification language (TSL) maps the data storage and graph model together. The parallel boost graph library is a parallel graph processing library also uses ghost nodes but with message passing mechanism (Siek et al. 2002).

4.2.3 Push/Pull Styles. A PP model is used with active messages. Active messages are those that carry both data and the operator that should be applied on them (Han et al. 2014; Zhang et al. 2012). This model is utilized by Beamer et al. (2012), direction optimization in BFS. The reason behind using this model is that the synchronization and communication in large-scale data is very expensive due to the poor-locality and irregular patterns of communication in graphs. To reduce the random communication and memory access on either shared-memory or distributed implementations, Beamer incorporates the conventional *top-down* BFS with a new *bottom-up* method. In the push style, the information flows (is pushed) from an active vertex to its adjacent vertices and in the pull style, the information flows (is pulled) from the neighbors of an active vertex to that active vertex. This kind of communication is using the PP computation model that was discussed in Section 3.4.1. In terms of consistency, the pull style is naturally consistent because the active vertex would be updated in this phase, but the push style needs to use locks for every neighbor's update. Active messages are sent asynchronously in this model and they will be executed when they are received by the destination vertex. The sending and receiving messages are even combined in a framework such as GRE (Yan et al. 2013) and it does not need to save intermediate states anymore. This mechanism can help in enhancing efficiency of algorithms such as PageRank (Gharaibeh et al. 2013; Shun and Blelloch 2013). It has been used by frameworks such as Ligra (Shun and Blelloch 2013) and Gemini (Zhu et al. 2016) on shared memory and distributed processing, respectively. A detailed analysis of PP approach has been provided in Besta et al. (2017), investigating the impacts of both PP mechanisms individually and also together on various graph algorithms and programming models. They have illustrated that a *PP dichotomy* approach can avoid extreme amount of locks in pushing and more memory access in pulling.

4.3 Storage View

Memory has become less of a problem these days as computing service providers such as Amazon are starting to provide machines with terabytes of memory. However, as described in Sharma et al. (2016), social networks such as Facebook and Twitter not only have graph of users but also graphs of connections between users, their likes, their locations, their posts and shares, photos, and so on that are heterogeneous. As a result, to store all these large graphs and datasets, a single server cannot provide enough space. Hence, many investigations have been done to process graphs on both single server and distributed environments such as clouds. Figure 11 shows various storage views in current graph processing systems.

4.3.1 Disk Based. According to a storage view of execution models, two common approaches can be used: (1) *disk-based* approach and (2) *memory-based* approach. A disk-based execution fetches the graph data from physical disks, not just when loading the graph initially but also actively writes and reads parts of the graph state to/from disk during the execution. The advantages of using a disk-based approach is that it is cheaper to add disk capacity rather than memory, some large graphs do not fit in distributed memory either, and one can persist the partial state of execution in the middle of the processing to enable recovery from faults (Chockler et al. 2009). Disk management is also easy, so many graph processing systems use this approach (Table 3).

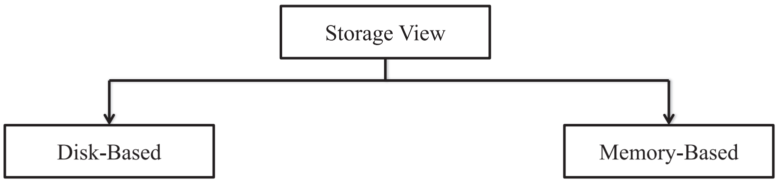


Fig. 11. Storage view.

On the other hand, the computation that is performed by graph algorithms is data-driven (Lumsdaine et al. 2007) and they need many random data accesses, and hard disks are still slow and inefficient compared to main memory. One of the challenging issues for disk-based graph processing is how to make disk access more efficient. BPP (BiShard Parallel Processor) (Najeebullah et al. 2014a), for example, provides a disk-based engine for processing large graphs on a single server. A novel storage structure called BiShard (BS) has been introduced, which divides the graph into subgraphs containing equal number of edges, and stores the in-edges and out-edges independently. This technique decreases the number of non-sequential I/O considerably and has two advantages compared to the single-shard storage mechanism that is presented by GraphChi. First, by storing in-edges and out-edges separately, access to each of them becomes independent and the system does not need to read the whole shard for every subset of vertices. Second, each edge has two copies in BS (one in each direction), which eliminates race condition among vertices to access their edges. Furthermore, BPP uses a novel asynchronous vertex-centric parallel processing model that leverages BS to provide full CPU parallelism for graph processing.

Other frameworks such as Giraph also support out-of-core execution using disk. When a graph is too big to fit into main memory (like small clusters) or a certain algorithm creates very large message sets (many messages or large ones) these frameworks can spill the excess messages or partitions to disk, later to be incrementally loaded and computed from disk. In addition, some frameworks such as FlashGraph (Zheng et al. 2015) and PrefEdge (Nilakant et al. 2014) use SSD instead of HDD to make data transmission and computation faster for out-of-core computation.

4.3.2 Memory Based. In the memory-based approach, the graph and its states are exclusively stored in memory during runtime for storing and processing the big data. For example, Giraph runs the whole computation in memory and reaches the disk for checkpointing and I/O, and Blogel keeps all neighbors of a typical high-degree vertex in the same block to be processed by in-memory algorithms and avoid message passing. The most important benefit of this approach is that using RAM or cache for processing is much faster than disk-based approach since the CPU can access memory much quicker than disk (Mittal and Vetter 2015). However, memory is much more expensive than spinning disks and this becomes challenging when we consider larger graphs. So, memory-based systems must be efficient in retaining only relevant data in memory and in a compact form. Memory-based models also have less scalability than disk-based models, especially in a single machine system.

In GoFFish, the framework only loads a subset of properties for a given property graph from disk into distributed memory based on those attributes that are used by the algorithm, in addition to the complete topology of the graph that is always loaded (Jamadagni and Simmhan 2016). This limits the memory footprint of the graph application during runtime. Many memory-based systems also use columnar representation since this offers a compact representation of data. Zhong and He (2013b) have indicated that GPU acceleration cannot reach considerable speedup if the data has to be loaded from disk because of the I/O costs that are themselves comparable to the total query runtime.

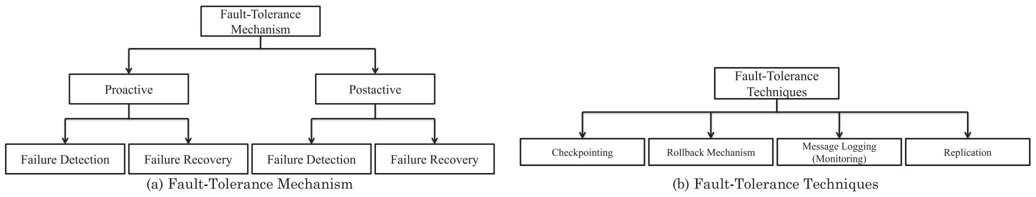

Fault-Tolerance Mechanism
Proactive
Postactive
Failure Detection
Failure Recovery
Failure Detection
Failure Recovery
(a) Fault-Tolerance Mechanism
Fault-Tolerance Techniques
Checkpointing
Rollback Mechanism
Message Logging (Monitoring)
Replication
(b) Fault-Tolerance Techniques

Fig. 12. Fault-tolerance in graph processing systems.

Microsoft's Trinity (Shao et al. 2013) is a distributed graph processing engine over a memory cloud. It is supporting both online query processing, which requires low latency (finding a path between users in a social network), and offline query processing, which requires high throughput (PageRank). Trinity uses TSL for communication that supports both synchronous and asynchronous modes. It stores objects as blobs of bytes that are economical and compact, with no serialization and deserialization burden. As a storage infrastructure, it structures the memory of numerous hosts to a distributed memory address space, which is universally addressable for maintaining huge graphs. Trinity has three main components, including (1) *slave* that stores graph data and computes on them, (2) *client* that acts as a user interface between Trinity and the user that communicates with Trinity proxies and slaves through the APIs provided by Trinity library, and (3) *proxy,* that is an optional component for handling messages as a middle tier between clients and slaves. These components, along with other features like user-defined communication protocol, graph schema, and computation models through TSL, enable Trinity to process the graph efficiently on memory cloud.

4.4 Fault Tolerance

Fault tolerance enables a system to continue performing properly even if some of its components face failures (Elena 2013). Since graph processing systems are created from distributed and commodity components, it is possible that components confront failures, which in turn will affect the execution and correctness of the applications. To improve the reliability and robustness of these systems, several techniques have been developed to support error handling and fault tolerance of the graph framework. Figure 12 shows the techniques that are used in many graph processing systems.

Error handling in a graph processing system, as with other systems, has two main phases: (1) *failure detection,* in which the system discovers the error, and (2) *fault recovery,* in which the system tries to resolve the problem and resume the operation. Numerous research has been done on various fault-tolerance techniques on parallel and distributed systems (Kavila et al. 2013; Treaster 2005; Elnozahy et al. 2002). In Treaster (2005), for example, two types of components in an application, called central components and parallel components, are investigated, where both mostly use rollback and replication methods for fault recovery. On the other hand, some graph processing systems do not support any error handling because it increases the complexity of the system, and the overheads can strongly affect the execution time.

Most graph processing systems use *checkpointing* and *rollback* mechanisms (Egwutuoha et al. 2013) for failure recovery, such as Pregel and Pregel-like systems like Giraph. Pregelix (Bu et al. 2014), for instance, checkpoints states to HDFS at any superstep boundary that is selected by the user. The checkpointing applies to vertices and messages at the end of each superstep and ensures that the user does not need to know anything about the failure. Whenever a host or disk failure occurs, the unsuccessful machine will be added to a blacklist. For recovery, Pregelix reloads the state of the latest checkpoint to a set of "failure-free" workers that is periodically updated.

Piccolo (Power and Li 2010) uses a global checkpoint-restore method to recover from failures by providing synchronous and asynchronous checkpointing APIs. Synchronous checkpoints are suitable for iterative algorithms such as PageRank where the state in different iterations are decoupled by global barriers and it is adequate to checkpoint the state every few iterations. But asynchronous checkpoint is used to save the state of long running algorithms, such as distributed crawler, periodically. Piccolo also utilizes Chandy-Lamport (CL) distributed snapshot algorithm (Chandy and Lamport 1985) for checkpointing. Once a failure is detected in a worker, the master will reset the status of all other machines and recover the operation from the latest finished universal checkpoint. The interior status of the master will not be checkpointed in Piccolo.

PowerGraph is another system that uses snapshots of the data-graph for fault-tolerance. The synchronous engine in PowerGraph creates the snapshot at the end of each superstep and before the start of the next superstep while the asynchronous engine suspends the execution of the system to create the snapshot. Many of these systems provide task rescheduling after the recovery phase. Some systems such as Pregel, Piccolo, and GraphLab benefit from rollback, which allows them to continue the computation from the point that failure happened while, in a number of systems, fault recovery is not completely provided and they need to restart the processing from scratch (Han et al. 2014; Krepska et al. 2011; Plimpton and Devine 2011). All these mechanisms are post-active fault tolerance approaches, which means they handle the failure after it has happened.

