Distributed Graph Processing: Techniques and Systems

Yanfeng Zhang(✉), Qiange Wang, and Shufeng Gong

Northeastern University, Shenyang, China
zhangyf@mail.neu.edu.cn, {wangqiange,gongsf}@stumail.neu.edu.cn

Abstract. During the past 10 years, there has been a surging interest in developing distributed graph processing systems. This tutorial provides a comprehensive review of existing distributed graph processing systems. We firstly review the programming models for distributed graph processing and then summarize the common optimization techniques for improving graph execution performance, including graph partitioning methods, communication mechanisms, parallel processing models, hardware-specific optimizations, and incremental graph processing. We also present an emerging hot topic, distributed Graph Neural Networks (GNN) frameworks, and review recent progress on this topic.

Keywords: Graph processing · Distributed systems · Parallel models

1 Introduction

Graphs have been widely used to abstract the relationships between entities for many applications such as social networks, website connections, collaboration networks, and co-purchase networks. Making sense of these relational data is critical for companies and organizations to make better business decisions and even bring convenience to our daily life. Recent advances in data mining, machine learning, and data analytics have led to a flurry of graph analytic techniques. With the magnitude of graph data growing rapidly, many distributed graph processing systems running on top of a cluster of commodity PCs have been proposed to perform data analytics and data mining on these massive graphs.

Graph Properties. Graphs are a common data structure to model relationships between data items. The graph data and the computations on graphs are usually endowed with the following properties:

- **Various Graph Representations**. Graphs can be represented by adjacency matrix, adjacency list, edge list, and so on. To leverage sparsity of graphs and support efficient access, graphs can be stored with CSR (compressed sparse row) format and CSC (compressed sparse column) format.
- **Complex Relationships**. Graphs are used to model the complex relationships between entities. There might be very complex connections between vertices, which can be used to mine the potential knowledge of graphs.

Q. Chen and J. Li (Eds.): APWeb-WAIM 2020 Workshops, CCIS 1373, pp. 14–23, 2021.
https://doi.org/10.1007/978-981-16-0479-9_2

- **Power-law Characteristics**. In many real-world graphs, e.g., internet graphs, biological networks and social networks, the vertex degree distribution usually follows a power law, which implies that a small subset of the vertices connects to a large fraction of the graph.
- **Many Random Accesses**. Since a vertex can connect to any arbitrary vertex in a graph, graph algorithms usually show erratic access pattern and involve many random accesses to vertex state.
- **Iterative Computations**. Many graph mining algorithms require to traverse graph or perform information propagation, which leads to many iterations for refining graph state.

Challenges. The above summarized properties of massive graphs bring several big graph processing challenges.

- **Programming Variations.** Due to various graph representations, it is necessary to provide a general and intuitive programming interface for users to easily implement their graph algorithms.
- **Heavy Communication.** Graph data are assigned to many workers for distributed computation, each worker taking charge of a subset of vertices or edges, where the communication between workers can be very heavy.
- **Unbalanced Workload.** Due to the power-law degree distribution, it is difficult to partition the graph with evenly distributed workload.
- **Poor Locality.** Many random access including long jumps exhibit poor locality, which degrades performance.
- **Long Convergence Time.** Most of graph mining algorithms involve iterative computation. Given a big graph as input, the iterative computation may take pretty long running time for convergence.

To address these challenges, researchers have put great efforts on optimizing large-scale distributed graph processing. In this tutorial, we will briefly review the popular optimization techniques that are widely used in distributed graph processing systems and will introduce several representative systems. In addition, we will also present two emerging hot topics, incremental graph processing systems and distributed Graph Neural Networks (GNN) frameworks, and review recent works in these fields.

2 Optimization Techniques

2.1 Programming Models

As mentioned in Sect. 1, different graph representation options lead to the programming variations challenge. The graph programming models provide users unified interfaces to specify their graph algorithms and improve the usability of graph processing frameworks. Among the existing programming models, **vertex-centric** model is the most popular one. Users express their algorithms by "thinking like a vertex". Each vertex contains information about itself and all its outgoing edges, and the computation is expressed at the level of a single vertex.

The graph computation is defined as a sequence of message exchanges amongst vertices. A number of popular systems employ vertex-centric model, such as Pregel [24], PowerGraph [13], and GraphX [14]. On the other hand, X-Stream [29] leverages **edge-centric** model to obtain fully sequential access to edges (at the cost of random access to vertices), which greatly reduces the random I/O cost for querying specific vertices and is best for disk-based systems. PathGraph [47] employs **path-centric** model to maximize sequential access and minimize random access by clustering highly correlated paths together as tree based partitions. Blogel [44], Giraph++ [36], GRAPE [9] utilizes **block-centric** model that extends vertex-centric programming to blocks (i.e., a subgraph) and to exchange messages among blocks.

