Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Review:

Large-scale graph processing systems: a survey*

Ning LIU, Dong-sheng LI‡, Yi-ming ZHANG, Xiong-lve LI

Science and Technology on Parallel and Distributed Processing Laboratory,
National University of Defense Technology, Changsha 410000, China

E-mail: liuning17a@nudt.edu.cn; dsli@nudt.edu.cn; zhangyiming@nudt.edu.cn; lixionglve17@nudt.edu.cn

Received Mar. 5, 2019; Revision accepted Sept. 16, 2019; Crosschecked Nov. 12, 2019

Abstract: Graph is a significant data structure that describes the relationship between entries. Many application domains in the real world are heavily dependent on graph data. However, graph applications are vastly different from traditional applications. It is inefficient to use general-purpose platforms for graph applications, thus contributing to the research of specific graph processing platforms. In this survey, we systematically categorize the graph workloads and applications, and provide a detailed review of existing graph processing platforms by dividing them into general-purpose and specialized systems. We thoroughly analyze the implementation technologies including programming models, partitioning strategies, communication models, execution models, and fault tolerance strategies. Finally, we analyze recent advances and present four open problems for future research.

Key words: Graph workloads; Graph applications; Graph processing systems
https://doi.org/10.1631/FITEE.1900127 **CLC number:** TP391.41

1 Introduction

With the rapidly dramatic expansion of volume, velocity, and variety, big data provide abundant information. There are multiple representations of the same data. Specifically, graph structure is an abstraction of group relationship that can describe relationships between objects. As a result, academic and industrial communities are paying much attention to analysis and processing of graph data, aiming at making the most efficient use of them. Graph data play a tremendous role in a variety of scenarios, such as social networks (Google page graph, Facebook social networks, and Amazon user behavior graph), bioinformatics (graph neural network (GNN) and protein interaction), and knowledge graphs. Taking social networks as an example, the graph data structure consists of a finite (possibly mutable) set of vertices which represent the individual together with a set of ordered (unordered) edges which represent connections between individuals (subordination or followers). Concealed relationship can be explored through data mining.

As the society develops, the size of graphs rapidly grows. Statistics reveal that as of March 2019, there are now more than four billion people around the world using the Internet and more than three billion using social media. As of the first quarter of 2019, Facebook had 2.38 billion monthly active users. YouTube had 1.5 billion users worldwide, and global users are projected to grow to 1.86 billion in 2021 (https://www.statista.com/statistics). Different social media platforms show varied growth. The high volume makes it a matter of urgency to guarantee efficiency. This stimulates the academic and industrial communities to tackle the challenge by theoretical invention and practical exploration.

‡ Corresponding author
* Project supported by the National Key Program of China (No. 2018YFB2101100), the National Natural Science Foundation of China (Nos. 61932001 and 61872376), and the Major State Research Development Program of China (No. 2016YFB0201305)
ORCID: Ning LIU, https://orcid.org/0000-0002-8966-7869; Dong-sheng LI, https://orcid.org/0000-0001-9743-2034
© Zhejiang University and Springer-Verlag GmbH Germany, part of Springer Nature 2020

Due to the complexity of graph structure, the differences of processing and analysis between graph and traditional applications are vast, which can be measured to be at least four aspects.

1. Graph applications have lower overall rate of computation and memory access

Traditional applications are computation-intensive and devote most of their execution time to computational requirements. The computational complexity is high enough to cover up the access delay. Graph applications with much less computation on each vertex and heavy interaction between vertices devote most of their processing time to input/output (I/O) and manipulation of data, which highlights the memory access latency.

2. Graph applications lack data locality

In traditional applications, related data can be kept close together, which signifies superior spatial locality. So, we can do a sequential read of a bunch of data and save time. Better spatial locality of traditional data can accelerate memory access by arranging data to take advantage of central processing unit (CPU) caching. However, the problems with graph data structure are worse adaptations for random-access queries, although some of the implementations try to overcome them by storing the graph through an index.

3. Graph applications have complex data dependency

Traditional data seldom call multiple use of the same location(s) in storage by different tasks, and thus parallelism is uninhibited. Yet tasks in graph applications are closely related to data of vertices and adjacent edges. We cannot accurately predict the computing structure, which limits parallelism.

4. Graph applications follow unstructured distribution

Graph data structure is usually unstructured and irregular. The irregular structure makes it difficult to extract parallelism by partitioning the input data. Scalability can be limited due to poorly partitioned data. What is more, many natural graphs in the real world have power-law features. The degree of power-law graphs in the real world is quite imbalanced, further contributing to load imbalance and communication overhead.

To tackle these challenges, huge endeavor has been made, resulting in a wealth of papers on and solutions to graph processing. In this survey, we try to provide an overview of the state-of-the-art graph processing systems.

There exist several comprehensive reviews on graph processing systems. Sakr et al. (2015) summarized some early graph processing systems and categorized them into big structured query language (SQL) systems, big graph processing systems, and large stream processing systems. Batarfi et al. (2015) reviewed Hadoop-based systems, Pregel-based systems, GraphLab family systems, and other systems. Experimental evaluation and analysis of the performance characteristics for five popular systems were also made. Doekemeijer and Varbanescu (2014) summarized the implementation of systems in this domain. Our paper differs from these works in that we systematically and comprehensively review different graph processing systems rather than focusing on one specific branch. Another difference is that we systematically categorize the workloads and applications and analyze the frameworks and implementation techniques.

This survey presents an extensive survey of graph processing systems with the following contributions:

1. We provide a detailed review over the state-of-the-art scalable graph processing systems and divide them into two categories: general-purpose and specialized systems.

2. We systematically categorize the workloads and applications and divide the algorithms into traversal- and dense-computation-based algorithms.

3. We analyze and discuss the implementation techniques including programming frameworks, graph partitioning models, communication methods, fault tolerance strategies, and load balancing methods.

4. We propose several open problems for future research. Graph processing systems suffer from a huge volume of data. There are still no effective methods for dealing with attributed and dynamic graphs. We provide analysis of each problem and propose future research directions.

2 Workloads and applications

Big graph processing has been explored in a wide range of problem domains across social networks, insurance, advertising, transportation, and others. In this section, we simply divide the application

workloads into two types of scenarios: (1) traversal-based scenarios (Cheung, 1983) in which most of the processing time is spent in I/O operations, such as the breadth-first search (BFS) (Awerbuch and Gallager, 1985; Brandes, 2001; Yoo et al., 2005; Ajwani et al., 2006, 2007; Leiserson and Schardl, 2010; Buluç and Madduri, 2011), single-source shortest paths (SSSP) (Arge et al., 2000; Maheshwari and Zeh, 2001; Bader and Madduri, 2006; Nanongkai, 2014), and minimum spanning tree (MST) (Bader and Cong, 2006; Lotker et al., 2006); (2) Dense-computation-based scenarios (Kamvar et al., 2003) in which heavy computation is devoted to vertices or edges, such as PageRank (PR) (Mihalcea, 2004; Desikan et al., 2005; Gonzalez et al., 2012; Sarma et al., 2013), Connected Component (CC) (Hirschberg et al., 1979; Nuutila and Soisalon-Soininen, 1994), and Triangle Counting (TC) (Becchetti et al., 2008; Kolountzakis et al., 2012; Tangwongsan et al., 2013; Kutzkov and Pagh, 2014). Due to the irregular data structure and memory access locality, graph applications are much more difficult to parallelize. In addition, different kinds of graph workloads require different strategies to balance the workload and fully use CPU parallelism. For these reasons, understanding the category of graph application workloads is important for researchers to propose effective solutions to address irregular workloads of various graph-related applications. The categories of the workloads and applications are given in Tables 1 and 2.

2.1 Workloads

2.1.1 Traversal-based workloads

Traversal-based algorithms are basic graph algorithms that visit nodes of a graph in certain order to check or update information and start from a subset of the input graph in each iteration to explore the adjacent vertices and edges until all reachable vertices are visited. Then the specific computations are finished. As a result, there exists a large amount of random access to memory in traversal-based workloads, which can lead to a large amount of communication overhead in distributed systems. So, we should take random access to memory and communication overhead into consideration when designing graph processing systems.

Table 1 Categories of graph workloads

Type	Algorithm(s)	Algorithm description	Service	Application
Traversal-based	BFS and DFS	Search	Link analysis	Transportation
	Dijkstra, Bellman Ford, SPFA, and APSP	Shortest path	Distance/path	Transportation
	Prim, MST, and Kruskal	Minimum spanning tree	Smooth traffic	Traffic project
Dense-computation-based	PageRank	Rank web page	Link analysis	Social network
	Connected component	Maximal connected subgraphs	Community detection	Social network
	Triangle counting	Counting triangles	Link analysis	Social network

BFS: breadth-first search; DFS: depth-first search; APSP: all pairs shortest path; MST: minimum spanning tree

Table 2 Categories of graph applications

Type	Research branch(es)	Algorithms or typical models
Graph database	Graph pattern matching	Neo4j, Oracle PGX, SAP HANA Graph, and Redis
Graph mining	Frequent subgraph mining	AGM, SPIN, gSpan, and CloseGraph
	Community detecting	FFSM, FSG, GREW, and CPM
	Influence maximization	CPMw, IS, FCM, EAGLE, IC, and LT
Graph machine learning	GNN	ChebNet and Neural FPs
	GCN	PATCHY-SAN and DCNN
	GAE	Neural FPs
	Graph autoencoder	DGCN and ARGA

GNN: graph neural network; GCN: graph convolutional network; GAE: generalized advantage estimator; CPM: clique percolation method; CPMw: CPM with weights; IS: iterative scan; EAGLE: agglomerative hierarchical clustering based on maximal clique; IC: independent cascade; LT: linear threshold; ARGA: adversarially regularized graph autoencoder

BFS, SSSP, Floyed-Warshall APSP (All Pairs Shortest Path) (Harish et al., 2009; Chan, 2010; Matsumoto et al., 2011; Nanongkai, 2014), and MST are all traversal-based graph algorithms and have been applied in many real-world applications, such as transportation and retail services.

2.1.2 Dense-computation-based workloads

Although traversal-based workloads exist in a larger percentage of graph applications, dense-computation-based workloads are also significant. In contrast to traversal-based workloads, for dense-computation-based workloads various computations are executed on several or all vertices with adjacent edges in each iteration. Dense-computation-based workloads have better data locality, but the challenge can be caused by the imbalance of vertices and edges, especially for power-law graphs. Besides, we should focus on the communication overhead of dense-computation-based workloads while designing graph processing systems.

PR, CC, TC, and many more algorithms are dense-computation-based graph workloads which are widely used in social networks.

2.2 Applications

Graphs are known to have complicated structures which contain rich value. Graph databases, graph mining, and graph machine learning are three main graph application directions to analyze graph data for inner relationships and hidden information.