Trinity (Shao et al. 2013) uses message logging and replication for pro-active fault-tolerance, where the failure will be considered before scheduling and releasing a job for execution. Trinity utilizes heartbeat messages to proactively detect failures in machines. In addition, machines that unsuccessfully try to access the address-space in other machines also report the inaccessible machine to the master and await for the addressing table to be updated before retrying the memory access. Meanwhile, in the recovery phase, the master reloads the data in the failed machine to another machine, updates the addressing table, and distributes it. Trinity provides checkpointing after every few iterations for synchronous BSP-based computations and provides "periodic interruption" mechanisms to generate snapshots in asynchronous computations. The buffered logging approach that is suggested in RAMCloud (Ousterhout et al. 2010) has been used in Trinity to recover from failures in online queries while, for read-only enquiries, it only restarts the faulty machine and loads the data again from the steady disk storage. GraphX (Xin et al. 2013) uses lineage-based fault tolerance that assumes its RDDs cannot be updated but only created afresh. It has a very light overhead compared to the systems that use checkpointing as well as arbitrary dataset replication. So, it attains fault-tolerance without explicit checkpoint recovery while retaining in-memory performance of (Zaharia et al. 2012).

4.5 Scheduling

Scheduling techniques help assign and manage jobs on the system resources (Pinedo 2012). This is particularly useful in parallel and distributed multitasking systems in which several computations have to be done on a limited number of resources. In graph processing systems with large scale graphs having billions of vertices and edges, the vertex or edge (depending on the programming model) will need to be scheduled for computation on a processing host. Typically, collections of vertices or edges are grouped into a coarser unit for scheduling, such as a partition or a subgraph, and it is the coarse unit that is actually scheduled on a CPU core. Within a processor, there may be multiple threads that execute individual vertices or edges in partition, leveraging, say, vertex level parallelism in a vertex-centric model.

According to Doekemeijer and Varbanescu (2014), three different types of scheduling methods have been used for graph processing, in general, which is shown in Figure 13.

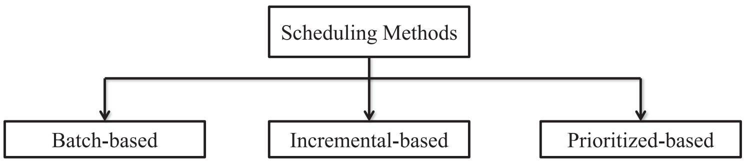


Fig. 13. Graph-processing scheduling methods.

In batch scheduling method, the entire graph would be scheduled for processing across computing resources. This is more beneficial in the bulk iteration model of computing such as Pregel. There is no priority or precedency in executing the partitions of the graph and they will be processed in any arbitrary order (Chen et al. 2012). This model has been used widely in dataflow frameworks such as Hadoop, Haloop (Bu et al. 2010), and Twister (Ekanayake et al. 2010). There is always a preferred situation, like a limited number of iterations that are used as a condition for finishing the process.

Scheduling can be done once at the beginning of the application or redone at the start of each superstep, e.g., Giraph allows partitions of the graph to be mapped to workers both at the start of the application and at each superstep, while TOTEM maps partitions to workers at the start of the application and retains that mapping. Remapping of partitions to workers as the application is executing also requires the ability to migrate both the graph and its updated state and messages to different workers. The mapping of vertices and edges to partitions may also change as a result. For example, Mizan migrates the vertices from busy workers to the one with fewer vertices to load the balance, and GPS repartitions the graph to distribute the load among idle workers during the computation.

Another aspect is whether the partitions are mapped to a static set of compute resources or the resources themselves can be elastic over the execution of the application. For example, Dindokar and Simmhan (2016) look at mapping partitions to an elastic set of VMs based on the expected computational complexity of the partition for stationary and non-stationary graph algorithms.

In contrast, in a prioritized scheduling method, jobs will be processed according to a priority condition that is defined by the user. A system such as Maiter (Zhang et al. 2012) shows that using this method results in quicker convergence for many graph algorithms. For instance, a defined prioritizing function can schedule jobs based on the number of vertices in each partition. In GoFFish, the largest subgraphs in a partition are executed first so that the computing of smaller subgraphs can be interleaved with the message passing from the large subgraph. Prioritized scheduling can be helpful in processing imbalanced workloads.

Doekemeijer and Varbanescu (2014) believe that incremental scheduling only processes a subdivision of data like active vertices. This model is used in a number of graph processing systems, e.g., Stratospher (Alexandrov et al. 2014) and GraphLab (Low et al. 2012), in which the processing continues until there are active vertices.

5 GRAPH DATABASES

A graph database is one where the data is natively stored as a graph structure that can be queried upon (Angles and Gutierrez 2008). The data itself is typically a property graph with not just vertices and edges but also name-value properties or labels defined on vertices and edges. Graph query models support different types of traversal queries such as path, reachability, and closure, in addition to filter queries over their properties (Jamadagni and Simmhan 2016; Sarwat et al. 2013). Graph databases contrast with relational databases that store graphs—the latter requires multiple joins for traversal of graphs rather than having direct references from a vertex to its neighbors

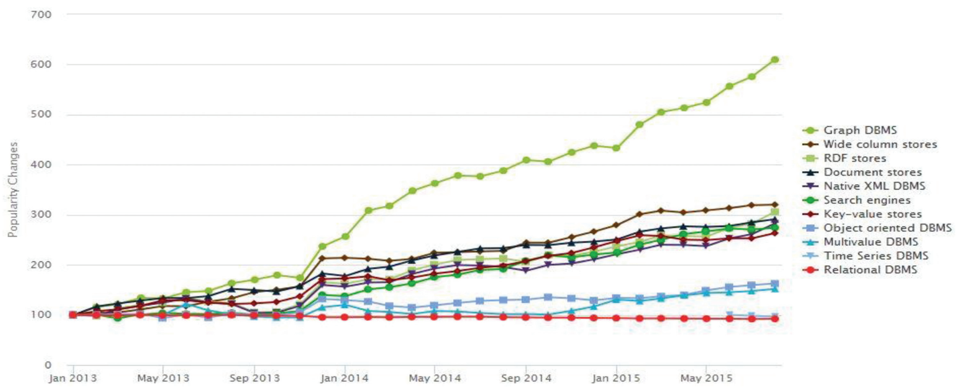


Fig. 14. Popularity changes in using databases.[2]

that allows for faster query processing in graph databases. Graph databases leverage the topological properties of graphs, including graph theory and query cost models, in answering the queries. The queries also provide a high-level declarative interface for processing and accessing the graph compared to a graph framework that requires users to write a program using their graph programing abstractions and executes it in batch (Vicknair et al. 2010). Another distinction from the graph frameworks discussed above is the need for low latency (*O(seconds)*) execution of hundreds of queries rather than high throughput analysis of single programs over large graphs. The aim of this section is not to provide a survey on graph databases, but to emphasize the increasing popularity of graphs and graph databases that provide a broader view of graph usages. For a detailed survey on graph databases, we refer the readers to works that appeared in *ACM Computing Survey* (Angels and Gutierrez 2008) and *VLDB Journal* (Kaoudi and Manolescu 2015).

Graph datasets are receiving more attention every day and several companies are starting to utilize graph databases to perform interactive queries to support their business needs. Even traditional database providers such as Microsoft have added extensions to their products by which the graph will be natively stored and queried inside the database on Azure SQL DB.[4] According to DB-Engines,[5] which is an industry observer, *"graph DBMSs are gaining popularity faster than any other database categories,"* which shows remarkable growth in the last few years (Figure 14).

Neo Technology (2015) is a popular graph database designed as an open-source NoSQL database. It supports ACID (Atomicity, Consistency, Isolation, Durability) properties by implementing a Property Graph Model efficiently down to the storage level. It is useful for single server deployments to query over medium sized graphs due to using memory caching and compact storage for the graph. Its implementation in Java also makes it widely usable. Besides the single server model, it also provides master-worker clustering with cache sharding for enterprise deployment. However, according to some reports, the scalability of the distributed version is not as good as even relational databases and it has deadlocks problems such as not being able to handle two concurrent *upserts* if they touch the same node[6].

[4]https://blogs.msdn.microsoft.com/sqlcat/2017/04/21/build-a-recommendation-system-with-the-support-for-graph-data-in-sql-server-2017-and-azure-sql-db.
[5]DB-Engines Ranking Per Database Model Category (August 5, 2015). Retrieved August 15, 2015, from DB-Engines: http://db-engines.com/en/ranking_categories.
[6]https://news.ycombinator.com/item?id=9699102.

OrientDB and Titan are two other well-used graph databases (OrientDB LTD 2015). OrientDB can save 220,000 records per second on ordinary hardware. It supports multi-master replication and sharing which give it better scalability. It also provides a security profiling system based on roles and users in the database. Titan[7] is another open-source distributed transactional graph database that provides linear elasticity and scalability for growing data, data distribution, and replication for fault-tolerance and performance. It supports ACID and different storage back-ends such as Apache Hbase (Apache Software Foundation 2015) and Apache Cassandra (Lakshman and Malik 2015). Titan also uses the Gremlin query language (Rodriguez 2009), in which traversal operators are chained together to form path-oriented expressions to retrieve data from the graph and modify them.

Twitter has developed its own graph database called FlockDB (Kallen et al. 2012) to store social graphs such as "who blocks whom" and "who follows whom." FlockDB is an open-source fault-tolerant distributed graph database that aims to support online data migration, add/delete/update operations, complicated sets of arithmetic queries, replication, archive/restore edges, and so on. In April 2010, the FlockDB cluster had stored more than 13 billion connections (edges) and supported a peak traffic of 20K writes per second with 100K reads per second (Green 2013). But it appears that FlockDB is not able to traverse graphs deeply as it is designed to only deal with Twitter's single-depth following/follower model and is not implementing the full stack of storage services.[8]

There is also research on distributed graph databases, though this is an emerging area. Horton+ (Sarwat et al. 2013) from Microsoft offers a graph query language that supports path, closure, and joint queries over property graphs. It converts the query into a deterministic finite automaton that is executed over a distributed database using a vertex-centric BSP model based on Giraph. GoDB (Jamadagni and Simmhan 2016) is another research database that leverages GoFFish to offer similar query capabilities over property graphs, but with support for scalable indexes and using a subgraph-centric model of execution that offers a much faster performance relative to Titan and Horton+. GBASE (Kang et al. 2011) introduces a *compressed block encoding* graph storage method that utilizes adjacency matrix representation to store homogeneous regions of graphs. It also uses a grid-based selection strategy for query optimization to provide quicker answers by minimizing disk accesses. Quegel (Yan et al. 2016) handles inquiries as "first-class citizens" by which the user is only required to determine the Pregel-like algorithm for a general inquiry. Then, it sets up the computing and processing of multiple inbound inquiries on demand.