Besides, Pegasus [18] first proposes **Matrix-Vector Multiplication-based** programming model, which abstracts graph mining operations as a repeated matrix-vector multiplication. iMapReduce [53] relies on MapReduce interface (**MapReduce-based**) to implement a series of graph mining algorithms and provides iterative optimization for Hadoop MapReduce framework. Maiter [54] proposes **delta-based** graph computation, which abstracts the graph computation as an update accumulation process and can avoid invalid (zero-delta) updates to improve computation efficiency. SQLoop [10] leverages DBMS to implement graph iterative computations (**DBMS-based**) and extends standard SQL with efficient recursive aggregation support. Socialite [30], BigDatalog [32], and PowerLog [40] rely on Datalog language to express distributed graph algorithms (**Datalog-based**) and allow users to use very concise declarative programs to specify large-scale graph computations.

2.2 Graph Partitioning Methods

Graph partitioning is an essential yet challenging task for massive graph analysis in distributed computing. Offline methods [19] first load the complete graph into memory and then divide it into partitions, while streaming graph partitioning [27] operates online by ingesting the graph data as a stream. Graph can be partitioned by **edge-cut** [24,28], **vertex-cut** [13], or **hybrid-cut** [5] methods.

Edge-cut partitioning divides the vertices of a graph into equal-sized partitions and cuts edges, such as Metis [19] and PuLP [34]. While the vertex-cut partitioning divides edges of a graph into equal-sized clusters by making vertex replicas [7], such as SBV-Cut [20], Coordinated [13] and Neighbor Expansion (NE) [48]. The edge-cut partitioning usually results in replication of edges as well as imbalanced messages with high contention. The vertex-cut partitioning incurs high communication overhead among partitioned vertices and excessive memory consumption [5]. The hybrid-cut partitioning that integrates both edge-cut and vertex-cut address the major issues on skewed graphs, such as Ginger [5], Chunk-based partitioning [57] and Application-Driven partitioning [7].

All of the above partitioning methods are designed for synchronous distributed graph processing systems. They assume that, in each iteration, each vertex is only processed once and each edge only delivers one message. While in asynchronous frameworks, vertices can be processed at any time. The number of

updates on each vertex and the number of messages passed through each edge are not consistent. Hotness balanced partitioning (HBP) [11,12] is a novel graph partitioning method designed for prioritized iterative graph processing systems [52,54]. HBP aims to balance the hotness values of vertices in each partition, minimizes the variance between hotness distributions of each partition and the original graph, and at the same time minimizes the communication cost between partitions.

In order to partition very large graphs that are not fit in main memory, the distributed graph partitioning methods or stream-based partitioning methods can offer solutions. The distributed graph partitioning algorithms first randomly divide graph data into several parts and assign them to distributed workers. Then they exchange edges/vertices between workers iteratively based on certain schemes. For example, Sheep [25] utilizes an elimination tree to partition the large graph distributively. Spinner [25] and XtraPuLP [35] employ label propagation to move vertices or edges iteratively. Although distributed graph partitioning algorithms are able to partition extremely large graphs, they suffer from performance issues since they may require multiple iterations to refine the partition results. Stream-based methods ingest the vertices or edges as a stream, and make partitioning decisions on the fly based on partial knowledge of the graph, such as Fennel [37], HDRF [27] and Pb-HBP [12]. Because only one pass of the graph data is needed, the stream-based partitioning methods are quite efficient. However, the quality of partitioning is sensitive to the stream order, and it is not able to take advantage of parallel partitioning.

2.3 Message Passing Models

Typical vertex-centric model relies on message passing to exchange intermediate results. **Push** and **pull** are two basic message passing operations, which are suitable for different scenarios [39,42].

A number of popular systems [9,24,54] employ push based message passing model. Push-based model is flexible, since only the active vertices need to be processed. Furthermore, push-based model allow more powerful scheduling strategy to accelerate the convergence. PrIter [52] prioritizes message passing by distinguishing the important messages from the negligible messages and frequently transferring these important messages, so that the computation/update is more effective resulting in fast convergence. However, push-based model is not suitable for parallel processing, since the single-read-multi-write scheme will cause write-write conflict and may incur atomic overhead.

In contrast, PowerGraph [13] and Pregel+ [45] leverage pull-based model. With pull-based model, a vertex pull updates from its in-neighbours. Due to the multi-read-single-write updating scheme, pull-based can be parallel without atomic operations. However, pull-based model cannot achieve selective processing, all vertices have to be accessed no matter active or not. Therefore, redundant computation is inevitable when the active set is small.