2.2.1 Graph database

Adopted by many large companies such as Facebook, Twitter, and Google, graph databases have been attracting increasing attention for simplicity and flexibility. Graph database is a database that uses graph structures for semantic queries with vertices, edges, and properties to present and store data. Compared with the traditional relational database, it performs better. In the traditional relational database, large amounts of tables are required to store the complex graph structure, while the graph database can easily represent inter relationships between vertices. More importantly, adding or deleting data in the graph database means adding or deleting vertices or edges rather than tables, which makes it more flexible and reusable. Many commercial graph databases, such as Neo4j, Oracle PGX, SAP, HANA Graph, Redis Graph, Cyper for Apache Spark, and TigerGraph, have made tremendous progress and have been widely deployed in many projects. Graph databases can be accessed by query languages like Cyper, Gremlin (Rodriguez, 2015), SARQL, and GSQL.

Graph pattern matching is a powerful mechanism for search on the graph database. It has applications in areas such as computer vision, biology, electronics, computer aided design, social networks, and intelligence analysis. The basic graph pattern matching problem is to find a unique subgraph from a large graph to match a specific pattern. As a result, graph pattern matching is essentially based on subgraph isomorphism, which is confirmed to be a non-deterministic polynomial (NP) complete problem (Washio and Motoda, 2003). Graph pattern matching algorithms can be divided into exact and approximate match. Exact match aims to find all subgraphs in the data graph that match the query graph. However, enumerating all subgraph isomorphic embeddings is NP-hard. Another disadvantage is that the graph data and results can be huge (MB-scale graph can produce PN-scale results). The first exact pattern matching algorithm was proposed by Ullmann (1976). VF2, QuickSI (Shang et al., 2008), TurboISO (Han et al., 2013a), and BoostISO (Ren and Wang, 2015) are typical exact algorithms. Approximate match is harder than exact match. The solution set could be huge. The spanning-tree-based matching algorithm (Zhu G et al., 2012) and Lawler-procedure-based top-k tree searching algorithm (Chang et al., 2015) both follow the idea to reduce the complexity of exact all-matching from graphs to trees.

2.2.2 Graph mining

Graph mining is one of the hot topics in graph database and artificial intelligence. Efficient graph mining algorithms contribute to inferring useful knowledge from structural graph data. Frequent subgraph mining (FSM), community detection, and influence maximization are three active research fields.

FSM is considered to be the essence of graph mining. It aims at resolving the structural pattern mining issues and finding all the subgraph patterns that frequently occur over the large graph. FSM is

widely used in graph clustering, classification, and building indices.

Many FSM algorithms have been proposed, such as SUBDUE (Holder et al., 1994), AGM (Inokuchi et al., 2000), gSpan (Yan XF and Han, 2002), FFSM (Huan et al., 2003), CloseGraph (Yan XF and Han, 2003), GREW (Kuramochi and Karypis, 2003), FSG (Kuramochi and Karypis, 2004), and SPIN (Huan et al., 2004). FSM algorithms can be classified as Apriori- and pattern-growth-based approaches. Apriori-based algorithms, which are suitable for small graphs, use the idea of "generation-and-test." The set of frequent k subgraphs is used to generate the set of frequent $(k+1)$ subgraphs. Pattern-growth-based algorithms (Borgelt and Berthold, 2002; Yan XF and Han, 2002, 2003; Huan et al., 2003) can tackle the overhead resulting from the graph scale. Pattern-growth-based algorithms, such as gSpan, CloseGraph, FFSM, MOFA, Gaston, and SUBDUE, explore the depth-first search (DFS) algorithm in generating candidates.

Community detection tries to reveal the inner relationships between the graph structures and functions. It aims at partitioning the vertices into different clusters with fewer edges between clusters. It can be classified as overlapping and non-overlapping. Various algorithms try to detect communities from different perspectives. The clique percolation method (CPM) and CPM with weights (CPMw) (Farkas et al., 2007) detect communities by searching for adjacent cliques. Iterative scan (IS) (Baumes et al., 2005; Kelley, 2009), FLM (Lancichinetti et al., 2009), agglomerative hierarchical clustering based on maximal clique (EAGLE) (Shen HW et al., 2008), and greedy clique expansion (GCE) (Lee et al., 2010) are implemented on top of local expansion. Zhang S et al. (2007) and Psorakis et al. (2011) proposed algorithms based on fuzzy clustering.

The influence maximization problem is a common problem in social networks. It has two branches. In one case, k vertices to be found in a certain graph as seeds can influence all the vertices in the network. In the other case, the influence is given. We are required to find the minimum set of vertices to satisfy the influence. Two typical models used in the influence maximization problem are the independent cascade (IC) diffusion model and the linear threshold (LT) diffusion model. The IC models, such as DiusionRank (Ma et al., 2008) and DynaDiffuse (Miao, 2015), are probability models. There is a possibility that an active vertex can activate one of its neighbors. The LT model is an accumulation model, where each vertex in the graph has a threshold. A vertex will be activated when the sum of in-edge weights from active vertices is higher than the threshold.

2.2.3 Graph machine learning

In the past few years, machine learning methods, such as convolutional neural networks (CNN) and recursive neural network (RNN), have been widely used for regular Euclidean data like images (two-dimensional grid) and text (one-dimensional sequence) in natural language processing, object detection, and many other applications. However, as a unique non-Euclidean structure, graph data are analyzed in the machine learning domain for node classification, link prediction, and clustering. Graph machine learning can be divided mainly into three categories: graph neural network (GNN), graph convolutional networks (GCN), and graph autoencoders.

Research on GNN is primitively motivated by CNNs. Scarselli et al. (2009) first presented the idea of GNN. In GNN, each vertex has the concept of "state," which is represented by a state embedding vector. The vertex state embedding vector is closely associated with the feature and the state of edges and vertices in its neighborhood. The target of GNN is to obtain a state embedding vector of each vertex in the graph. Due to the fact that vertices are dependent on each other in the neighborhood, the computation will converge to a stable condition after multiple iterations.

GCN tries to use graph convolution. Standard convolution used in CNNs on images and texts is not suitable for graphs. It is clear that different vertices have different numbers of neighbors, which makes it impossible to use convolution kernels of the same size when processing different vertices. To tackle this problem, GCN uses spectral and spatial approaches. Spectral approaches use the graph Laplacian matrix (Belkin and Niyogi, 2001). Spatial approaches directly use the graph topology to collect information through the neighborhood of each vertex. Bruna et al. (2014) first introduced the spectral method using the graph Laplacian matrix. The efficiency

problem has been solved through the optimization of the convolution kernel (Defferrard et al., 2016; Kipf and Welling, 2016a). As for the spatial method, the difficulty lies in the determination of the "receptive field" (Niepert et al., 2016) of each vertex and the treatment with different numbers of neighbors. Neural FPs (Duvenaud et al., 2015), PATCHY-SAN (Niepert et al., 2016), DCNN (Atwood and Towsley, 2016), and DGCN are spatial methods. Inspired by the attention mechanism (Vaswani et al., 2017), graph attention networks (GATs) (Veličković et al., 2017) can benefit a lot for aggregating information, integrating outputs of multiple models, and generating material-oriented random walks.

Graph autoencoder, e.g., the sparse autoencoder (SAE) (Wang DX et al., 2016), variational graph autoencoder (VGAE) (Kipf and Welling, 2016b), and adversarially regularized graph autoencoder and adversarially regularized variational graph autoencoder (ARGA/ARVGA) (Pan et al., 2018), is a network embedding method which aims at learning low-dimensional vertex representations.

3 Existing solutions

In this section, we divide the existing graph processing solutions into general-purpose and specialized systems. The specialized systems are composed of shared-memory, out-of-core, and distributed systems. We will introduce the typical systems for each category and analyze their advantages and disadvantages.

3.1 General-purpose systems

Before specialized graph processing systems appeared, driven by the increasing demand of graph applications, people usually develop systems based on mature frameworks like MapReduce (Dean and Ghemawat, 2008), which are usually general processing platforms used for data analysis on a mass scale. Hadoop is designed as an open-source implementation of MapReduce.

MapReduce is a distributed parallel computing framework suitable for the cloud computing platform, and its processing can be regarded as based on the data flow model. MapReduce applications must have the characteristic of "map" and "reduce," meaning that the task or job can be divided into smaller pieces to be processed in parallel.

Each MapReduce job can be regarded as an iteration in graph algorithms like PR. The final output of MapReduce is the intermediate result of an iteration. As a result, a graph algorithm like PR can be implemented by serially executing several MapReduce jobs. The result of each job is used as the input of the next job. However, most graph algorithms are iterative, no matter traversal- or dense-computation-based graph algorithms. Serial execution would cause inefficient graph analysis. The MapReduce system is designed to be a simple paradigm for writing code that needs to be massively parallel. However, the MapReduce system is unfriendly to graph computing with high coupling and calculation that require multiple iterations. What is more, the shuffle stage in each iteration requires high communication overhead, which is also a challenge for efficient graph processing. Specifically, when the data scale sharply grows, the computation and communication burden rises quickly.

Hadoop is the open-source implementation of MapReduce. After Hadoop was proposed in 2008, PEGASUS (Kang et al., 2009) was implemented on top of the Hadoop platform to process graph data which reach several giga-, tera-, or peta-bytes in 2009. It uses generalized iterated matrix-vector multiplication (GIM-V) to perform graph mining operations (PR, spectral clustering, diameter estimation, and connected components) on large-scale graph data. Block multiplication reduces the high communication overhead during the shuffle stage.

In 2010, there are a lot of general-purpose graph processing systems including HaLoop (Bu et al., 2010), Twister (Ekanayake et al., 2010), Surfer (Chen R et al., 2010), HAMA (Seo et al., 2010), Giraph (Avery, 2011), and many others.

3.2 Specialized systems

MapReduce is designed for single-pass applications to do parallel calculation rather than graph applications with high data dependency and large numbers of iterations. So, there are several disadvantages about systems modified from the MapReduce platform:

1. Warm-up overhead

Job scheduling is an important process in Hadoop MapReduce. The workers are assigned work by the master, thus causing high overhead. Besides, the pre-processing of tasks and jobs takes a lot of

time. A Hadoop task with empty load needs 25–30 min to warm up, let alone tasks with high-frequency iterative jobs.

2. Static data operating overhead

Static data refers to the data that do not change during iterations, such as graph topology in the PR algorithm. Static data require repeated reading/writing operations on the local machine through the Hadoop distributed file system (HDFS) and are transmitted at the shuffle stage. Such operations lead to enormous data access overhead and communication overhead.

3. Data storage inefficiency

Data storage in the Hadoop platform relies on HDFS. HDFS is not well suited for graph applications that require low-latency (tens of milliseconds) access because HDFS is designed for high throughput at the cost of some latency. HDFS has one single master and all requests to files go through it. Large numbers of data requests definitely cause delay.