There are several other open source and commercial graph databases such as HyperGraphDb (Kobrix Software 2015), AllegroGraph (Franz Inc 2015), InfiniteGraph (Objectivity Inc 2015), InfoGrid (NetMesh Inc 2015), JCoreDB Graph (Maier et al. 2015), ArangoDB (ArangoDB GmbH 2015), GraphBase (GraphBase Inc 2015), MapGraph (SYSTAP, LLC 2015; Fu et al. 2014), and Weaver[9]. All these projects try to provide modern solutions for storing and retrieving large-scale graph data, and it seems that this area is a very promising field of research and commercial investment for the future. Jouili and Vansteenberghe (2013) have compared some of these graph databases.

6 SYSTEM CLASSIFICATION AND GAP ANALYSYS

Table 3 presents the key graph processing systems with their characteristics according to the proposed taxonomy. The notations in the table for each category are as follows:

— *Programming model*: Vertex-centric (V), edge-centric (E), component-centric (C), path-centric (P), data-centric (Da), or block-centric (B).

[7] TITAN Distributed Graph Database. 2015. Home Page. Retrieved August 10, 2015, from http://thinkaurelius.github.io/titan.
[8] http://stackoverflow.com/questions/2629692/how-does-flockdb-compare-with-neo4j.
[9] Weaver. 2015. Home Page. Retrieved August 10, 2015, from http://weaver.systems.

Table 3. Overview of Existing Graph Processing Frameworks

Year	System	Programming Model	Architecture	Computational Model	Communication Model	Coordination	Storage
2009	PEGASUS (Kang et al. 2009)	N/A	D	N/A	DF	Synch	DB
2010	Pregel (Malewics et al. 2010)	V	D	BSP	MP	Synch	DB
2010	Signal/Collect (Stutz et al. 2010)	V	S	Signal/collect	MP	Both	DB
2010	Surfur (Chen et al. 2010)	V	D	Transfer-combine	MP	Synch	DB
2010	JPregel (Prakasam and Chandrasekhar 2010)	V	D	BSP	MP	Synch	DB
2010	GraphLab (Low et al. 2010)	V	S	N/A	SM	Asynch	DB
2010	Piccolo (Power and Li 2010)	Da	D	Three phases	Dataflow	Synch	DB
2011	GoldenOrb (Cao 2011)	V	D	BSP	SM	Synch	DB
2011	GBase (Kang et al. 2011)	E	D	N/A	Dataflow	Synch	DB
2011	HipG (Krepska et al. 2011)	V	D	BSP	SM	Both	DB
2012	Giraph (Apache Software Foundation 2012)	V	D	BSP	MP	Synch	DB
2012	Distributed GraphLab (Low et al. 2012)	V	D	GAS	SM	Both	DB
2012	KineoGraph (Cheng et al. 2012)	V	D	Push/pull	MP	Synch	MB
2012	PowerGraph (Gonzalez et al. 2012)	E	D	GAS	SM	Both	DB
2012	Sedge (Yang et al. 2012)	V	D	BSP	MP	Synch	DB
2012	GraphChi (Kyrola et al. 2012)	V	S	PSW	SM	Asynch	DB
2013	TOTEM (Gharaibeh et al. 2013)	V	H	BSP	Both MP and SM	Asynch	MB
2013	Mizan (Kayyat et al. 2013)	V	D	BSP	MP	Synch	DB
2013	Trinity (Shao et al. 2013)	V	D	TSL	SM	Asynch	MB
2013	Grace (Wang et al. 2013)	V	S	Three phases	MP	Asynch	DB
2013	GPS (Salihoglu and Widom 2013)	V	D	BSP	MP	Synch	DB
2013	Giraph++ (Tian et al. 2013)	C	D	BSP	Both MP and SM	Both	DB
2013	Naiad (Murray et al. 2013)	V	D	Timely dataflow	SM	Both	MB
2013	PAGE (Shao et al. 2013)	V	D	Partition-aware	MP	Synch	DB
2013	Stratospher (Ewen et al. 2013)	V	D	Push/pull	Dataflow	Synch	DB
2013	TurboGraph (Han et al. 2013)	V	S	Pin-and-slide	SM	Asynch	DB
2013	xDGP (Vaquero et al. 2013)	V	D	BSP	MP	Synch	DB
2013	X-Stream (Roy et al. 2013)	E	S	Scatter-gather	MP	Synch	DB
2013	GiraphX (Tasci and Demirbas 2013)	V	D	BSP	SM	Asynch	DB
2013	GraphX (Xin et al. 2013)	E	D	GAS	Dataflow	Synch	MB
2013	Galois (Nguyen et al. 2013)	V	S	ADP	SM	Asynch	DB
2013	GRE (Yan et al. 2013)	V	D	Scatter-Combine	MP	Synch	DB
2013	Ligra (Shun and Blelloch 2013)	C	S	Push-pull	SM	Asynch	MB
2013	LFGraph (Hoque and Gupta 2013)	V	D	N/A	SM	Synch	MB
2013	PowerSwitch (Xie et al. 2015)	V	D	Hybrid	SM	Both	DB
2013	Presto (Venkataraman et al. 2013)	V	D	N/A	Dataflow	Synch	DB
2013	Medusa (Zhong and He 2013a)	V	H	EMV	MP	Synch	MB
2014	RASP (Yoneki et al. 2014)	V	S	Scatter-gather	SM	Asynch	DB

(Continued)

Table 3. Continued

Year	System	Programming Model	Architecture	Computational Model	Communication Model	Coordination	Storage
2014	GoFFish (Simmhan et al. 2014)	C	D	Iterative BSP	Both MP and SM	Synch	MB
2014	GasCL (Che 2014)	V	H	GAS	MP	Synch	MB
2014	CuSHa (Khorasani et al. 2014)	V	H	GAS	SM	Asynch	MB
2014	BPP (Najeebullah et al. 2014b)	V	S	BSP	SM	Asynch	DB
2014	Imitator (Wang et al. 2014)	V	D	BSP	MP	Synch	DB
2014	GraphHP (Chen et al. 2014)	V	D	BSP	MP	Synch	DB
2014	PathGraph (Yuan et al. 2014)	P	S	Scatter-gather	SM	Asynch	DB
2014	Seraph (Xue et al. 2014)	V	D	GES	MP	Synch	DB
2014	GraphGen (Nurvitadhi et al. 2014)	V	H	N/A	SM	Synch	MB
2014	Blogel	B	D	N/A	MP	Synch	MB
2015	Pregelix (Bu et al. 2014)	V	D	Join-operator based	MP	Synch	DB
2015	FlashGraph (Zheng et al. 2015)	V	S	BSP	Both MP and SM	Asynch	DB
2015	GraSP (Battaglino et al. 2015)	V	D	N/A	MP	Synch	MB
2015	Chaos (Roy et al. 2015)	E	D	GAS	MP	Synch	DB
2015	GraphMap (Lee et al. 2015)	V	D	BSP	MP	Synch	DB
2015	GridGraph (Zhu et al. 2015)	E	S	Streaming-Apply	SM	Asynch	DB
2015	GraphTwist (Zhou et al. 2015)	E	S	Slice/Cut pruning	SM	Asynch	DB
2015	GraphQ (Wang et al. 2015)	V	S	Check/Refine	SM	Asynch	DB
2016	Gunrock (Wang et al. 2015)	Da	H	BSP	SM	Synch	MB
2016	GraphIn (Sengupta et al. 2016)	V	D	I-GAS	MP	Synch	MB
2016	LCC-Graph (Cheng et al. 2016)	V	D	LLC-BSP	MP	Synch	MB
2016	DUALSIM (Kim et al. 2016)	V	S	N/A	SM	Asynch	DB
2016	iGiraph (Heidari et al. 2016)	V	D	BSP	MP	Synch	DB
2017	GraphMP (Sun et al. 2017)	V	S	VSW	SM	Asynch	DB
2017	GraphGen (Xirogiannopoulos et al. 2017)	V	S	N/A	SM	Asynch	MB
2017	Mosaic (Maass et al. 2017)	V/E	S	PRA	MP	Synch	DB

— *Architecture*: Distributed (D), single machine (S), or heterogeneous (H).
— *Computational model*: Different names are used by different systems
— *Communication model*: Message passing (MP), shared memory (SM) or dataflow (DF)
— *Coordination*: Synchronous (Synch), asynchronous (Asynch) or both timing approach together
— *Storage*: Disk-based (DB) or memory-based (MB) storage approach
— N/A means that there is no specific name or method mentioned by the article that is describing the system.

Although, many frameworks have been proposed for processing large-scale graphs, there are still several gaps that need to be addressed, as highlighted by this table. Among these observations: (1) Many graph processing systems have been developed based on a vertex-centric programming model because it is the simplest way of partitioning and processing large-scale graphs. Although edge-centric and component-centric systems are more difficult to implement, it has been empirically shown that frameworks such as PowerGraph and GoFFish can scale more

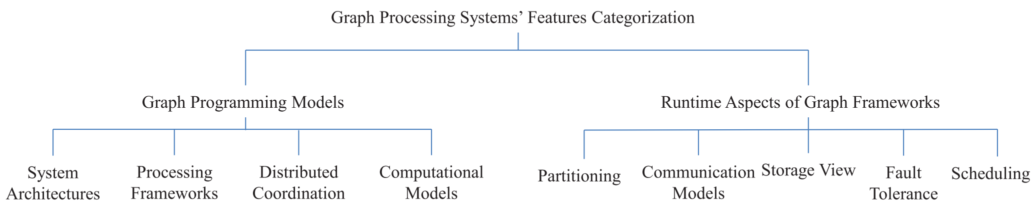


Fig. 15. Proposed graph processing features' categorization in this article.

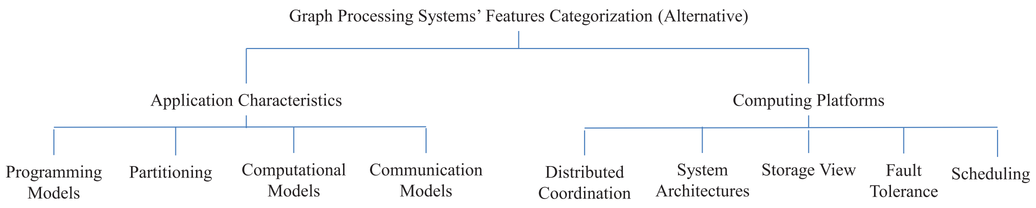


Fig. 16. Graph processing features' categorization according to application characteristics and computing platforms.

efficiently than vertex-centric ones. So, those types of systems need to be investigated more. (2) Disk-based approach is the dominant mechanism that is used by most frameworks. It also includes the frameworks that support out-of-core computation. Disks are cheap but much slower than memory. On the other hand, memory is faster than disk but it is more expensive and memory management makes it more complicated to develop a system based on this approach. (3) Synchronous programming is popular on distributed systems as they avoid race conditions, but often require message passing and have longer runtimes due to the coordination. While asynchronous methods work well on single machine and heterogeneous based frameworks, its effect on distributed frameworks is less studied.

7 DIFFERENT VIEWPOINTS ON CATEGORIZATION OF GRAPH PROCESSING SYSTEMS

Graph processing systems can be categorized based on different intuitions. In this article, we have categorized various features as depicted in Figure 15. We consider both graph programming models and runtime aspects as two broad aspects of graph processing systems, while each can contain multiple sets of features to simplify the understanding of graph processing mechanisms. As part of graph programming models, we explained various system architectures (Section 3.1), current frameworks and how they look at the processing paradigm (Section 3.2), possible distributed coordination that conveys timing (Section 3.3) and computational models (Section 3.4). On the other side, runtime aspects discuss partitioning as the heart of the system (Section 4.1), different communication models (Section 4.2), storage views (Section 4.3), fault tolerance (Section 4.4), and scheduling (Section 4.5). This allows readers to obtain a clear insight about each part of the system, the relationship among various components, and possible improvements.

However, there can be different viewpoints on this. One might separate the features according to application-related and computing-related aspects of the features (Figure 16). In this viewpoint, application characteristics refer to the features that are designed based on the graph system itself so they might have different implementation accordingly. For example, programming models, partitioning, computational models, and communication models are specific characteristics of the system. Computing platform aspects refer to more general features that can vary in different systems but are not specific characteristics of the system. This view is illustrated in Figure 16.

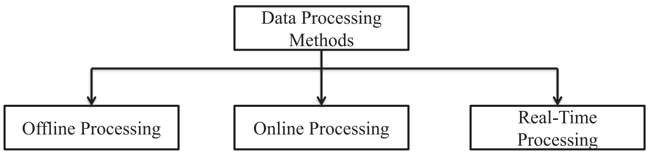


Fig. 17. Data processing approaches.

Another viewpoint (McCune et al. 2015) has classified frameworks based on four major characteristics and called it "four pillars of think like a vertex frameworks": (1) timing, (2) communication, (3) execution model, and (4) partitioning. Overall, regardless of various viewpoints about categorizing graph processing features, there are certain characteristics in every system that are usually considered to be improved in research works.

8 FUTURE DIRECTIONS

Although several works have looked into improving graph processing systems (see Table 3), there are still a number of issues that are open. For example, not many graph processing frameworks use dynamic repartitioning, which performs better than simple static partitioning in many cases. Most of the frameworks use checkpointing for error handling, which can be costly, and other approaches to fault recovery are not well studied. While many researchers have studied classic graph algorithms such as PageRank and shortest paths, it is not clear whether these frameworks can still perform as well for more sophisticated and real-world applications such as machine learning algorithms.

Besides these, there are several other advances to the programming and data models of large graphs and the runtime execution of the graph platforms that need to be examined. These are discussed below.

8.1 Incremental Processing Models

Regardless of the type of framework or algorithms used for processing big graphs, or how large the graph is, data can be processed in three different ways as shown in Figure 17.

According to the problem domain that a framework wants to present solutions for, each data processing approach shown above can be considered in the framework. Offline processing (batch processing) is done where a number of analogous tasks are gathered together to be processed by a computing system all at once instead of individually. In this method, which is used in many graph processing frameworks, the whole graph dataset is loaded into the system, processed for an application, and the results are returned to the user. The original graph is not changed externally, other than through modifications by the running application, and this leads to predictable partitioning and scheduling strategies, which make their design easy.

In online processing, the user can communicate with the system and can make changes to the graph data stored in the system. Thus, the system will be updated automatically and re-process the data with new values periodically or based on user-defined events, which is not necessarily real-time and immediate.

Real-time processing allows the graph to change over time based on incremental updates that it receives to the graph topology or properties. Processing such dynamic graphs is more like an event-driven system where sensors may generate a stream of updates about a vertex or edge that the sensor represents (e.g., road network with traffic cameras or sensors). Real-time processing requires that the computation should be done immediately after the changes happen to the data, and the updated results should be returned with very short delays. Sometimes, the operations may be performed on the delta events themselves before they are actually applied to the graph data. Such requirements are increasingly important in competitive businesses such as social networks, and in IoT domains. There is limited work in the area of real-time processing of large dynamic graphs. For example, Twitter uses a graph-based content recommendation engine called GraphJet (Sharma et al. 2016), which is an in-memory framework that supplements real-time with batch processing by keeping real-time bipartite interplay graph between tweets and users.

The temporal dimension can also come through the notion of time-series graphs where different states of a graph are available, and the application has to operate over both the spatial and temporal dimension. However, this database is collected *a priori* and available offline, and distinct from the changing states of the graph arriving in real-time, e.g., GoFFish operates over time-series graphs for algorithms such as time-dependent shortest path and tracking meme propagation (Simmhan et al. 2015).

8.2 Complex Workflows

A workflow is a dependency graph of different tasks that should be done in a specific order to complete a bigger job. Current graph processing frameworks are based on very simple workflows, typically singleton workflows with one operation executed in a data-parallel manner. They pick a dataset and an algorithm and execute the algorithm on the data. They usually try to solve very simple problems such as finding shortest path or PageRank problem. But, many real world problems are not as easy as this. For example, in a social network, a typical scenario can be like this: an algorithm finds all friends and followers of somebody, then finds the common interests between them using another algorithm, draws a map of his/her communication history, combines all this information with the information from other people in that city to find the whole trends, and so on. Such a complicated series of processes cannot be modeled seamlessly based on existing graph processing systems without manually creating multiple jobs and passing data explicitly between them through the file system. Although Master-Compute model allows a master task to change the phase of computation on the workers, this mechanism can be used only to model simple sequential operations on a single graph and cannot handle more complex operations with more than one graph. So, new frameworks are needed to allow the users to perform more complicated operations.

On the other hand, such complex scenarios also require efficient resource provisioning. That is, proper scheduling is critical to minimize the monetary costs and execution time on one side, and improve resource utilization and performance of the system on the other side. Graph tasks in the workflow may have different processing needs, and may arrive at different intervals, and with different priorities and profitability metrics. Managing these graph workloads offers novel challenges as well. Some research issues on this include the following questions:

- How to schedule complex graph workflows to gain minimum cost and maximum resource utilization?
- What factors influence workflow management in graph processing systems considering graph algorithms characteristics and features of graph datasets?
- How does workflow management in graph processing frameworks–especially for complex scenarios—affect the energy consumption of resources?

8.3 Graph Databases

Relational databases have existed since the 1980s, and have grown mature. While they deal well with structured data tuples stored in tables, their use for storing and querying graph datasets is limited. As discussed in Section 5, graph databases, while not a novel concept, are still in their early stages when considering large property and semantic graphs. It is because relationships in a graph database are much stronger than those hypostatized at runtime in a relational database since they are being treated as high priority entities (Robinson et al. 2015). Relational databases are much slower than graph databases for connected data, hence using graph databases is recommended for highly connected environments and applications such as social networks, IoT, business transactions, and web searching.

Despite the usefulness of graph databases in the aforementioned environments, they are not as mature as relational databases, particularly in terms of tools for data mining purposes on massive graph data on distributed systems. Therefore, future directions for research include the following questions:

—What are the canonical query models for static and dynamic graphs? What is the equivalent of a relational algebra for graph databases?
—What are supporting graph queries in combination with graph kernel algorithms, e.g., find all websites hosted in Australia (property) whose PageRank (algorithm output) is greater than X.
—What are the cost models to be developed for efficient execution of graph queries on distributed environments?
—How to improve the ability to sustain low-latency processing of large numbers of transactional graph queries on distributed and elastic systems like cloud.
—How can analysis be performed across data stored in traditional relational databases and graph databases seamlessly and effectively?
—How do we manage distributed data and indexes in graph databases that have data constantly changing or streaming in?

8.4 Cloud Features and Cost Models

The cloud computing paradigm has modified hardware, software, and data centers' implementation and design. It offers new economical and technological solutions such as utilizing distributed computing, pay-as-you-go pricing models, and resource elasticity. Cloud computing offers computing as a utility in which users can have access to different services according to their needs without heed to where the services are hosted or how they are delivered.

Computing as a service is the infrastructure service most relevant to graph processing. While the scalability offered by VMs has been used for graph processing, these are treated as yet another distributed resource by graph processing frameworks rather than considering their ability to elastically scale or consider their costs. There is limited work on this regard. iGiraph has started to consider cost optimization on clouds (Heidari et al. 2016). It classifies the graph algorithms into convergent and non-convergent algorithms and utilizes a dynamic repartitioning algorithm, which reduces the number of VMs for the graphs that are shrinking during the operation to decline the price. It also performs better on non-convergent applications compared to the famous Giraph.

Dindokar et al. (2016) have proposed an approach to model the computational behavior of non-stationary graph algorithms using a meta-graph model for subgraph-centric programming model. The meta-graph model is able to offer predictions on the subgraphs that will be active in different supersteps, and this is used to schedule subgraphs to VMs in different supersteps, including elastically scaling the VMs in and out (Dindokar and Simmhan 2016). Their strategies show a pricing reduction by half for large graphs like Orkut and for costly graph algorithms like betweenness centrality, with minimal increase in the runtime relative to static over-provisioning of VMs. Elasticity has also been examined in Pundir et al. (2016), where it uses two partitioning mechanisms called contiguous vertex repartitioning and ring-based vertex repartitioning to (1) scale in/out without interfering graph computation, (2) decrease the network overhead after scaling, and (3) keep the load balanced by reducing stragglers across servers.

Cloud providers have different cost models for their VM resources, typically, on-demand VMs that you pay for based on the minutes or hours used, and spot VMs that have dynamic pricing based on demand-supply, and are pre-emptible when the demand out-strips supply. Spot VMs are much cheaper than on-demand VMs and their use can also be explored for large graph applications, while addressing the faults that can occur due to out of bid event when prices spike. While this has been examined for applications like MapReduce (Chohan et al. 2010), there is no work in this regard yet for graph processing.

A service-level agreement (SLA) (Patel et al. 2009) is a contract between a service provider and a service user to define the service features, the time for delivering the service, the steps that should be taken in the case of service crashes, service domain, prices, and so on. Using SLAs, both user and provider can ensure that the service is delivered exactly based on what had been agreed upon and penalties can be applied in case of commitment violation. Quality of a service (QoS) (Ardagna et al. 2014) provides a level of performance, availability, and reliability offered by software, platform, and infrastructure that the service is hosted on them. If we consider graph processing as a service, then the quality of this service should be in an acceptable level from both provider and customer points of views. According to the aforementioned SLA and QoS definitions, and taking graph processing characteristics into consideration, some research directions can be defined as follows:

- Which parameters have the most impact on the performance of a graph processing service and quality of that?
- What factors should be considered for selecting an appropriate graph processing service among other analogous services?
- How SLA-based resource provisioning and scheduling mechanisms for managing graph processing systems and services can be?