4. Shuffle overhead

The shuffle operation in MapReduce sorts data by default. The sorting operation costs much time.

5. Poor programming interface

The MapReduce framework offers map and reduces interface, which simplifies the programming of general algorithms. Nevertheless, it is difficult to implement convergence detection and other graph algorithm expressions.

The demands of machine learning and data mining are sharply growing at the same time. All the shortcomings listed above make it extremely urgent to design specialized graph processing systems. We divide the specialized systems into three categories and will introduce them in this subsection.

3.2.1 Single-machine shared-memory systems

Single-node shared-memory systems with multicores support more than a terabyte of memory, which can fit graphs with tens or even hundreds of billions of edges (Shun and Blelloch, 2013). Compared to distributed systems, the advantages of single-node shared-memory systems are as follows:

1. Data are spread over several physical servers. Single-node systems reduce unnecessary overhead caused by a large amount of extra communication across servers.

2. Storing data in memory significantly saves the time of disk I/O.

3. The demand for programmers to design graph algorithms in a distributed system is higher than that in a stand-alone system.

Since 2010, many single-node shared-memory systems have been proposed. The state-of-the-art systems include Ligra (Shun and Blelloch, 2013), Galois (Nguyen et al., 2013), GRACE (Xie WL et al., 2013), Polymer (Zhang KY et al., 2015), GraphMat (Sundaram et al., 2015), NXgraph (Chi et al., 2016), and CGraph (Zhang Y et al., 2018).

Ligra provides routines for mapping over edges and vertices, and is suitable for traversal-based graph algorithms. Galois implements more sophisticated algorithms by domain-specific languages (DSLs) and finds that the performance can be improved by orders of magnitude, especially on road networks and similar graphs. GRACE provides a synchronous iterative graph programming model and processes graphs based on message passing. GraphMat is the first lightweight graph processing framework using a vertex-centric model and achieves good multicore scalability. Polymer improves the performance of graph algorithms on a non-uniform memory access architecture. NXgraph proposes a destination-sorted sub-shard (DSSS) structure which divides vertices and edges in a graph into intervals and sub-shards, respectively. Edges in each shard are sorted according to their destination vertices to ensure graph data access locality and enable fine-grained scheduling. CGraph uses the correlation-aware execution model, together with a core-subgraph-based scheduling algorithm, and achieves improvement on concurrent iterative graph processing (CGP) jobs.

Obviously, the single-machine shared-memory systems can achieve scalability only by adding CPU or expanding memory. However, scalability is limited by the hardware. Shared-memory systems involve multiple threads calling for the same data, and a large number of locks are needed to ensure data consistency. Besides, frequency locking and unlocking operations are time-consuming and expensive, which makes the processing inefficient. Meanwhile, careless handling of a large number of locks may cause deadlock, resulting in system crash.

3.2.2 Single-machine out-of-core systems

With the sharp increase of graph data, the storage hierarchy of single-machine memory-shared systems is extended from random access memory

(RAM) to external memory, such as solid state drive (SSD), Flash, serial attached SCSI (SAS), and hard disk drive (HDD) to tackle the challenge of scalability. However, limited by the computing capacity and data exchanging bandwidth of external memory, it is hard to process large graphs under acceptable conditions. The biggest challenge of out-of-core systems is random disk access. State-of-the-art single-machine out-of-core systems include GraphChi (Kyrola et al., 2012), TurboGraph (Han et al., 2013b), X-Stream (Roy et al., 2013), PathGraph (Yuan et al., 2014), GridGraph (Zhu XW et al., 2015), GraphQ (Wang K et al., 2015), MMap (Sabrin et al., 2013), Mosaic (Maass et al., 2017), Graspan (Wang K et al., 2017), and RStream (Wang K et al., 2018).

GraphChi proposes parallel sliding windows (PSW) that reduce non-sequential access to the disk, thus contributing to a more efficient asynchronous computing model. TurboGraph uses a record identity document (RID) table instead of a mapping table to store the billion vertex IDs, and the pin-and-slide computing model uses the buffer pool to improve performance. X-Stream uses an edge-centric rather than vertex-centric model and reduces random disk access through streaming completely unordered edge lists. PathGraph uses a path-centric graph processing model, which significantly improves the memory and disk locality for iterative computation algorithms on large graphs. GridGraph applies different partitioning strategies to vertices and edges, and groups edges into grids. The dual sliding windows guarantee the locality of vertex access. GraphQ uses a partition-check-refine programming model to support programmable analytical queries processed through incremental access to graph data, and the abstraction refinement algorithm provides efficient query processing. MMap uses the memory mapping, which treats edge files as if they were fully loaded into memory but actually do not fit in memory. The memory-mapped file reduces data copy to and from the user-space buffer, which improves efficiency. Mosaic leverages the hybrid execution model. Vertex-centric operations on a global graph are assigned to fast host processors, and edge-centric operations on the local graph are assigned to co-processors, thus achieving scale-up and scale-out and enabling graph analytics on one trillion edges. Graspan turns the programs into graphs and treats the inter-procedural program analysis as graph traversal. Then we can leverage parallel graph processing systems to analyze large programs efficiently. RStream is designed for processing streaming graphs.

FalshGraph (Da et al., 2015) and Graphene (Liu and Huang, 2017) are two semi-external memory graph engines. The former stores the vertex state in memory and edge lists on SSDs. The latter reads the graph data on SSDs while managing the metadata in DRAM. Both perform better than several external-memory systems and are comparable to memory-shared systems.

3.2.3 Distributed shared-memory systems

A distributed system is composed of multiple processing nodes and each node has its own memory. As a result, compared with single-machine shared-memory systems, distributed systems are less limited by hardware evolution for scalability. However, it is a challenge to find appropriate data partitioning strategies which greatly affect the performance. Data are partitioned over nodes and communication is required during computation, so communication between machines becomes a performance bottleneck. The overall performance and data scale are also limited by the network bandwidth (Zhang YM et al., 2017a, 2019).

Google's Pregel (Malewicz et al., 2010) is the first shared-memory distributed graph processing system leveraging the bulk synchronous parallel (BSP) model (Valiant, 1990) and providing a synchronous vertex centric framework. Piccolo (Power and Li, 2010) is an open-source implementation of Pregel. It uses distributed tables to perform distributed graph processing.

Many other distributed systems have been developed on top of Pregel. Trinity (Shao et al., 2013) organizes the memory of multiple machines into a globally addressable memory cloud, which is essentially a distributed key-value store. GraphLab (Low et al., 2010, 2012) is an asynchronous distributed shared-memory abstraction designed for machine learning and data mining algorithms on graphs. GraphX (Gonzalez et al., 2014) unifies advances in graph processing systems with advances in dataflow systems. It can efficiently execute general-purpose dataflow operations as well as graph computation.

PowerGraph (Gonzalez et al., 2012) is designed to tackle the challenge of power-law graphs. PowerGraph supports both the highly parallel

bulk-synchronous Pregel model of computation and the computationally efficient asynchronous GraphLab model of computation. PowerSwitch (Xie CN et al., 2015) and PowerLyra (Chen R et al., 2015) are designed on top of Powergraph. PowerSwitch makes the first comprehensive characterization on the performance of the synchronous and asynchorous modes on a set of typical graph-parallel applications, and proposes Hsync, a hybrid graph computation mode that adaptively switches a graph-parallel program between the two modes for optimal performance. PowerLyra provides different partition and computation strategies for low-degree vertices and high-degree vertices. Gemini (Zhu XW et al., 2016) proposes that computation, rather than communication, appears to be the actual bottleneck of evaluated distributed systems, so Gemini is computation-centric. It identifies an effective chunk-based graph partitioning scheme and adapts Ligra's hybrid push-pull computation model. Cyclops (Chen R et al., 2014) executes synchronous computing over a distributed immutable view, granting a vertex with read-only access to all its neighboring vertices. Cube (Zhang MX et al., 2016) based on 3D partitioning is a novel category of task partition algorithms that significantly reduces network traffic for certain machine learning and data mining (MLDM) applications. Naiad (Murray et al., 2013), KickStarter (Vora et al., 2017), Wukong+s (Zhang YH et al., 2017), and DS2 (Kalavri et al., 2018) are designed for streaming graphs.

3.2.4 Distributed out-of-core systems

Single-machine out-of-core systems can be expanded to distributed out-of-core systems. There is only one such system called Chaos (Roy et al., 2015).

Chaos builds on X-Stream to scale out graph processing systems based on secondary storage to multiple machines. It consists of a computation sub-system and a storage sub-system. It uses a very cheap partitioning scheme and achieves sequential access to secondary storage, leading to a short pre-processing time. The load balance is guaranteed by work stealing. However, there exist design flaws in Chaos, which will become a system bottleneck along with the scale out of the system. Separation of the computation sub-system and the storage sub-system increases system complexity and causes inevitable communication overhead.

4 Implementation techniques

4.1 Programming model

Graph processing algorithms are commonly based on a series of iterative operations. According to the basic unit of iteration, the programming model of graph processing systems can be divided into three categories. Vertex-centric model is used by most systems because it is simple to program. Edge-/path-centric model reduces random access to edges, which is suitable specifically for disk-based systems to significantly improve performance. Graph-centric model can reduce the overall communication and the number of iterations.

4.1.1 Vertex-centric model

The first published vertex-centric graph processing system is Pregel (Malewicz et al., 2010). Vertex-centric means the basic computation unit is a vertex. User-defined programs are executed on vertices in each iteration, and the intermediate information is passing to the adjacent vertices along with edges for the next iteration. It is easy to know that the vertex-centric model provides a natural way to express and compute many iterative graph algorithms (Yoo et al., 2005; Malewicz et al., 2010; Gonzalez et al., 2012). Pregel and many other vertex-centric systems are implemented on the BSP model and employ synchronous execution. The flaws mentioned in Section 3.1 inspire other systems to implement asynchronous execution, where an update function is able to use the most recent values of the edges and vertices. This has been demonstrated to improve performance in some instances (Xie CN et al., 2015). Most state-of-the-art graph systems are implemented on top of vertex-centric models, like Giraph, HAMA, Spark (Zaharia et al., 2010), and PowerGraph. The original BSP model is not suitable for real-world power-law graphs, which have the characteristic that a small subset of vertices connects to a large fraction of the graph. As a result, PowerGraph proposes a gather-apply-scatter (GAS) model based on the BSP model. The gather and scatter operations are implemented on the edge level rather than the vertex level. VoteToHalt is adapted to the convergence of graph algorithms. Each vertex has a state of active or inactive, but only active vertices will take part in iterations. When the states of all

vertices change into inactive, the iterations stop. McCune et al. (2015) gave a detailed survey of vertex-centric frameworks for large-scale distributed graph processing.