8.5 Network Optimizations

The network communication and messaging aspects are less studied in current graph processing frameworks. The factors such as network latency, network bandwidth, network traffic, and topology can affect the runtime performance of the system. The problem also becomes more complicated when it comes to the cloud environments. Most existing distributed graph frameworks have been developed for integrated clusters in which resource management and communication is more predictable. But in a cloud-based framework, the network performance can be variable and VM placement not in the control of users. Hence it becomes essential to consider network factors. Unlike earlier works that considered the role of the network as trivial in graph processing (Ousterhout et al. 2015), particularly for the graphs that can fit into the memory of a single machine, most recent experiments showed that the network plays a major role in the performance of a graph processing system, whether the graph can fit in the memory of a single machine or it is processed on a distributed system (McSherry and Schwarzkopf 2015). For example, allocating larger or denser partitions to the machines with higher bandwidth on one side and reducing the network traffic by decreasing the number of messages transferred between machines on the other hand can enhance the efficiency of the system (Chen et al. 2010).

8.6 Graph Compression

According to Shun and Blelloch (2013), processing large graphs on share memory can be remarkably quicker than processing in a distributed memory environment. Although the amount of data created in the form of a graph is growing every day, the capacity of available memory is also increasing, which enables very large graph datasets to be fit into the memory of a single machine. However, improving the space utilization and execution time of graph algorithms has become crucial. This leads toward compressing graphs to use the memory efficiently.

Graph compression is a technique that has been investigated in the past in frameworks such as WebGraph (Boldi and Vigna 2004) that could store web-based graphs using graph compression algorithms such as referentiation, intervalization, and the like in a limited memory. By the emergence of graph processing frameworks, some systems started proposing similar mechanisms to process large-scale graphs. Ligra + (Shun et al. 2015), for example, is a shared memory graph processing system that is developed based on Ligra to reduce memory usage. Using methods introduced in Blandford et al. (2003), Blandford et al. (2004), and Kourtis et al. (2010), Ligra+ combines encoding (compression) and decoding techniques, utilizing byte codes and nibble codes to represent data. Vertices are being parallelized in encoding where the edge list of each vertex will be compressed by coding the differences between source and target vertices of sequential edges. This framework uses two separate methods for decoding out-edge and in-edge lists. Compression on single machines has been implemented on a number of frameworks (Maass et al. 2017; Sun et al. 2017).

Compression techniques have been used for both single machine and multi-computing frameworks. Liakos et al. (2016) proposed a compression mechanism to optimize memory usage on distributed frameworks. Their solution, which has been developed based on Pregel paradigm includes (1) considering out-edges of each vertex as a row in the graph neighboring matrix for the compression to efficiently represent the space, (2) quick mining the graph without decompression, and (3) considering memory limitation to operate on graph algorithms. GBASE (Kang et al. 2011) is another distributed framework that utilizes graph compression. To store the graph efficiently, GBASE uses block compression by creating multiple regions that contain adjacency matrices.

Apart from various compression techniques that have been implemented on graph processing frameworks, there are some issues that make this topic promising for further research:

— Although compression helps in reducing memory utilization, encoding and decoding data are time-consuming and current solutions have made limited improvement in execution time of the operations.
— Some compression techniques might positively influence particular graph algorithms but not the others. Is there any mechanisms that can be useful for different types of graph algorithms? How about switching between different techniques based on the graph application that is being used?
— How do compression mechanisms affect the bandwidth and other computing resources in a single server or distributed environment?

8.7 Other Improvements

Since scalable graph processing is still in its infancy, there are many open issues to improve the performance and features of each component discussed in Section 2.1, and the overall performance of the system. For example, read and write from/to disk is costly in these systems and usually acts as a bottleneck. In research such as Zheng et al. (2015) and Nilakant et al. (2014), SSD is used as a faster storage device compared to traditional HDDs and cheaper compared to main memory. Further, efficient storage models for graph datasets on disks also need to be explored, e.g., when processing large graphs, the time to load data from disk to memory can outstrip the time to perform the analysis. Compact and compressed graph data representation on disk, loading necessary subsets of the graph on-demand, and support for efficient storage of property graphs are some novel topics to explore. Literature has also examined processing of large graphs on single machines and tries to keep the whole graph and computation results in memory (Section 3.1.2). They rely on memory costs dropping and capacities increasing with new technologies like 3D stacked RAM, when single machines will become viable even for billion-vertex graphs. So there are several aspects of storage and memory management that can be explored.

In addition to these, other parts of graph processing system pipeline can be improved as well. These include the following questions:

— What initial partitioning or pre-processing techniques can improve the performance of the system and speed up the computation process? How can repartitioning improve the efficiency of the system, and can it speed up the computation process?
— How can we better model and predict the behavior of different graph algorithms for graphs with different characteristics, such as power law, small world, planar, and so on? How are these affected by the different programming models? Can we use these to determine the ideal graph processing technique or strategy to be chosen, e.g., synchronous vs. asynchronous algorithms, computation-bound algorithms vs. memory-bound algorithms, denser datasets vs. less dense datasets and so on.
— Are there any computational mechanisms that uses less memory size or can reduce network traffic by reducing the number of messages between machines?
— What fault-tolerance techniques can be used other than check-pointing to improve system reliability and performance?
— What resource provisioning and scheduling algorithms can be used to optimize the processing framework, particularly in a competitive environment such as cloud spot-markets?

9 RELATED WORK

In the literature, there are few surveys on graph processing systems. Each work has investigated the topic from its own perspective and provided limited explanations and comparisons or have not considered many key aspects. For example, McCune et al. (2015) has studied graph processing systems from various aspects including timing, communication, execution model, partitioning, and architecture. Compared to this, we have provided a more comprehensive study and added many new issues.

We have provided many real-world examples of graph-based applications in detail along with an overall overview (see Section 2.1). This has been found vital since readers will understand the future discussions and flow of the article. We also provided a wide categorization and descriptions of algorithms that have been or can be used in graph processing experiments for various purposes. Table 4 provides comprehensive comparison between the coverage of our article and existing work (McCune et al. 2015).

Unlike McCune et al. (2015), which says "TLAV frameworks generally always employ the master-slave architecture," we discuss both distributed and single-server architectures as two major system architectures in existing graph processing frameworks and do not explain them separately. We have also distinguished between programing models and computational models to make the differences clear. Another issue that has not been investigated in other surveys is in-memory execution, which is covered here. Indeed, the taxonomy that we have provided for partitioning and fault tolerance in graph processing systems is more comprehensive compared to other articles and we have discussed the challenges and the ideas for further improvements.

A comprehensive classification of many graph processing systems, since 2010, is presented in this article, which includes various characteristics of the systems. Another novel part of our article is the gap analysis, which is crucial for understanding the shortcomings of current systems and

Table 4. Overview of Existing Graph Processing Frameworks

	McCune et al.	**Our Work**
Examples of graph-based applications	√	√ (*)
Graph processing overall scheme	×	√
Algorithms and challenges	×	√
System architectures	√	√ (*)
Graph processing frameworks	√	√
Distributed coordination (Timing)	√	√
Communication models	√	√
Computational models	√	√
Distinguish between programming model and computational model	×	√
Partitioning	√	√ (*)
In-memory execution	×	√
Fault tolerance	√	√ (*)
Scheduling	×	√
Graph databases	×	√
System classification	×	√
Gap analysis	×	√
Categorizations in graph processing ecosystem	×	√
Incremental processing models (future directions)	×	√
Complex workflows (future directions)	×	√
Graph databases (future directions)	×	√
Cloud features and cost models (future directions)	×	√
Network optimizations (future directions)	×	√
Graph compression	×	√
Possible improvements (future directions)	×	√

* means we offer comprehensive discussion.

provides critical review of the existing systems. Finally, many future directions have been discussed in the article for the first time compared to all other existing surveys. This section enlightens the path for possible opportunities for future works and provides ideas for more research and practical works.

10 SUMMARY AND CONCLUSIONS

Huge quantities of data are being created, analyzed, and used every day in the contemporary world of internet communications and connected devices. "Big data" is the term used to signify the challenges posed by this massive data influx. A growing majority of big data is in the form of "graphs," which are one of the major computational methods of huge data analysis. Social network applications and web searches, IoT, knowledge graphs, and deep learning, financial transactions, and neuroscience are some examples of large-scale graphs that need to be analyzed for various domains. Several works have investigated the creation of effective systems for processing large-scale graphs in recent years.

In this article, we have investigated and categorized existing graph processing frameworks and systems from different perspectives. First, we explained how different parts of a graph processing system, including read and write from/to disk or memory, pre-processing, partitioning, communication, computation, and error handling work together to process large-scale graphs. Second, we presented a taxonomy of different abstractions and approaches that are used in existing graph processing systems within each of these phases. In addition, we described notable frameworks that have used these techniques, and analyzed their advantages and disadvantages to support our discussions. We further summarized the features of graph processing frameworks developed since 2009 in Table 3. It gives a comprehensive overview of current systems and enables making comparisons between them. Finally, future research directions are discussed, which show that scalable graph processing is still at a nascent stage and there are many issues that remain unsolved.

ACKNOWLEDGMENTS

We thank Prof. Sartaj Sahni (editor-in-chief), Prof. Azzedine Boukerche (associate editor), and anonymous reviewers for their constructive comments that helped in improving the article substantially. We also thank Benjamin Rubinstein and Maria Rodriguez for their suggestions on enhancing the article.

REFERENCES

A. Abou-Rjeili and G. Karypis. 2006. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 124–134.

F. N. Afrati, A. Das Sarma, S. Salihoglu, and J. D. Ullman. 2012. Vision paper: Towards an understanding of the limits of map-reduce computation. In *Proceedings of the Coud Futures 2012 Workshop*. Microsoft.

F. Akbari, A. Tajfar, and A. Farhoodi Nejad. 2013. Graph-based friend recommendation in social networks using artificial bee colony. In *Proceedings of the IEEE 11th International Conference on Dependable, Autonomic, and Secure Computing (DASC'13)*. IEEE, 464–468.

A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, and D. Warneke. 2014. The stratosphere platform for big data analytics. *VLDB J. —Int. J. Large Data Bases* 23, 6, 939–964.

Franz Inc. 2015. Home Page. Retrieved August 10, 2015, from http://allegrograph.com.

P. Ammann, D. Wijesekera, and S. Kaushik. 2002. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM, 217–224.

K. Andreev and H. Racke. 2004. Balanced graph partitioning. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*. ACM, 120–124.

R. Angles and C. Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1, 1–39.

Apache Software Foundation. 2007. Retrieved May 28, 2016, from http://activemq.apache.org.

Apache Software Foundation. 2012. Retrieved May 28, 2016, from https://avro.apache.org.

Apache Software Foundation. 2012. Home Page. Retrieved April 9, 2018, from http://giraph.apache.org.

Apache Software Foundation. 2015. Home Page. Retrieved August 10, 2015, from http://hbase.apache.org.

Apache Software Foundation Contributors. 2011. Home Page. Retrieved April 9, 2018, from https://hive.apache.org.

Apache Software Foundation. 2008. Home Page. Retrieved April 9, 2018, from https://pig.apache.org.

Apache Software Foundation. 2008. Home Page. Retrieved May 28, 2016, from https://thrift.apache.org.

Apache Software Foundation. 2011. Home Page. Retrieved July 23, 2015, from https://hadoop.apache.org.

ArangoDB GmbH. 2015. Home Page. Retrieved August 10, 2015, from https://www.arangodb.com.

D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang. 2014. Quality-of-service in cloud computing: Modeling techniques and their applications. *J. Internet Serv. Appl.* 5, 11, 1–17.

D. A. Bader and K. Madduri. 2008. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Systems (IPDPS'08)*. IEEE, 1–12.

D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 2013. *Graph Partitioning and Graph Clustering*. American Mathematical Society, Atlanta, GA.

H. Bagci and P. Karagoz. 2015. Context-aware location recommendation by using a random walk-based approach. *Knowl. Inform. Syst.* 47, 2, 241–260.

M. J. Bannister and D. Eppstein. 2012. Randomized speedup of the Bellman-Ford algorithm. In *Proceedings of the Conference on Analytic Algorithmics and Combinatorics (ANALCO'12)*. Society for Industrial and Applied Mathematics, 41–47.

C. Battaglino, R. Pienta, and R. Vuduc. 2015. GraSP: Distributed streaming graph partitioning. In *Proceedings of the 1st High Performance Graph Mining Workshop (HPGM'15)*. Sydney, Australia.

S. Beamer, K. Asanovic, and D. Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, 1–10.

T. Beseri Sevim, H. Kutucu, and M. Ersen Berberler. 2012. New mathematical model for finding minimum vertex cut set. In *Proceedings of the International Conference on Problems of Cybernetics and Informatics (PCI'12)*. IEEE, 1–2.

M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*. ACM, 93–104.

D. K. Blandford, G. E. Blelloch, and I. A. Kash. 2003. Compact representations of separable graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, Baltimore, Maryland, USA.

D. K. Blandford, G. E. Blelloch, and I. A. Kash. 2004. An experimental analysis of a compact graph representation. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, New Orleans, Louisiana, USA.

P. Boldi and S. Vigna. 2004. The WebGraph framework I: Compression techniques. In *Proceeding of the 13th International Conference on World Wide Web*. ACM, New York, USA.

V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE'11)*. IEEE, 1151–1162.

F. Bourse, M. Lelarge, and M. Vojnovi. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. IEEE, 1456–1465.

Y. Boykov and V. Kolmogorov. 2004. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 9, 1124–1137

T. Britton, M. Deijfen, and A. Martin-Lof. 2006. Generating simple random graphs with prescribed degree distribution. *J. Stat. Phys.* 124, 6, 1377–1397.

Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. 2014. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.* 8, 2, 161–172.

Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.* 3, 1–2, 285–296.

A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. 2016. Recent advances in graph partitioning. In *Algorithm Engineering*. Lecture Notes in Computer Science, Vol. 9220. Springer, 117–158.

R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comput. Syst.* 25, 6, 599–616.

L. Cao. 2011. GoldenOrb. Retrieved July 25, 2015, from https://github.com/jzachr/goldenorb.

U. Catalyurek and C. Aykanat. 1996. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*. Springer-Verlag, 75–86.

M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. 2007. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC'07)*. ACM, 1–14.

C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, 363–375.

K. Chandy and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1, 63–75.

S. Che. 2014. GasCL: A vertex-centric graph model for GPUs. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'14)*. IEEE, 1–6.

G. Chen, Z. Fan, and X. Li. 2005. Modelling the complex Internet topology. In *Complex Dynamics in Communication Networks*. Springer, 213–234.

Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan. 2014. *GraphHP: A Hybrid Platform for Iterative Graph Processing*. Technical Report. arXIV:1706.07221.

R. Chen, X. Weng, B. He, and M. Yang. 2010. Large graph processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, 1123–1126.

R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, 1–13.

Y. Chen, Y.-H. Lee, W. Wong, and D. Guo. 2008. A race condition graph for concurrent program behavior. In *Proceedings of the 3rd International Conference on Intelligent System and Knowledge Engineering (ISKE'08)*. IEEE, Los Alamitos, CA, 662–667.

R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, and E. Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, 85–98.

G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. 2009. Reliable distributed storage. *Computer* 42, 4, 60–67.

N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. 2010. See Spot run: Using spot instances for mapreduce workflows. In *Proceedings of the 2nd USENIX Workshop on Hot Topics inCloud Computing (HotCloud'10)*. USENIX, 1–7.

J. Cohen. 2009. Graph twiddling in a MapReduce world. *Comput. Sci. Eng.* 11, 4, 29–41.

T. E. Commission. 2010. *Social Networks Overview: Current Trends and Research Challenges*. Information Society and Media, Publications Office of the European Union, Luxembourg.

G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. 2012. *Distributed Systems Concepts and Design* (5th ed.). Addison-Wesley, Boston, MA

Committee on the Analysis of Massive Data, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences and Their Applications, Division on Engineering and Physical Sciences and National Research Council. 2013. *Frontiers in Massive Data Analysis*. National Academies Press, Washington, DC.

G. Dai, Y. Chi, Y. Wang, and H. Yang. 2016. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. ACM, 105–110.

J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. ACM, 107–113.

K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE, 1–10.

M. N. Dias de Assuncao, R. Calheiros, S. A. S. Bianchi, M. Netto, and R. Buyya. 2015. Big data computing and clouds: Trends and future directions. *J. Parallel Distrib. Syst.* 79–80, 3–15.

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

R. Dindokar and Y. Simmhan. 2016. Elastic partition placement for non-stationary graph algorithms. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'16)*. IEEE, 90–93.

R. Dindokar, N. Choudhury, and Y. Simmhan. 2016. A meta-graph approach to analyze subgraph-centric distributed programming models. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, 37–47.

N. Doekemeijer and A. Varbanescu. 2014. *A Survey of Parallel Graph Processing Frameworks*. Technical Report. Delft University of Technology, Delft, Netherlands.

D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and O. Lluis Larriba-Pey. 2010. A discussion on the design of graph database benchmarks. In *Proceedings of the 2nd TPC Technology Conference (TPCTC'10)*. Springer-Verlag Berlin, 25–40.

D. Ediger and D. A. Bader. 2013. Investigating graph algorithms in the BSP model on the cray XMT. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*. IEEE, 1638–1645.

I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.* 65, 3, 1302–1326.

J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, and S. H. Bae. 2010. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*. ACM, 810–818.

D. Elena. 2013. *Fault-Tolerant Design*. Springer-Verlag, New York, NY.

E. Elnozahy, L. Alvist, Y.-M. Wang, and D. B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3, 375–408.

H. El-Rewini and M. Abd-El-Barr. 2005. Message passing interface (MPI). In *Advanced Computer Architecture and Parallel Processing*. Wiley, 205–234.

U. Elsner. 2002. *Static and Dynamic Graph Partitioning. A Comparative Study of Existing Algorithms*. Logos Verlag, Berlin, Germany.

N. Engelhardt and H. K. So. 2016. Vertex-centric graph-processing on FPGA. In *Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, 92–92.

S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. 2013. Iterative parallel data processing with stratosphere: An inside look. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 1053–1056.

U. Feige, M. Hajiaghayi, and J. R. Lee. 2005. Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the 37 Annual ACM Symposium on Theory of Computing (STOC'05)*. ACM, 563–572.

U. Feige, M. Hajiaghayi, and J. R. Lee. 2008. Improved approximation algorithms for minimum weight vertex separators. *SIAM J. Comput.* 38, 2, 629–657.

F. Fouss, Pirotte, J.-M. Renders, and M. Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Trans. Knowl. Data Eng.* 19, 3, 355–369.

Z. Fu, M. Personick, and B. Thompson. 2014. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems (GRADES'14)*. ACM, 1–6.

N. Gehani. 1990. Message passing in concurrent C: Synchronous versus asynchronous. *Softw: Pract. Exp.* 20, 6, 571–592.

F. Geier. 2015. *The Differences Between SSD and HDD Technology Regarding Forensic Investigations*. Linnaeus University, Småland, Sweden.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. 1994. *PVM: A Parallel Virtual Machine*. MIT Press, Cambridge, MA.

A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. 2013. On graphs, GPUs, and blind dating: A workload to processor matchmaking quest. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE, 851–862.

A. Gharaibeh, T. Reza, E. Santos-Neto, L. Beltrao Costa, S. Sallinen, and M. Ripeanu. 2013. Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv:1312.3018.

Google. 2008. Retrieved May 28, 2016, from https://github.com/google/protobuf.

J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 17–30.

M. Grabowski, J. Hidders, and J. Sroka. 2013. Representing mapreduce optimisations in the nested relational calculus. In *Proceedings of the 29th British National Conference on Databases*. Springer-Verlag Berlin, 175–188.

he Graph 500 List. 2010. Home Page. Retrieved March 25, 2016, from http://www.graph500.org.

GraphBase Inc. 2015. Home Page. Retrieved August 10, 2015, from http://graphbase.net.

C. Green. 2013. *An Introduction to Graph Databases*. Retrieved July 28, 2015, from http://www.information-age.com/technology/information-management/123457275/an-introduction-to-graph-databases.

D. Gregor and A. Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. In *Proceedings of the Conference on Parallel Object-Oriented Scientific Computing (POOSC'05)*. Glasgow, UK, 1–18.

Marko A. Rodriguez. 2009. Aurelius. Retrieved August 22, 2016, from http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html.

Y. Gu, L. Lu, R. Grossman, and A. Yoo. 2010. Processing massive sized graphs using sector/sphere. In *Proceedings of the 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS'10)*. IEEE, 1–10.

S. Gunelius. 2014, July 12. *The Data Explosion in 2014 Minute by Minute—Infographic*. Retrieved July 25, 2015, from http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic.

Y. Guo, A. Varbanescu, A. Iosup, and D. Epema. 2015. An empirical performance evaluation of GPU-enabled graph-processing systems. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'15)*. IEEE, 423–432.

P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. 2013. WTF: The who to follow service at Twitter. In *Proceedings of the 22nd International Conference on World Wide Web (WWW'13)*. ACM, 505–514.

M. Han and K. Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in Pregel-like graph processing systems. *Proc. LDB Endow.* 8, 9, 950–961.

W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, and E. Chen. 2014. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, ACM, Amsterdam, Netherland, 1–14.

W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, Chicago, IL, 77–85.