4.1.2 Edge-/path-centric model

The vertex-centric programming model is easy to program and avoids unnecessary access to inactive vertices for some algorithms. However, there exists a lot of random access to the out-edges when message passes to the adjacent vertices, especially in out-of-core systems, leading to expensive I/O overhead. Random access to any storage medium delivers less bandwidth than sequential access. So, there must be some trade off between sequential and random access to improve performance.

The single-machine out-of-core vertex-centric system GraphChi (Kyrola et al., 2012) is the first system to avoid random access to edges using the PSW method. Vertices are divided into intervals. Each interval is associated with a shard and loads in-edges of vertices in the interval and fully into memory. The in-edges are sorted by the source, thus avoiding random access to the disk. However, preprocessing time is extra overhead.

X-Stream (Roy et al., 2013) is an edge-centric processing system and exposes a scatter-gather programming model to reduce random access to edges. Fig. 1a illustrates the common vertex-centric implementation of the scatter-gather programming model. Both the scatter and the gather phases iterate over all vertices. Fig. 1b is the edge-centric scatter-gather programming model. The edge-centric scatter phase scans the out-edges to obtain the source vertex. Updated messages will be sent along with the out-edges if the source vertices are in active state. The edge-centric gather phase scans the update message and updates the value of the destination vertex. Both the scatter and gather operations iterate over edges rather than vertices. X-Stream does not need to sort the edge list, thus reducing pre-processing delay in GraphChi.

A single-machine out-of-core system, PathGraph, implements a path-centric model. Traversal graph algorithms are fundamental in graph processing. The path-centric model partitions the graph into paths, each of which is represented as a traversal tree and a forward tree. Traversal algorithms using data locality are executed along the traversal trees.

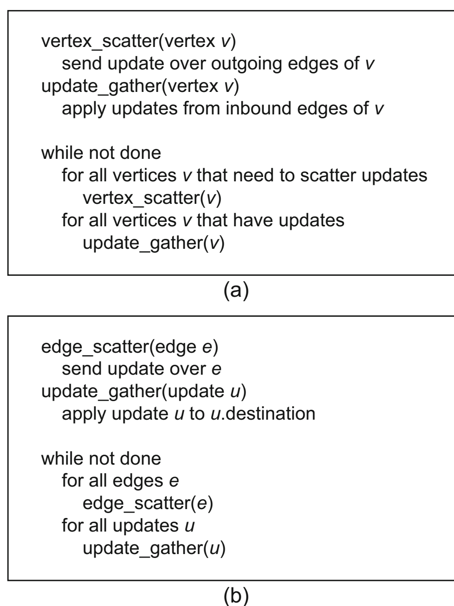

Fig. 1 Vertex-centric (a) and edge-centric (b) scatter-gather

4.1.3 Graph-centric model

Real-world graphs have the characteristics of skewed degree distribution, high density, and large diameter. The vertex-centric model partitions vertices along with their adjacent edges into different machines, which leads to load imbalance, heavy message passing, and more iterations on real-world graphs. The graph-centric model is proposed to address the bottlenecks created by real-world graphs in vertex-centric systems. The graph-centric model partitions a graph into fine-grained subgraph units, which are called blocks. Vertices and most of their neighbors are located in the same block and can be accessed sequentially in memory. This reduces message passing between machines. Iterations are executed in each block and then the block communicates with the other blocks, which also reduces message passing. Thanks to the coarser granularity, the number of iterations is also significantly reduced.

Giraph++ (Tian et al., 2013), GoFFish (Simmhan et al., 2014), and Blogel (Yan D et al., 2014) are three typical graph-centric systems. The communication between machines is significantly reduced. This is the main reason why graph-centric systems perform better than vertex-centric systems. However, more redundant information is produced and passed on the local machine with higher local

processing overhead. As a result, the graph-centric model cannot completely replace the vertex-centric model. Which model should be used depends on the characteristic of graph data and graph applications.

4.2 Partitioning strategy

Graph partitioning is an obvious prerequisite for distributed graph processing systems. A good partition algorithm should ensure fewer crossing edges between subgraphs, since crossing edges incur communication overhead. On the other hand, a good partition algorithm should use data locality to speed up graph processing. However, balanced graph partitioning is NP-complete (Garey et al., 1974). Buluç et al. (2016) gave a detailed survey of graph partitioning. Similar to the distributed graph partitioning strategy, the main purpose of the disk-based graph partitioning strategy is to optimize data locality and reduce the communication between machines or reading/writing with the disk. Here follows four online partitioning methods used by most graph processing systems.

4.2.1 Edge cut

Traditional balanced p-way edge-cut (Fig. 2) evenly assigns vertices of a graph to n machines in the cluster (Schloegel et al., 2000; Stanton and Kliot, 2012) . Completing information for each vertex is stored only once, but some edges would be split into two machines. This strategy is easy to implement by a hash function with vertices' IDs as input. This strategy saves storage space, but the disadvantage is that when we do computation based on the edges, messages along with the split edges would be transmitted across the machines, thus leading to high internal network communication overhead. However, balanced p-way edge-cut with simple logic does not need any relevant statistics information. This increases the partition efficiency and accelerates the processing.

Many distributed systems use the balanced p-way edge-cut. Edge-cut creates replicated vertices and edges to form a locally consistent graph state in each machine. However, real-world natural graphs with skewed distribution are difficult to partition with edge-cut (Lang, 2004; Abou-Rjeili and Karypis, 2006; Zhang YM et al., 2017b), since skewed vertices will cause a burst of communication cost and work imbalance.

4.2.2 Vertex cut

In the case of natural graphs, both Pregel and GraphLab using edge-cut are forced to resort to hash-based (random) partitioning, which has extremely poor locality.

PowerGraph uses a new fast approach to data layout for power-law graphs in distributed environments. It allows a single vertex-program to span multiple machines. GraphBuilder (Jain et al., 2013) and LightGraph (Zhao et al., 2014) use the same partitioning strategy as PowerGraph.

The balanced p-way vertex-cut (Fig. 3) can improve work balance and reduce communication and storage overhead by evenly assigning edges to machines in the cluster (Rahimian et al., 2014). There is a global mapping table to record the distribution of data. Compared to edge-cut, vertex-cut achieves load balance by allowing edges of a single vertex to be split over multiple machines, which helps a lot with high-degree vertices in skewed graphs.

Although vertex-cut works well in parallel processing, equally treating vertices with high and low degrees will cause a communication problem. If the edges of low-degree vertex are evenly assigned to multiple machines, high communication overhead among the machines will be caused and the additional overhead will affect the performance of the iteration.

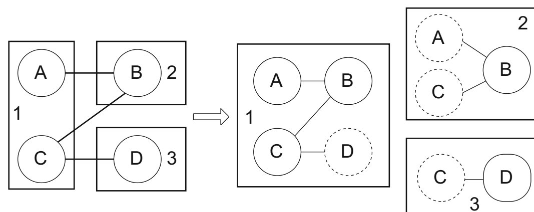

Fig. 2 An edge-cut example of a graph into three parts, where solid and dashed vertices are ghosts and mirrors, respectively (Gonzalez et al., 2012)

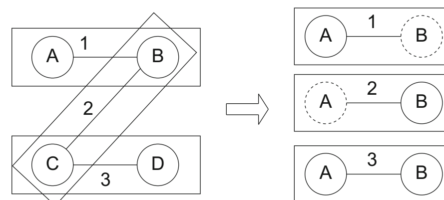

Fig. 3 A vertex-cut example of a graph into three parts, where solid and dashed vertices are ghosts and mirrors, respectively (Gonzalez et al., 2012)

The edge-cut and vertex-cut adopt a "one size fits all" design, where different vertices are equally processed, leading to suboptimal performance and scalability. Furthermore, distributed vertex-cut results in high locking overhead, thus decreasing the computing efficiency.

Balanced p-way hybrid-cut proposed by PowerLyra focuses on both high- and low-degree vertices. It uses differentiated partitioning to low- and high-degree vertices. For low-degree vertices, hybrid-cut adopts low-cut to evenly assign vertices along with in-edges to machines by hashing their target vertices. For high-degree vertices, hybrid-cut adopts high-cut to distribute all in-edges by hashing their source vertices. After that, hybrid-cut creates replicas and constructs local graphs, as is done in typical vertex-cuts.

Hybrid-cut addresses the major issues in edge- and vertex-cuts on skewed graphs. However, the threshold between low degree and high degree is hard to define. Different thresholds will seriously influence the partition result and performance.

4.2.3 Parallel sliding windows cut

The single-machine-disk-based system GraphChi proposes PSW for partitioning very large graphs from disk. Data in the disk are partitioned into p intervals according to the destination of vertices, each of which can be loaded completely into memory. Each interval is associated with a shard, and in-edges for the vertices in interval (p) are sorted by their source. One interval is loaded into the memory once for execution, and user-defined functions will be executed on vertices in parallel in the interval. In-edges are loaded into the memory, while out-edges are needed to sequentially read from other intervals on the disk. The update of the interval will be written back to the disk after all the processing is finished. The number of non-sequential disk writes for an execution interval is P, the same as the number of reads. The novel partitioning strategy for data from disk significantly reduces the number of non-sequential I/Os to the disk. Fig. 4 shows an illustration of the operation of PSW-cut on a toy graph.

4.2.4 Grid cut

The PSW-cut strategy needs to be sorted by source for in-edges in each shard. It consumes preprocessing time to finish the sorting over all of the shards. Based on PSW, GridGraph (Zhu XW et al., 2015) proposes a two-level hierarchical partitioning scheme to improve the locality and reduce the number of I/Os. Grid cut equally partitions vertices into P one-dimensional chunks based on the range. Edges are partitioned into a $P \times P$ two-dimensional block grid. The row of the blocks represents the source vertex and the column of the blocks represents the destination vertex. The edge blocks can be transformed from an unsorted edge list. During computation, small blocks will be appended to larger blocks to use the I/O bandwidth. Besides the reduction of preprocessing overhead, grid cut provides a selective scheduling mechanism which is not supported by PSW cut. I/O to blocks without active edges can be controlled to decrease unnecessary access, which contributes a lot to many graph algorithms (BFS and WCC) for most inactive edges.