A. Harshvardhan Fidel, N. M. Amato, and L. Rauchwerger. 2013. The STAPL parallel graph library. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC'12)*. Tokyo, Japan, Springer, Berlin, 46–60.

S. Heidari, R. N. Calheiros, and R. Buyya. 2016. iGiraph: A cost-efficient framework for processing large-scale graphs on public clouds. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*. IEEE, Cartagena, Colombia, 301–310.

B. Hendrickson and J. W. Berry. 2008. Graph analysis with high-performance computing. *Comput. Sci. Eng.* 10, 2 (2008), 14–19.

D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. 1979. Computing connected components on parallel computers. *Commun. ACM* 22, 8, 461–464.

S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. 2012. Green-marl: A DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, London, UK, 349–362.

I. Hoque and I. Gupta. 2013. LFGraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13).* ACM, Farmington, Pennsylvania, 1–17.

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

P. Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.* 21, 3, 359–411.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (59–72). ACM Lisboa, Portugal, 59–72.

J. Jackson. 2013. *Facebook's Graph Search puts Apache Giraph on the map.* Retrieved July 25, 2015 from PCWorld: http://www.pcworld.com/article/2046680/facebooks-graph-search-puts-apache-giraph-on-the-map.html.

A. K. Jain. 2008. Data clustering: 50 years beyond k-means. *Patt. Recognit. Lett.* (5211); 651–666, Springer, Berlin.

N. Jain, G. Liao, and T. L. Willke. 2013. Graphbuilder: Scalable graph ETL framework. In *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES'13).* ACM, New York, NY, 1–6.

N. Jamadagni and Y. Simmhan. 2016. GoDB: From Batch Processing to Distributed Querying over Property Graphs. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16).* IEEE Cartagena, Colombia, 281–290.

S. Jouili and V. Vansteenberghe. 2013. An empirical comparison of graph databases. In *Proceedings of the International Conference on Social Computing (SocialCom'13)* . IEEE, Alexandria, VA, 708–715.

U. Kang, H. Tong, J. Sun, C. Y. Lin, and C. Faloutsos. 2011. GBASE: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11).* ACM, San Diego, California, 1091–1099.

U. Kang, C. E. Tsourakakis, and C. Faloutsos. 2009. PEGASUS: A peta-scale graph mining system - implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM'09).* IEEE, Miami, FL, 229–238.

Z. Kaoudi and I. Manolescu. 2015. RDF in the Cloud: A survey. *VLDB J.* 24, 1, 67–91.

G. Karypis and V. Kumar. 1995. Multilevel graph partitioning schemes. In *Proceedings of the International Conference on Parallel Processing (ICPP'95).* Raleigh, NC, 1–12.

S. D. Kavila, G. P. Raju, S. C. Satapathy, A. Machiraju, G. Kinnera, and K. Rasly. 2013. A survey on fault management techniques in distributed computing. In *Proceedings of the 2nd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA'13).* Bhubaneswar, Odisha, India. Springer, Berlin, 593–602.

Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoo, D. Williams, and P. Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13).* ACM, Prague, Czech Republic, 169–182.

F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14).* ACM, Vancouver, BC, Canada, 239–252.

H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. Jarrah. 2016. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16).* ACM, San Francisco, California, 1231–1245.

M. Kim and K. Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowl. Eng.* 72, 285–303.

Kobrix Software. 2015. Retrieved August 10, 2015 from http://www.hypergraphdb.org/index.

S. Koo, S. Kwon, S. Kim, and T.-S. Chung. 2015. Dual RAID technique for ensuring high reliability and performance in SSD. In *Proceedings of The IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS'15).* Las Vegas, NV, IEEE, 1–6.

K. Kourtis, G. Goumas, and N. Koziris. 2010. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Transactions on Architecture and Code Optimization* 7, 3 (2010).

D. C. Kozen. 1992. *The Design and Analysis of Algorithms.* Springer, New York, pp. 19–24.

E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. 2011. HipG: Parallel processing of large-scale graphs. *ACM SIGOPS Oper. Syst. Rev.* 45, 2, 3–13.

A. Kyrola and C. Guestrin. 2014. *GraphChi-DB: Simple Design for a Scalable Graph Database System—on Just a PC.* CoRR abs/1403.0701.

A. Kyrola, G. Blelloch, and C. Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12).* USENIX, Hallywood, CA, USA, 31–46.

Avinash Lakshman and Prashant Malik. 2015. Retrieved August 10, 2015, from http://cassandra.apache.org.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for fortran. *ACM Trans. Math. Softw.* 5, 3, 308–323.

H. K. Lau. 2012. Error detection in swarm robotics: A focus on adaptivity to dynamic environments, Ph.D. thesis, University of York.

K. H. Lee, Y.-J. Lee, H. Choi, Y. Chung, and B. Moon. 2011. Parallel data processing with mapreduce: A survey. *ACM SIGMOD Rec.* 40, 4, 11–20.

K. Lee, L. Liu, K. Schwan, C. Pu, Q. Zhang, Y. Zhou, E. Yigitoglu, and P. Yuan. 2015. Scaling iterative graph computations with graphmap, In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).* ACM, Austin, Texas, 1–12.

T. Leimbach, D. Hallinan, D. Bachlechner, A. Weber, M. Jaglo, L. Hennen, and G. Hunt. 2014. *Potential and Impacts of Cloud Computing Services and Social Network Websites.* European Parliamentary Research Service. Brussels: Science and Technology Options Assessment (STOA).

C. Leiserson and T. B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the non-determinism of reducers). In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10).* ACM, Thira, Greece, 303–314.

J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. 2008. *Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.* Pittsburgh, PA. arXiv:0810.1355 (cs.DS).

P. Liakos, K. Papakonstantinopoulou, and A. Delis. 2016. Memory-optimized distributed graph processing through novel compression techniques. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM'16).* ACM, Indianapolis, Indiana, 1–6.

X. Liu, L. Xiao, A. Kreling, and Y. Liu. 2006. Optimizing overlay topology by reducing cut vertices. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'06).* ACM, Newport, Rhode Island, 1–6.

C. Lochert, M. Mauve, H. Füßler, and H. Hartenstein. 2005. Geographic routing in city scenarios. *ACM SIGMOBILE—Mob. Comput. Commun. Rev.* 9, 1, 69–72.

B. Lorica. 2013. *Single server systems can tackle big data.* Retrieved July 25, 2015 from O'Reilly Radar: http://radar.oreilly.com/2013/04/single-server-systems-can-tackle-big-data.html.

B. Lorica. 2014. One year later: Some single server systems that can tackle big data. In *Big Data Now* (29–30). Sebastopol, CA: O'Reilly Media Inc.

Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. 2010. GraphLab: A new framework for parallel machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI'10).* AUAI, Catalina Island, 340–349.

Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. 2012. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8, 716–727.

Y. Lu, J. Cheng, D. Yan, and H. Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.* 8, 3, 281–292.

A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. 2007. Challenges in parallel graph processing. *Parallel Proc. Lett.* 17, 1, 5–20.

X. Ma, D. Zhang, and D. Chiou. 2017. FPGA-accelerated transactional execution of graph workloads. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Array (FPGA'17).* ACM, 227–236.

S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17).* ACM, 527–543.

K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarría- Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09).* IEEE, 1–8.

D. Maier, A. Fiedler, E. Weigelt, and C. Maier. 2015. Retrieved August 10, 2015 from https://sites.google.com/site/jcoredb.

S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, K. Pingali. 2016. DSMR: A parallel algorithm for single-source shortest path problem. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16).* ACM, 1–2.

G. Malewicz, M. H. J. C. Austern, A. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A system for large-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* ACM, 135–146.

A. Marburger and B. Westfechtel. 2010. Graph-based structural analysis for telecommunication systems. In *Graph Transformations and Model-Driven Engineering*, G. Engels, C. Lewerentz, W. Schafer, A. Schurr, and B. Westfechtel (Eds.). Springer, Berlin, Germany, 363–392.

C. Martella, D. Logothetis, A. Loukas, and G. Siganos. 2015. Spinner: Scalable graph partitioning in the cloud. In *Proceedings of the IEEE 33rd International Conference on Data Engineering (ICDE'17).* IEEE, 1083–1094.

R. R. McCune, T. Weninger, and G. Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* 48, 2, 1–39.

F. McSherry and M. Schwarzkopf. 2015. *The Impact of Fast Networks on Graph Analytics*. Retrieved August 28, 2015, from http://www.frankmcsherry.org/pagerank/distributed/performance/2015/07/08/pagerank.html#fn0.

K. Mehlhorn and S. Näher. 1995. The LEDA platform of combinatorial and geometric computing. *Commun. ACM* 38, 1, 96–102.

S. Mittal and J. S. Vetter. 2015. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distribut. Syst.* PP, 99, 1–14.

R. C. Murphy and P. M. Kogge. 2007. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Trans. Comput.* 56, 7, 937–945.

D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 439–455.

K. Najeebullah, K. Khan, W. Nawaz, and Y.-K. Lee. 2014a. BiShard parallel processor: A disk-based processing engine for billion-scale graphs. *Int. J. Multimed. Ubiquitous Eng.* 9, 2, 199–212.

K. Najeebullah, K. Khan, M. Waqas Nawaz, and Y.-K. Lee. 2014b. BPP: Large graph storage for efficient disk based. *arXiv:1401.2327*.

Neo Technology. 2015. Home Page. Retrieved August 10, 2015, from http://neo4j.com.

NetMesh Inc. 2015. Retrieved August 10, 2015 from http://infogrid.org/trac.

D. Nguyen, A. Lenharth, and K. Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 456–471.

D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. 2015. *Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases*. University of Waterloo, Waterloo, Canada.

K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. 2014. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage (SYSTOR'14)*. ACM, 1–12.

B. Nitzberg and V. Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8, 52–60.

E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, and C. Guestrin. 2014. GraphGen: An FPGA Framework for vertex-centric graph computation. In *Proceedings of the IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. IEEE, 25–28.

Objectivity Inc. 2015. Retrieved August 10, 2015 from http://www.objectivity.com/products/infinitegraph.

Joshua O'Madadhain, Danyel Fisher, Tom Nelson, Scott White, and Yan-Biao Boey. 2003. Retrieved June 25, 2016 from http://jung.sourceforge.net.

OrientDB LTD. 2015. OrientDB vs Neo4j. Retrieved August 10, 2015, from http://orientdb.com/orientdb-vs-neo4j.

J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, and R. Stutsman. 2010. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper. Syst. Rev.* 43, 4, 92–105.

K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. 2015. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. USENIX, 293–307.

L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. *The PageRank Citation Ranking: Bringing Order to the Web*. Stanford InfoLab.

P. Patel, A. Ranabahu, and A. Sheth. 2009. *Service Level Agreement in Cloud Computing*. Kno.e.sis Centre, Wright State University, Fairborn, Ohio, USA.