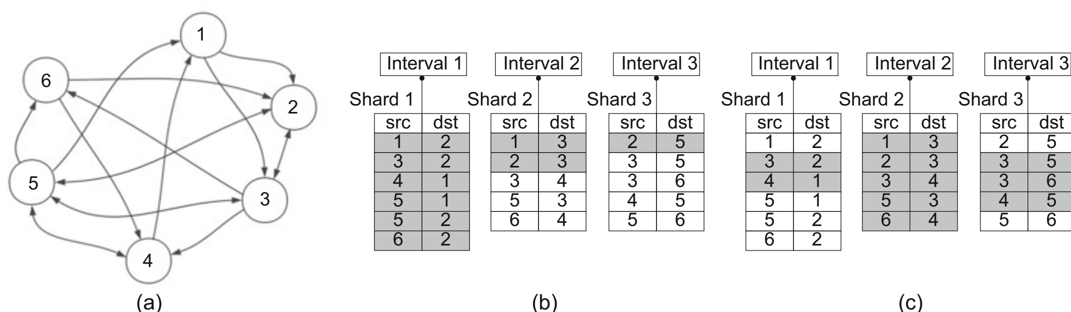

Fig. 4 Operation of PSW-cut on a toy graph: (a) the example graph with six vertices is partitioned into three equal intervals according to the destination; (b) PSW begins by executing interval 1 and the shaded parts are loaded into memory; (c) PSW moves to the next interval. Reprinted from Kyrola et al. (2012), Copyright 2012, with permission from the authors

4.3 Communication model

In graph applications, vertices need to send messages along with the edges to their neighbors. Generally speaking, the number of edges is far larger than that of vertices in most graphs. So, it means a lot to appropriately process the message, especially in large-scale graphs. Otherwise, communication will become the bottleneck of the performance.

4.3.1 Push style

The push style communication model is source-vertex centric. The graph program traverses source-vertices and executes user-defined update functions. Then messages will be sent to the neighbors through the out-edges. In other words, a vertex can learn its neighbors' values via only the messages that its neighbors push to it. Push style is easy to implement. Parallelism can be guaranteed when there is enough memory. Nevertheless, a large amount of disk I/O overhead will appear when memory is limited. In synchronous execution, computation and update of a vertex will start only when messages from all of the former iterations have been received. In addition, the destination vertex needs to save all of the messages until source vertices finish pushing. This means the local machine needs to buffer all the messages from remote vertices, and perform random access to retrieve the messages when running algorithms on the local vertices, thus apparently increasing storage pressure on the receiving end.

Pregel, GPS, Giraph, and many other systems use the push style communication model. Giraph supports both in-memory and out-of-core modes. The distributed shared-memory system Trinity optimizes message passing by creating a bipartite partition of the local and remote graphs. Messages required by the local vertices are all in the same partition. Messages need only to be delivered once from the remote machine to the local machine and can be processed in time. However, bipartite partitioning is NP-hard. Trinity ignores remote vertices with high degree, and messages pushed by this category of vertices need to be saved into memory, which is inefficient for power-law graphs in the real-world.

4.3.2 Pull style

The pull style communication model is destination-vertex centric. The graph program traverses destination vertices and asks related messages from source vertices. The user-defined update function will be executed when all related messages have been pulled. It is unnecessary to save a lot of messages into memory for the pull style. However, the source vertex ID used to locate the related source vertex also needs to be transferred during asking messages, thus adding extra communication in distributed systems. In some graph algorithms, especially SSSP, most vertices have reached the steady state when the algorithm is close to convergence. Pulling messages for these steady vertices causes unnecessary overhead.

Kylin (Ho et al., 2013) presents an optimization technique for the native pulling message model. Vertices are tagged with active state or inactive state. Only active vertices that will perform computation need to pull data from all their neighbors before computation. In this way, the pull model can eliminate a significant number of unnecessary messages. Shared-memory systems like GraphLab also use the pull model for message passing. They keep shared vertex states in memory, and then a remote pull operation can be translated to a local operation. Zhou et al. (2014) found that the vertex-replication strategy of GraphLab incurs huge memory consumption especially for large datasets. Even worse, if the memory space is limited and shared states are stored on disk, the frequent access to shared states will lead to a large number of disk I/Os.

Gemini (Zhu XW et al., 2016) is a distributed system adopting a hybrid push-pull communication model. In the push mode, master vertices push messages to there mirrors and then mirrors update their neighbors through the out-edges. In the pull mode, mirror vertices first execute the user-defined update function based on data pulling from neighbors along with in-edges and then send the update messages to their master.

Fig. 5 details an example of PR pseudocode in push and pull styles, from which the differences of the two models can be distinguished intuitively.

4.4 Execution model

The execution model of distributed graph processing systems can be divided into two categories: synchronous and asynchronous execution. In the synchronous execution, synchronous control is needed after one iteration is finished and then the

```
1   Vertex.compute(){
2      Message msgs=getRecMsg();
3      double sum=0.0;
4      for (Message m: msgs.iterator())
5         sum=sum+m.getValue();
6      double newVal=0.15/getMunVertices()+0.85*sum;
7      setValve(newVal);

8      for (Edge e: getOutEdges().iterator())
9         sendMsgTo(e.Target(),getValue()/getOutDegree());

10     if (getNumSupersteps()>maxNum)
11        voteToHalt();
12 }
```

(a)

```
1   Vertex.pullRes(){

2      if (getUpd() is true)
3         for (Edge e: getOutEdges.iterator())
4            sendMsgTo(e.Target(), getValue()/getOutDegree());
5   }

6   Vertex.update(){
7      Messages msgs=getRecMsg();
8      double sum=0.0;
9      for (Message m: msgs.iterator())
10        sum=sum+m.getValue();
11     double newVal=0.15/getMunVertices()+0.85*sum;
12     setValue(newVal);
13     setUpd();
14     if (getNumSupersteps()>maxNum)
15        voteToHalt();
16 }
```

(b)

Fig. 5 PageRank algorithm in push (a) and pull (b) styles

next iteration starts. The synchronous execution of graph systems is often based on the BSP model. A BSP program is composed of a series of supersteps, each of which consists of a computation where each processor uses only locally held values, a global message transmission from each processor to any subset of the others, and barrier synchronization.

Fig. 6 is an illustration of the BSP model. The BSP model is the mainstream of distributed graph processing. The most famous one is Pregel. Many open-source projects are designed around the BSP model following Pregel. HAMA and Giraph are two representatives. Systems referring to Pregel continue to appear recently, such as X-pregel (Bao and Suzumura, 2013), Pregelix (Bu et al., 2014), and Giraph++, which are closely related to Pregel and Giraph.

There exist many limitations in the synchronous execution. Jobs are different from each other in terms of execution speed as the data scale and complexity can be quite different. Frequent synchronous

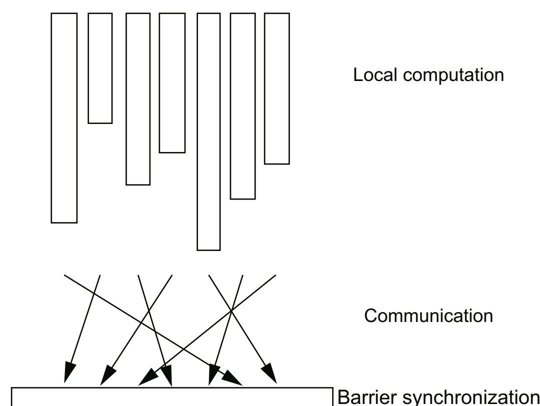

Fig. 6 Illustration of the processing of the bulk synchronous parallel model

operations make the slowest job become the bottleneck of performance, which creates a lower convergence speed.

Asynchronous computation has been demonstrated to speed up the convergence of many iterative graph analytics. The destination vertex can immediately execute the update function without waiting until all the other vertices have received messages. The asynchronous execution model works well to eliminate bucket effect and reduce the synchronous overhead, but it increases the difficulty of programming and debugging.

4.5 Fault tolerance

Distributed systems must pay attention to the potential failure of one or more nodes during computation. When an execution node fails, the jobs running on the node fail, and data on the node may be lost. An efficient strategy must be adopted to make sure that the processing can be continued. That is called fault tolerance.

Checkpointing is commonly used in many distributed graph processing systems. Take Pregel as an example. The node will set up one checkpoint every few supersteps. The vertex values, edge values, and incoming messages will be saved in persistent storage, like HDFS. When one or more nodes fail, they will reload the system state from the latest checkpoint. Then the node recovers and restarts to compute from the recovery state. Wang P et al. (2014) evaluated the overhead of checkpoint based on HAMA to estimate checkpoint and recovery cost. Checkpointing mechanism will cause notable

overhead during normal computation and a lengthy recovery time. As a result, most systems use it but disable it by default.

GraphLab proposes asynchronous checkpointing based on the Chandy-Lamport snapshot. The asynchronous snapshot slows down execution only when the synchronous snapshot stops execution. So, the benefit of asynchronous snapshot is obvious.

Replication-based fault-tolerance mechanism Imitator was presented in Wang P et al. (2014). Imitator is implemented based on the fact that many distributed graph parallel systems require creating replicas of vertices to provide local access semantics such that graph computation can be programmed as accessing local memory. So, Imitator leverages the existing replicas to achieve fault tolerance.

Partition-based fault-tolerance mechanism was proposed in Shen YY et al. (2014) to enable faster failure recovery than checkpointing. Failed partitions will be reassigned to healthy nodes and the workload will be rebalanced. It accelerates the recovery process through simultaneous reduction of recovery communication cost and parallelization of recovery computations.

5 Recent advances and open challenges

Fig. 7 provides a timeline representation of graph processing systems. It is obvious that there is increasing interest in developing graph processing platforms. Besides aforementioned typical techniques, many graph processing systems take advantage of the latest technology or hardware to provide better performance.

Recently, researchers have focused on static graphs. However, many modern graphs are dynamic, with new edges and vertices being added at high rates, and some edges and vertices may be removed. How to maintain a large amount of updating in dynamic graphs and how to achieve efficient real-time computation? It is a challenge for processing dynamic graphs. Some efforts have been devoted to dynamic graph partitioning (Huang and Abadi, 2016). LEOPARD was recently proposed to incorporate replication alongside dynamic partitioning. LEOPARD is light-weight and updates the partitioning as new changes are streamed into the system, which performs better than updating after a batch of changes. Vora et al. (2016) proposed a dynamic

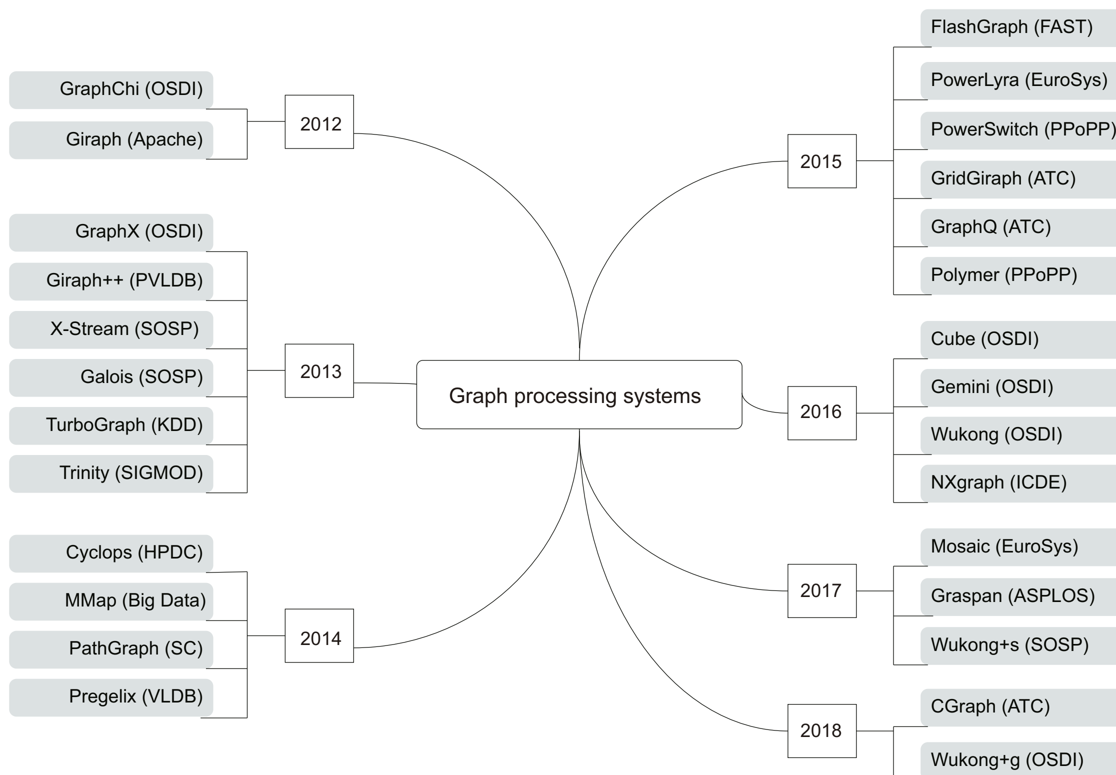

Fig. 7 Timeline representation of graph processing systems

partitioning strategy instead of static partitioning in static graphs. This dynamic partitioning strategy reduces I/O by loading only the partial partition data that have to be updated, rather than the entire partition data.

There is increasing interest in remote direct memory access (RDMA) to reduce communication overhead in distributed graph processing systems. RDMA represents the ability of accessing memory on the remote machine without interrupting the processing of the CPU(s) on that system. RDMA has three main characteristics: low latency, low CPU overhead, and high bandwidth. Wukong (Shi et al., 2016) proposed in 2016 uses one-side RDMA to accelerate the process of SPARQL queries. At the same time, there has been increasing interest from both academia and industry in building RDMA-based data center applications (Dragojević et al., 2014; Binnig et al., 2016; Chen YZ et al., 2016). The characteristic of RDMA makes it also suitable for reducing replication overhead for fault tolerance (Taleb et al., 2018).

Emerging storages such as SSD, NVM, and DDR4 provide opportunity for improving the scalability of single-machine graph processing systems. Graph processing accelerators such as GPU, Xeon-Phi, and FPGA make it possible to break the bottlenecks in graph processing. There is an emerging trend of using specific graph processing accelerators as a high-performance computing tool for graph computation and analysis.

Recently, a significant amount of research has been devoted to deep learning on graph data, which injects vitality to graph processing. The research upon graph mining, graph databases also flourishes.

There are still many challenges. How to deal with graphs in super-large scale is an important problem to be solved urgently. How to work out the large amount of communication between data centers that are far apart from each other? How to solve the graph partitioning problem and achieve workload balancing? Recent research focuses on static graphs with simple graph attribution, but how about dynamic graphs and graphs with complex structure and attributions? How to achieve real-time computation on dynamic graphs and reasonable partitioning of complex property graphs?

6 Conclusions

Over the past few years, graph data have become powerful and significant in machine learning, data mining and many other domains. This development is attributed to to the specific structure that graph data maintains. In this survey, first we have conducted a comprehensive review of graph processing systems. To understand graph processing systems, we should figure out the characteristic of graph workloads. So, we first introduce the workloads and applications of graph processing. Then we summarize several existing solutions and divide them into general-purpose and specific systems. We introduce four categories of specific systems to give a detailed review. In terms of implementation, we overview the programming models, partitioning strategies, communication models, execution models, and fault tolerance strategies. Finally, we summarize the recent advances and list some open challenges that indicate future research directions.

Contributors

Dong-sheng LI guided the research. Ning LIU designed the research and drafted the manuscript. Xiong-lve LI helped organize the manuscript. Dong-sheng LI and Yi-ming ZHANG revised and edited the final version.

Compliance with ethics guidelines

Ning LIU, Dong-sheng LI, Yi-ming ZHANG, and Xiong-lve LI declare that they have no conflict of interest.

References

Abou-Rjeili A, Karypis G, 2006. Multilevel algorithms for partitioning power-law graphs. Proc 20th IEEE Int Parallel and Distributed Processing Symp, Article 10. https://doi.org/10.1109/IPDPS.2006.1639360

Ajwani D, Dementiev R, Meyer U, 2006. A computational study of external-memory BFS algorithms. Proc 17th Annual ACM-SIAM Symp on Discrete Algorithm, p.601-610.

Ajwani D, Meyer U, Osipov V, 2007. Improved external memory BFS implementations. Proc Meeting on Algorithm Engineering and Expermiments, p.3-12.

Arge L, Brodal GS, Toma L, 2000. On external-memory MST, SSSP, and multi-way planar graph separation. Proc 7th Scandinavian Workshop on Algorithm Theory, p.433-447. https://doi.org/10.1007/3-540-44985-X_37

Atwood J, Towsley D, 2016. Diffusion-convolutional neural networks. https://arxiv.org/abs/1511.02136

Avery C, 2011. Giraph: large-scale graph processing infrastructure on Hadoop. Proc Hadoop Summit, p.5-9.

Awerbuch B, Gallager RG, 1985. Distributed BFS algorithms. 26th Annual Symp on Foundations of Com-

puter Science, p.250-256.
https://doi.org/10.1109/SFCS.1985.20

Bader DA, Cong G, 2006. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J Parall Distr Comput*, 66(11):1366-1378. https://doi.org/10.1016/j.jpdc.2006.06.001

Bader DA, Madduri K, 2006. Parallel algorithms for evaluating centrality indices in real-world networks. Int Conf on Parallel Processing, p.539-550. https://doi.org/10.1109/ICPP.2006.57

Bao NT, Suzumura T, 2013. Towards highly scalable pregel-based graph processing platform with x10. Proc 22nd Int Conf on World Wide Web, p.501-508. https://doi.org/10.1145/2487788.2487984

Batarfi O, El Shawi R, Fayoumi AG, et al., 2015. Large scale graph processing systems: survey and an experimental evaluation. *Clust Comput*, 18(3):1189-1213. https://doi.org/10.1007/s10586-015-0472-6

Baumes J, Goldberg M, Magdon-Ismail M, 2005. Efficient identification of overlapping communities. IEEE Int Conf on Intelligence and Security Informatics, p.27-36. https://doi.org/10.1007/11427995_3

Becchetti L, Boldi P, Castillo C, et al., 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. Proc 14th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.16-24. https://doi.org/10.1145/1401890.1401898

Belkin M, Niyogi P, 2001. Laplacian eigenmaps and spectral techniques for embedding and clustering. Proc 14th Int Conf on Neural Information Processing Systems, p.585-591.

Binnig C, Crotty A, Galakatos A, et al., 2016. The end of slow networks: it's time for a redesign. *Proc VLDB Endowm*, 9(7):528-539. https://doi.org/10.14778/2904483.2904485

Borgelt C, Berthold MR, 2002. Mining molecular fragments: finding relevant substructures of molecules. IEEE Int Conf on Data Mining, p.51-58. https://doi.org/10.1109/ICDM.2002.1183885

Brandes U, 2001. A faster algorithm for betweenness centrality. *J Math Sociol*, 25(2):163-177. https://doi.org/10.1080/0022250X.2001.9990249

Bruna J, Zaremba W, Szlam A, et al., 2014. Spectral networks and locally connected networks on graphs. https://arxiv.org/abs/1312.6203

Bu YY, Howe B, Balazinska M, et al., 2010. HaLoop: efficient iterative data processing on large clusters. *Proc VLDB Endowm*, 3(1-2):285-296. https://doi.org/10.14778/1920841.1920881

Bu YY, Borkar V, Jia J, et al., 2014. Pregelix: big(ger) graph analytics on a dataflow engine. *Proc VLDB Endowm*, 8(2):161-172. https://doi.org/10.14778/2735471.2735477

Buluç A, Madduri K, 2011. Parallel breadth-first search on distributed memory systems. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 65. https://doi.org/10.1145/2063384.2063471

Buluç A, Meyerhenke H, Safro I, et al., 2016. Recent advances in graph partitioning. In: Kliemann L, Sanders P (Eds.), Algorithm Engineering. Springer, Cham, p.117-158. https://doi.org/10.1007/978-3-319-49487-6_4

Chan TM, 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J Comput*, 39(5):2075-2089. https://doi.org/10.1137/08071990X

Chang LJ, Lin XM, Zhang WJ, et al., 2015. Optimal enumeration: efficient top-k tree matching. *Proc VLDB Endowm*, 8(5):533-544. https://doi.org/10.14778/2735479.2735486

Chen R, Weng X, He B, et al., 2010. Large graph processing in the cloud. Proc ACM SIGMOD Int Conf on Management of Data, p.1123-1126. https://doi.org/10.1145/1807167.1807297

Chen R, Ding X, Wang P, et al., 2014. Computation and communication efficient graph processing with distributed immutable view. Proc 23rd Int Symp on High-Performance Parallel and Distributed Computing, p.215-226. https://doi.org/10.1145/2600212.2600233

Chen R, Shi J, Chen Y, et al., 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. 10th European Conf on Computer Systems, Article 1.

Chen YZ, Wei XD, Shi JX, et al., 2016. Fast and general distributed transactions using RDMA and HTM. Proc 11th European Conf on Computer Systems, Article 26. https://doi.org/10.1145/2901318.2901349

Cheung TY, 1983. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans Softw Eng*, 9(4):504-512. https://doi.org/10.1109/TSE.1983.234958

Chi Y, Dai G, Wang Y, et al., 2016. NXgraph: an efficient graph processing system on a single machine. IEEE 32nd Int Conf on Data Engineering, p.409-420. https://doi.org/10.1109/ICDE.2016.7498258

Da Z, Mhembere D, Burns R, et al., 2015. FlashGraph: processing billion-node graphs on an array of commodity SSDS. Proc 13th USENIX Conf on File and Storage Technologies, p.45-58.

Dean J, Ghemawat S, 2008. MapReduce: simplified data processing on large clusters. *Commun ACM*, 51(1):107-113. https://doi.org/10.1145/1327452.1327492

Defferrard M, Bresson X, Vandergheynst P, 2016. Convolutional neural networks on graphs with fast localized spectral filtering. https://arxiv.org/abs/1606.09375

Desikan P, Pathak N, Srivastava J, et al., 2005. Incremental page rank computation on evolving graphs. Special Interest Tracks and Posters of the 14th Int Conf on World Wide Web, p.1094-1095. https://doi.org/10.1145/1062745.1062885

Doekemeijer N, Varbanescu AL, 2014. A Survey of Parallel Graph Processing Frameworks. Technical Report No. PDS-2014-003, Delft University of Technology, the Netherlands.