A. Paul. 2013. Graph based M2M optimization in an IoT environment. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems (RACS'13)*. ACM, 45–46.

F. Pellegrini. 2011. Current challenges in parallel graph partitioning. *Comptes Rendus Méc.* 339 (2–3), 90–95.

C. Pettey. 2011. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data. Retrieved July 21, 2015, from http://www.gartner.com/newsroom/id/1731916.

R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Progr.—Dyn. Grids Worldw. Comput.* 13, 4, 277–298.

M. L. Pinedo. 2012. *Scheduling: Theory, Algorithms, and Systems* (4th ed.). Springer-Verlag, New York, NY.

S. J. Plimpton and K. D. Devine. 2011. MapReduce in MPI for large-scale graph algorithms. *Parallel Comput.* 37, 9, 610–632.

R. Power and J. Li. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX, 293–306.

K. Prakasam and M. Chandrasekhar. 2010. JPregel. Retrieved July 24, 2015, from http://kowshik.github.io/JPregel.

M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell. 2016. Supporting on-demand elasticity in distributed. In *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E'16)*. IEEE, 12–21.

F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. 2014. Distributed vertex-cut partitioning. In *Proceedings of the 14th IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, Berlin, 186–200.

M. Redekopp, Y. Simmhan, and V. K. Prasanna. 2013. Optimizations and analysis of BSP graph processing models on public clouds. In *Proceedings of the 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE, 203–214.

I. Robinson, J. Webber, and E. Eifrem. 2015. *Graph Databases.* O'Reilly, Sebastopol, CA.

A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. 2012. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP'12).* ACM, 1–5.

A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15).* ACM, 410–424.

A. Roy, I. Mihailovic, and W. Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13).* ACM, Farmington, Pennsylvania, USA, 472–488.

P. Roy. 2014. A new memetic algorithm with GA crossover technique to solve single source shortest path (SSSP) problem. In *Proceedings of the 2014 Annual IEEE India Conference (INDICON'14).* IEEE, 1–5.

S. Salihoglu and J. Widom. 2013. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM'13).* ACM, 1–12.

S. Salihoglu and J. Widom. 2014. Optimizing graph algorithms on Pregel-like systems. *Proc. VLDB Endow.* 7, 7, 577–588.

S. Salihoglu, J. Shin, V. Khanna, B. Truong, and J. Widom. 2015. Graft: A debugging tool for Apache Giraph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15).* ACM, 1403–1408.

M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. 2013. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proc. VLDB Endow.* 5, 14, 1918–1929.

K. Schloegel, G. Karypis, and V. Kumar. 2001. Graph partitioning for high performance scientific simulations. In *CRPC Parallel Computing Handbook.* Morgan Kaufmann, San Francisco, CA, 491–541.

R. Sedgewick and K. Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional, Upper Saddle River, NJ.

D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan. 2016. GraphIn: An online high performance incremental graph processing framework. In *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing.* Springer-Verlag, New York, 319–333.

B. Shao, H. Wang, and Y. Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13).* ACM, 505–516.

Y. Shao, B. Cui, L. Ma, and J. Yao. 2013. PAGE: A partition aware engine for parallel graph computation. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM'13).* ACM, 1–14.

A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proc. VLDB Endow.* 9, 13, 1281–1292.

J. Shun and G. E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13).* ACM, 135–146.

J. Siek, L.-Q. Lee, and A. Lumsdaine. 2002. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley, Upper Saddle River, NJ.

Y. Simmhan and A. Kumbhare. 2013. *Floe: A Dynamic, Continuous Dataflow Framework for Elastic Clouds.* Technical Report. University of Southern California, Los Angeles, CA.

Y. Simmhan, N. Choudhury, C. Wickramaarachchi, and A. Kumbhare. 2015. Distributed programming over time-series graphs. In *Proceedings of the IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS'15).* IEEE, 809–818.

Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. 2014. GoFFish: A sub-graph centric framework for large-scale graph analytics. In *Proceedings of the Euro-Par 2014 Parallel Processing Conference.* Springer, Cham, 451–462.

Y. Simmhan, C. Wickramaarachchi, A. Kumbhare, M. Frincu, S. Nagarkar, S. Ravi, and V. Prasanna. 2014. Scalable analytics over distributed time-series graphs using GoFFish. arXiv:1406.5975.

I. Stanton and G. Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'12).* ACM, 1222–1230.

M. A. Stelzner. 2015. *2015 Social Media Marketing Industry Report.* Social Media Examiner, Poway, CA.

P. Strandmark and F. Kahl. 2011. Parallel and distributed graph cuts by dual decomposition. *Comput. Vis. Image Underst.* 115, 12, 1721–1732.

P. Stutz, A. Bernstein, and W. Cohen. 2010. Signal/Collect: Graph algorithms for the (Semantic) web. In *Proceedings of the 9th International Semantic Web Conference (ISWC'10).* Springer-Verlag Berlin, 764–780.

Z. Sun, H. Wang, H. Wang, B. Shao, and L. Jianzhong. 2012. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.* 5, 9, 788–799.

P. Sun, Y. Wen, T. Nguyen Binh Doung, and X. Xiao. 2017. GraphMP: An efficient semi-external-memory big system on a single machine. arXiv:1707.02557.

S. Suri and S. Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11).* ACM, 607–614.

SYSTAP, LLC. 2015. Home Page. Retrieved August 10, 2015, from http://mapgraph.io.

A. S. Szalay. 2011. Extreme data-intensive scientific computing. *Comput. Sci. Eng.* 13, 6, 34–41.

S. Tasci and M. Demirbas. 2013. Giraphx: Parallel yet serializable large-scale graph processing. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13).* Springer-Verlag Berlin, 458–469.

N. Thien Bao and T. Suzumura. 2013. Towards highly scalable Pregel-based graph processing platform with x10. In *Proceedings of the 22nd International Conference on World Wide Web (WWW'13 Companion).* ACM, 501–508.

J. Tian, J. Hahner, C. Becker, I. Stepanov, and K. Rothermel. 2002. Graph-based mobility model for mobile ad hoc network simulation. In *Proceedings of the 35th Annual Simulation Symposium.* IEEE, 337–345.

Y. Tian, A. Balmin, S. Andreas Corsten, S. Tatikond, and J. McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph." *Proc. VLDB Endow.* 7, 3, 193–204.

M. Treaster. 2005. A survey of fault-tolerance and fault-recovery techniques in parallel systems. arXiv:cs/0501002.

C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. 2014. FENNEL: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM'14).* ACM, 333–342.

N. Kallen, R. Pointer, J. Kalucki and Ed Ceaser. 2012. Twitter/FlockDB. Retrieved July 28, 2015, from https://github.com/twitter/flockdb#readme.

L. G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8, 103–111.

L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. 2013. xDGP: A dynamic graph processing system with adaptive partitioning. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13).* ACM, 1–2.

S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. 2013. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13).* ACM, 197–210.

C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. 2010. A comparison of a graph database and a relational database. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE'10).* ACM, 1–6.

G. Wang, W. Xie, A. Demers, and J. Gehrke. 2013. Asynchronous largescale graph processing made easy. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13).* 1–12.

K. Wang and G. Xu. 2015. GraphQ: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single PC. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15).* USENIX, 387–401.

P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. 2014. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14).* IEEE, 562–573.

Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. Owens. 2015. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15).* ACM, 1–12.

X. Wu, X. Ying, K. Liu, and L. Chen. 2010. A survey of algorithms for privacy-preservation of graphs and social networks. In *Managing and Mining Graph Data, Advances in Database Systems*, vol. 40. Springer, Boston, MA, USA.

C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. 2015. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15).* ACM, 194–204.

R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. 2013. GraphX: A resilient distributed graph system on spark. In *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES'13).* ACM, 1–6.

K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. 2017. GraphGen: Adaptive graph processing using relational databases. In *Proceedings of the 5th International Workshop on Graph Data-Management Experiences and Systems.* ACM, 1–7.

N. Xu, L. Chen, and B. Cui. 2014. LogGP: A log-based dynamic graph partitioning method. *Proc. VLDB Endow.* 7, 14, 1917–1928.

J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. 2014. Seraph: An efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14).* ACM, 227–238.

Y. Yamato. 2015. Use case study of HDD-SSD hybrid storage, distributed storage and HDD storage on OpenStack. In *Proceedings of the 19th International Database Engineering and Applications Symposium (IDEAS'15).* ACM, 228–229.

D. Yan, J. Cheng, Y. Lu, and W. Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.* 7, 14, 1981–1992.

D. Yan, J. Cheng, M. Ozsu, F. Yang, Y. Lu, J. C. Lui, and W. Ng. 2016. Quegel: A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.* 9, 7, 564–575.

J. Yan, G. Tan, and N. Sun. 2013. GRE: A graph runtime engine for large-scale distributed graph-parallel applications. arXiv:1310.5603.

S. Yang, X. Yan, B. Zong, and A. Khan. 2012. Towards effective partition management for large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. ACM, 517–528.

E. Yoneki, K. Nilakant, V. Dalibard, and A. Roy. 2014. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage (SYSTOR'14)*. ACM, 1–12.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Kumar Gunda, and J. Currey. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX, 1–14.

P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. 2014. Fast iterative graph computation: A path centric approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. IEEE, 401–412.

Matei Zaharia. 2012. Home Page. Retrieved March 23, 2016, from http://spark.apache.org.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, and I. Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX, 1–14.

Z.-J. Zha, T. Mei, J. Wang, Z. Wang, and X.-S. Hua. 2009. Graph-based semi-supervised learning with multiple labels. *J. Vis. Commun. Image Represent.* 20, 2, 97–103.

T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang. 2015. Efficient graph computation on hybrid CPU and GPU systems. *J. Supercomput.* 71, 4, 1563–1586.

Y. Zhang, Q. Gao, L. Gao, and C. Wang. 2012. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Data (ScienceCloud'12)*. ACM, 13–22.

Y. Zhang, Q. Gao, L. Gao, and C. Wang. 2012. PrIter: A distributed framework for prioritized iterative computations. *IEEE Trans. Parallel Distrib. Syst.* 24, 9, 1884–1893.

Y. Zhang, Q. Gao, L. Gao, and C. Wang. 2014. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.* 25, 8, 2091–2100.

D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. 2015. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, 45–58.

J. Zhong and B. He. 2013a. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6, 1543–1552.

J. Zhong and B. He. 2013b. Towards GPU-accelerated large-scale graph processing in the cloud. In *Proceedings of the 5th International Conference on Cloud Computing Technology and Science (CloudCom'13)*. IEEE, 9–16.

J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. 2012. SCOPE: Parallel databases meet MapReduce. *VLDB J. —Int. J. Very Large Data Bases* 21, 5, 611–636

X. Zhu, W. Han, and W. Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*. USENIX, 375–386.

X. Zhu, W. Chen, W. Zheng, and X. Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX, 301–316.

Received February 2017; revised January 2018; accepted March 2018