Dragojević A, Narayanan D, Hodson O, et al., 2014. FaRM: fast remote memory. Proc 11th USENIX Conf on Networked Systems Design and Implementation, p.401-414.

Duvenaud D, Maclaurin D, Aguilera-Iparraguirre J, et al., 2015. Convolutional networks on graphs for learning molecular fingerprints. Proc 28th Int Conf on Neural Information Processing Systems, p.2224-2232.

Ekanayake J, Li H, Zhang B, et al., 2010. Twister: a runtime for iterative MapReduce. Proc 19th ACM Int Symp on High Performance Distributed Computing, p.810-818.

Farkas IJ, Ábel D, Palla G, et al., 2007. Weighted network modules. *New J Phys*, 9(6):180. https://doi.org/10.1088/1367-2630/9/6/180

Garey MR, Johnson DS, Stockmeyer L, 1974. Some simplified NP-complete problems. Proc 6th Annual ACM Symp on Theory of Computing, p.47-63. https://doi.org/10.1145/800119.803884

Gonzalez JE, Low Y, Gu H, et al., 2012. PowerGraph: distributed graph-parallel computation on natural graphs. Proc 10th USENIX Conf on Operating Systems Design and Implementation, p.17-30.

Gonzalez JE, Xin RS, Dave A, et al., 2014. GraphX: graph processing in a distributed dataflow framework. Proc 11th USENIX Conf on Operating Systems Design and Implementation, p.599-613.

Han WS, Lee J, Lee JH, 2013a. TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. Proc Int Conf on Management of Data, p.337-348.

Han WS, Lee S, Park K, et al., 2013b. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. Proc 19th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.77-85. https://doi.org/10.1145/2487575.2487581

Harish P, Vineet V, Narayanan P, 2009. Large graph algorithms for massively multithreaded architectures. Technical Report No. IIIT/TR/2009/74. Centre for Visual Information Technology, University of Hyderabad, India.

Hirschberg DS, Chandra AK, Sarwate DV, 1979. Computing connected components on parallel computers. *Commun ACM*, 22(8):461-464. https://doi.org/10.1145/359138.359141

Ho LY, Li TH, Wu JJ, et al., 2013. Kylin: an efficient and scalable graph data processing system. IEEE Int Conf on Big Data, p.193-198. https://doi.org/10.1109/BigData.2013.6691574

Holder LB, Cook DJ, Djoko S, 1994. Substructure discovery in the SUBDUE system. Proc 3rd Int Conf on Knowledge Discovery and Data Mining, p.169-180.

Huan J, Wang W, Prins J, 2003. Efficient mining of frequent subgraphs in the presence of isomorphism. 3rd IEEE Int Conf on Data Mining, p.549-552. https://doi.org/10.1109/ICDM.2003.1250974

Huan J, Wang W, Prins J, et al., 2004. SPIN: mining maximal frequent subgraphs from graph databases. 10th Int Conf on Knowledge Discovery and Data Mining, p.581-586. https://doi.org/10.1145/1014052.1014123

Huang J, Abadi DJ, 2016. Leopard: lightweight edge oriented partitioning and replication for dynamic graphs. *Proc VLDB Endowm*, 9(7):540-551. https://doi.org/10.14778/2904483.2904486

Inokuchi A, Washio T, Motoda H, 2000. An Apriori-based algorithm for mining frequent substructures from graph data. European Conf on Principles of Data Mining and Knowledge Discovery, p.13-23. https://doi.org/10.1007/3-540-45372-5_2

Jain N, Liao G, Willke TL, 2013. GraphBuilder: scalable graph ETL framework. 1st Int Workshop on Graph Data Management Experiences and Systems, Article 4. https://doi.org/10.1145/2484425.2484429

Kalavri V, Liagouris J, Hoffmann M, et al., 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. 13th USENIX Symp on Operating Systems Design and Implementation, p.783-798.

Kamvar SD, Haveliwala TH, Manning CD, et al., 2003. Extrapolation methods for accelerating PageRank computations. Proc 12th Int Conf on World Wide Web, p.261-270. https://doi.org/10.1145/775152.775190

Kang U, Tsourakakis CE, Faloutsos C, 2009. PEGASUS: a peta-scale graph mining system implementation and observations. 9th IEEE Int Conf on Data Mining, p.229-238. https://doi.org/10.1109/ICDM.2009.14

Kelley S, 2009. The existence and discovery of overlapping communities in large-scale networks. PhD Thesis, Rensselaer Polytechnic Institute, Troy, NY, USA.

Kipf TN, Welling M, 2016a. Semi-supervised classification with graph convolutional networks. https://arxiv.org/abs/1609.02907

Kipf TN, Welling M, 2016b. Variational graph auto-encoders. https://arxiv.org/abs/1611.07308

Kolountzakis MN, Miller GL, Peng R, et al., 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Int Math*, 8(1-2):161-185. https://doi.org/10.1080/15427951.2012.625260

Kuramochi M, Karypis G, 2003. GREW: a scalable frequent subgraph discovery algorithm. 4th IEEE Int Conf on Data Mining, p.439-442. https://doi.org/10.1109/ICDM.2004.10024

Kuramochi M, Karypis G, 2004. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans Knowl Data Eng*, 16(9):1038-1051. https://doi.org/10.1109/TKDE.2004.33

Kutzkov K, Pagh R, 2014. Triangle counting in dynamic graph streams. Scandinavian Workshop on Algorithm Theory, p.306-318. https://doi.org/10.1007/978-3-319-08404-6_27

Kyrola A, Blelloch GE, Guestrin C, 2012. GraphChi: large-scale graph computation on just a PC. Proc USENIX Symp on Operating Systems Design and Implementation, p.31-46.

Lancichinetti A, Fortunato S, Kertész J, 2009. Detecting the overlapping and hierarchical community structure in complex networks. *N J Phys*, 11(3):19-44.

Lang K, 2004. Finding good nearly balanced cuts in power law graphs. Yahoo Research Labs, CA, USA. http://www.optimization-online.org/db_file/2004/12/1023.pdf [Assessed on Sept. 16, 2019].

Lee C, Reid F, Mcdaid A, et al., 2010. Detecting highly overlapping community structure by greedy clique expansion. 4th SNA-KDD Workshop on Social Network Mining and Analysis, p.1-10.

Leiserson CE, Schardl TB, 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). Proc 22nd Annual ACM Symp on Parallelism in Algorithms and Architectures, p.303-314. https://doi.org/10.1145/1810479.1810534

Liu H, Huang HH, 2017. Graphene: fine-grained IO management for graph computing. Proc 15th USENIX Conf on File and Storage Technologies, p.285-300.

Lotker Z, Patt-Shamir B, Peleg D, 2006. Distributed MST for constant diameter graphs. *Distr Comput*, 18(6):453-460. https://doi.org/10.1007/s00446-005-0127-6

Low Y, Gonzalez JE, Kyrola A, et al., 2010. GraphLab: a new framework for parallel machine learning. https://arxiv.org/abs/1408.2041

Low Y, Bickson D, Gonzalez J, et al., 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc VLDB Endowm*, 5(8):716-727. https://doi.org/10.14778/2212351.2212354

Ma H, Yang H, Lyu MR, et al., 2008. Mining social networks using heat diffusion processes for marketing candidates selection. Proc 17th ACM Conf on Information and Knowledge Management, p.233-242. https://doi.org/10.1145/1458082.1458115

Maass S, Min C, Kashyap S, et al., 2017. Mosaic: processing a trillion-edge graph on a single machine. Proc 20th European Conf on Computer Systems, p.527-543. https://doi.org/10.1145/3064176.3064191

Maheshwari A, Zeh N, 2001. I/O-efficient algorithms for graphs of bounded treewidth. Proc 12th Annual ACM-SIAM Symp on Discrete Algorithms, p.89-90.

Malewicz G, Austern MH, Bik AJ, et al., 2010. Pregel: a system for large-scale graph processing. Proc ACM SIGMOD Int Conf on Management of Data, p.135-146.

Matsumoto K, Nakasato N, Sedukhin SG, 2011. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. IEEE 13th Int Conf on High Performance Computing and Communications, p.145-152. https://doi.org/10.1109/HPCC.2011.28

McCune RR, Weninger T, Madey G, 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput Surv*, 48(2):25. https://doi.org/10.1145/2818185

Miao X, 2015. DynaDiffuse: a dynamic diffusion model for continuous time constrained influence maximization. Proc 29th AAAI Conf on Artificial Intelligence, p.346-352.

Mihalcea R, 2004. Graph-based ranking algorithms for sentence extraction, applied to text summarization. Proc ACL on Interactive Poster and Demonstration Sessions, Article 20. https://doi.org/10.3115/1219044.1219064

Murray DG, McSherry F, Isaacs R, et al., 2013. Naiad: a timely dataflow system. Proc 24th ACM Symp on Operating Systems Principles, p.439-455.

Nanongkai D, 2014. Distributed approximation algorithms for weighted shortest paths. Proc 46th Annual ACM Symp on Theory of Computing, p.565-573. https://doi.org/10.1145/2591796.2591850

Nguyen D, Lenharth A, Pingali K, 2013. A lightweight infrastructure for graph analytics. Proc 24th ACM Symp on Operating Systems Principles, p.456-471. https://doi.org/10.1145/2517349.2522739

Niepert M, Ahmed M, Kutzkov K, 2016. Learning convolutional neural networks for graphs. https://arxiv.org/abs/1605.05273

Nuutila E, Soisalon-Soininen E, 1994. On finding the strongly connected components in a directed graph. *Inform Process Lett*, 49(1):9-14. https://doi.org/10.1016/0020-0190(94)90047-7

Pan SR, Hu RQ, Long GD, et al., 2018 Adversarially regularized graph autoencoder for graph embedding. https://arxiv.org/abs/1802.04407

Power R, Li JY, 2010. Piccolo: building fast, distributed programs with partitioned tables. Proc 9th USENIX Conf on Operating Systems Design and Implementation, p.293-306.

Psorakis I, Roberts S, Ebden M, et al., 2011. Overlapping community detection using Bayesian non-negative matrix factorization. *Phys Rev E*, 83(2):066114. https://doi.org/10.1103/PhysRevE.83.066114

Rahimian F, Payberah AH, Girdzijauskas S, et al., 2014. Distributed vertex-cut partitioning. IFIP Int Conf on Distributed Applications and Interoperable Systems, p.186-200. https://doi.org/10.1007/978-3-662-43352-2_15

Ren XG, Wang JH, 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc VLDB Endowm*, 8(5):617-628. https://doi.org/10.14778/2735479.2735493

Rodriguez MA, 2015. The Gremlin graph traversal machine and language (invited talk). Proc 15th Symp on Database Programming Languages, p.1-10. https://doi.org/10.1145/2815072.2815073

Roy A, Mihailovic I, Zwaenepoel W, 2013. X-Stream: edge-centric graph processing using streaming partitions. Proc 24th ACM Symp on Operating Systems Principles, p.472-488. https://doi.org/10.1145/2517349.2522740

Roy A, Bindschaedler L, Malicevic J, et al., 2015. Chaos: scale-out graph processing from secondary storage. Proc 25th Symp on Operating Systems Principles, p.410-424. https://doi.org/10.1145/2815400.2815408

Sabrin KM, Lin Z, Chau DHP, et al., 2013. MMap: Mining Billion-Scale Graphs on a PC with Fast, Minimalist Approach via Memory Mapping. Technical Report No. GT-CSE-2013-04, Georgia Institute of Technology, Atlanta, USA.

Sakr S, Bajaber F, Barnawi A, et al., 2015. Big data processing systems: state-of-the-art and open challenges. Int Conf on Cloud Computing, p.1-8.

Sarma AD, Molla AR, Pandurangan G, et al., 2013. Fast distributed PageRank computation. Int Conf on Distributed Computing and Networking, p.11-26. https://doi.org/10.1007/978-3-642-35668-1_2

Scarselli F, Gori M, Tsoi AC, et al., 2009. The graph neural network model. *IEEE Trans Neur Netw*, 20(1):61-80. https://doi.org/10.1109/TNN.2008.2005605

Schloegel K, Karypis G, Kumar V, 2000. Parallel multilevel algorithms for multi-constraint graph partitioning. Proc 6th Int European Conf on Parallel Processing, p.296-310. https://doi.org/10.1007/3-540-44520-X_39

Seo S, Yoon EJ, Kim J, et al., 2010. HAMA: an efficient matrix computation with the MapReduce framework. IEEE Second Int Conf on Cloud Computing Technology and Science, p.721-726. https://doi.org/10.1109/CloudCom.2010.17

Shang HC, Zhang Y, Lin XM, et al., 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc VLDB Endowm*, 1(1):364-375. https://doi.org/10.14778/1453856.1453899

Shao B, Wang HX, Li YT, 2013. Trinity: a distributed graph engine on a memory cloud. Proc ACM SIGMOD Int Conf on Management of Data, p.505-516. https://doi.org/10.1145/2463676.2467799

Shen HW, Cheng XQ, Cai K, et al., 2008. Detect overlapping and hierarchical community structure in networks. *Phys A*, 388(8):1706-1712. https://doi.org/10.1016/j.physa.2008.12.021

Shen YY, Chen G, Jagadish HV, et al., 2014. Fast failure recovery in distributed graph processing systems. *Proc VLDB Endowm*, 8(4):437-448. https://doi.org/10.14778/2735496.2735506

Shi JX, Yao YY, Chen R, et al., 2016. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.317-332.

Shun JL, Blelloch GE, 2013. Ligra: a lightweight graph processing framework for shared memory. *ACM SIGPLAN Not*, 48(8):135-146. https://doi.org/10.1145/2442516.2442530

Simmhan Y, Kumbhare A, Wickramaarachchi C, et al., 2014. GoFFish: a sub-graph centric framework for large-scale graph analytics. European Conf on Parallel Processing, p.451-462. https://doi.org/10.1007/978-3-319-09873-9_38

Stanton I, Kliot G, 2012. Streaming graph partitioning for large distributed graphs. Proc 18th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.1222-1230. https://doi.org/10.1145/2339530.2339722

Sundaram N, Satish N, Patwary MMA, et al., 2015. GraphMat: high performance graph analytics made productive. *Proc VLDB Endowm*, 8(11):1214-1225. https://doi.org/10.14778/2809974.2809983

Taleb Y, Stutsman R, Antoniu G, et al., 2018. Tailwind: fast and atomic RDMA-based replication. USENIX Annual Technical Conf, p.850-863.

Tangwongsan K, Pavan A, Tirthapura S, 2013. Parallel triangle counting in massive streaming graphs. Proc 22nd ACM Int Conf on Information and Knowledge Management, p.781-786. https://doi.org/10.1145/2505515.2505741

Tian YY, Balmin A, Corsten SA, et al., 2013. From "think like a vertex" to "think like a graph." *Proc VLDB Endowm*, 7(3):193-204. https://doi.org/10.14778/2732232.2732238

Ullmann JR, 1976. An algorithm for subgraph isomorphism. *J ACM*, 23(1):31-42. https://doi.org/10.1145/321921.321925

Valiant LG, 1990. A bridging model for parallel computation. *Commun ACM*, 33(8):103-111. https://doi.org/10.1145/79173.79181

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. https://arxiv.org/abs/1706.03762

Veličković P, Cucurull G, Casanova A, et al., 2017. Graph attention networks. https://arxiv.org/abs/1710.10903

Vora K, Xu GH, Gupta R, 2016. Load the edges you need: a generic I/O optimization for disk-based graph processing. USENIX Annual Technical Conf, p.507-522.

Vora K, Gupta R, Xu GQ, 2017. KickStarter: fast and accurate computations on streaming graphs via trimmed approximations. Proc 22nd Int Conf on Architectural Support for Programming Languages and Operating Systems, p.237-251. https://doi.org/10.1145/3037697.3037748

Wang DX, Cui P, Zhu WW, 2016. Structural deep network embedding. 22nd ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.1225-1234. https://doi.org/10.1145/2939672.2939753

Wang K, Xu GH, Su Z, et al., 2015. GraphQ: graph query processing with abstraction refinement-scalable and programmable analytics over very large graphs on a single PC. USENIX Annual Technical Conf, p.387-401.

Wang K, Hussain A, Zuo ZQ, et al., 2017. Graspan: a single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGPLAN Not*, 52(4):389-404. https://doi.org/10.1145/3093336.3037744

Wang K, Zuo ZQ, Thorpe J, et al., 2018. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.763-782.

Wang P, Zhang K, Chen R, et al., 2014. Replication-based fault-tolerance for large-scale graph processing. 44th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.562-573. https://doi.org/10.1109/DSN.2014.58

Washio T, Motoda H, 2003. State of the art of graph-based data mining. *ACM SIGKDD Explor Newsl*, 5(1):59-68. https://doi.org/10.1145/959242.959249

Xie CN, Chen R, Guan HB, et al., 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Not*, 50(8):194-204. https://doi.org/10.1145/2858788.2688508

Xie WL, Wang GZ, Bindel D, et al., 2013. Fast iterative graph computation with block updates. *Proc VLDB Endowm*, 6(14):2014-2025. https://doi.org/10.14778/2556549.2556581

Yan D, Cheng J, Lu Y, et al., 2014. Blogel: a block-centric framework for distributed computation on real-world graphs. *Proc VLDB Endowm*, 7(14):1981-1992. https://doi.org/10.14778/2733085.2733103

Yan XF, Han JW, 2002. gSpan: graph-based substructure pattern mining. Proc IEEE Int Conf on Data Mining, p.721-724. https://doi.org/10.1109/ICDM.2002.1184038

Yan XF, Han JW, 2003. CloseGraph: mining closed frequent graph patterns. Proc ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.286-295. https://doi.org/10.1145/956750.956784

Yoo A, Chow E, Henderson K, et al., 2005. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. Proc ACM/IEEE Conf on Supercomputing, Article 25. https://doi.org/10.1109/SC.2005.4

Yuan PP, Zhang WY, Xie CF, et al., 2014. Fast iterative graph computation: a path centric approach. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.401-412.

Zaharia M, Chowdhury M, Franklin MJ, et al., 2010. Spark: cluster computing with working sets. Proc 2nd USENIX Conf on Hot Topics in Cloud Computing, Article 10.

Zhang KY, Chen R, Chen HB, 2015. NUMA-aware graph-structured analytics. *ACM SIGPLAN Not*, 50(8):183-193. https://doi.org/10.1145/2858788.2688507

Zhang MX, Wu YW, Chen K, et al., 2016. Exploring the hidden dimension in graph processing. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.285-300.

Zhang S, Wang RS, Zhang XS, 2007. Identification of overlapping community structure in complex networks using fuzzy *c*-means clustering. *Phys A*, 374(1):483-490. https://doi.org/10.1016/j.physa.2006.07.023

Zhang Y, Liao XF, Jin H, et al., 2018. CGraph: a correlations-aware approach for efficient concurrent iterative graph processing. USENIX Annual Technical Conf, p.1-12.

Zhang YH, Chen R, Chen HB, 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. Proc 26th Symp on Operating Systems Principles, p.614-630. https://doi.org/10.1145/3132747.3132777

Zhang YM, Li DS, Guo CX, et al., 2017a. CubicRing: exploiting network proximity for distributed in-memory key-value store. *IEEE/ACM Trans Netw*, 25(4):2040-2053. https://doi.org/10.1109/TNET.2017.2669215

Zhang YM, Li DS, Zhang CX, et al., 2017b. GraphA: efficient partitioning and storage for distributed graph computation. *IEEE Trans Serv Comput*, online. https://doi.org/10.1109/TSC.2017.2778737

Zhang YM, Li DS, Liu L, 2019. Leveraging glocality for fast failure recovery in distributed RAM storage. *ACM Trans Stor*, 15(1):3. https://doi.org/10.1145/3289604

Zhao Y, Yoshigoe K, Xie M, et al., 2014. LightGraph: lighten communication in distributed graph-parallel processing. IEEE Int Congress on Big Data, p.717-724. https://doi.org/10.1109/BigData.Congress.2014.106

Zhou C, Gao J, Sun B, et al., 2014. MOCgraph: scalable distributed graph processing using message online computing. *Proc VLDB Endowm*, 8(4):377-388. https://doi.org/10.14778/2735496.2735501

Zhu G, Lin X, Zhu K, et al., 2012. TreeSpan: efficiently computing similarity all-matching. Proc ACM SIGMOD Int Conf on Management of Data, p.529-540. https://doi.org/10.1145/2213836.2213896

Zhu XW, Han WT, Chen WG, 2015. GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. USENIX Annual Technical Conf, p.375-386.

Zhu XW, Chen WG, Zheng WM, et al., 2016. Gemini: a computation-centric distributed graph processing system. USENIX Symposium on Operating Systems Design and Implementation, p.301-316.

Dr. Dong-sheng LI, corresponding author of this invited review article, received the BS degree (with honors) and PhD degree (with honors) in computer science from College of Computer Science, National University of Defense Technology (NUDT), Changsha, China, in 1999 and 2005, respectively. He was awarded the prize of National Excellent Doctoral Dissertation by the Ministry of Education of China in 2008. He is now a full professor at the National Lab for Parallel and Distributed Processing, NUDT. He is a corresponding expert of *Frontiers of Information Technology & Electronic Engineering*. His research interests include parallel and distributed computing, cloud computing, and large-scale data management.