Existing work demonstrates that, the size of active nodes set might be different in different stages, which implies that different stages may prefer different

message passing strategies [33,39,57]. Ligra [33] proposed **hybrid push-pull** model for shared-memory system, which automatically switches between push and pull based on the size of active set, to reduce both redundant computation and atomic overhead. Gemini [57] extends the hybrid model to distributed environment and adopts similar approach. On the other hand, reducing redundant computation and atomic operation overhead is also critical for efficiently executing graph algorithm on massive parallel hardware, e.g., SEPgraph [39] extends hybrid push-pull model on GPUs and supports automatic push-pull model switch.

2.4 Parallel Processing Models

Graph computation usually exhibits iterative computing nature, where input data is computed iteratively until a convergence condition is reached. Synchronous parallel model requires all vertex updates completed before starting next iteration, while asynchronous parallel model does not have this requirement. With **synchronous** parallel processing [9,13,24,57], all workers keep the same pace The messages can be packed before being sent in order to reduce communication overhead (BSP model). Furthermore, synchronous programs are easy to write, tune and debug. However, the slowest worker will become the straggler and dominate the run time. With **asynchronous** parallel processing, workers do not need to keep consistent pace. Fast workers can perform more computations to accelerate the convergence [3,43,54]. Asynchronous parallel processing may incur irregular and redundant communications [40,43], and may lead to stale computation [8].

In order to avoid the shortcomings of both synchornous and asynchronous parallel models, **sync-async hybrid** parallel processing systems have emerged in the pass few years. PowerSwitch [43] proposes Hsync, which can automatically switch between synchronous model and asynchronous model. In PowerSwitch, in the same time period all workers in the cluster universally use the sync model or the async model. Grape+ [8] proposes Adaptive Asynchronous Parallel (AAP) processing, by monitoring the incoming message rate. AAP adaptively tunes the stale delay to achieve different parallel model (BSP, Staleness Parallel Processing, Asynchronous Parallel Processing). Different from PowerSwitch, in Grape+, the workers can use different parallel models, and each worker determines to choose its own parallel model. In addition, Grape+ is based on block-centric model, which can help combining messages and reduce communication. PowerLog [40] proposes that asynchronous parallel model under different message passing rates can result in different performance. PowerLog proposes a unified sync-async processing model by monitoring the local update frequency and can adaptively adjust the asynchronous degree to achieve better performance.

2.5 Hardware-Specific Optimizations

Hardware-specific optimizations are essential and emerging to provide the performance improvement significantly beyond those pure software optimizations

can offer. GPU is adopted to pursue high performance of graph processing due to its data parallel capability. A number of graph processing systems with **GPUs** have been proposed for high-performance graph processing, such as Medusa [56], Gunrock [41], and SEP-Graph [39]. FPGA is an integrated circuit that enables designers to repeatedly configure digital logic in the fields after manufacturing, also called field-programmable. A number of existing works integrate **FPGAs** to support high-performance graph processing, such as CyGraph [2] and ForeGraph [6]. Application-specific integrated circuit (ASIC) is usually fabricated on a wafer composed of silicon or other semiconductor materials that are customized for a particular use. Researchers follow vertex-centric model and form the **ASIC** circuit to support fast graph processing, e.g., Graphicionado [15].

On the other hand, graph processing can use disks, flashes or other external storage devices to store extremely large scale graphs. A number of studies aim to reduce the transmission cost of I/Os to improve performance. GraphChi [21] and GridGraph [58] are typical **out-of-core** solutions that reorganize the file storage structure of graph data to realize sequential disk file accesses and can process large graph in a single machine. There exist several studies using DRAM and **SSDs** to build hybrid graph system which stores vertex state in memory and edge lists on SSDs, such as MaiterStore [4], FlashGraph [55], and SMaiter [22]. There are also other studies that are optimized for communication networks. GraphRex [50] provides specific optimizations for high-performance network and under cross-rack cluster environment.

3 Emerging Applications

3.1 Incremental Graph Processing

With the continuously evolving nature of real-life graphs, the results of graph mining become stale and obsolete over time. Incremental processing graph [26, 31,38,51] is a promising approach for refreshing graph mining results. It utilizes previously saved states to avoid re-computation from scratch.

In order to process graph incrementally, i^{2}MapReduce [51] uses MapReduce Bipartite Graph (MRBGraph) to model the data flow in MapReduce. Each vertex in the map task represents an individual map function. For the input delta graph, which contains the added/deleted edges and vertices, the i^{2}MapReduce engine invokes the Map function for every record in the delta input. The Map function outputs the Delta MRBGraph that only contains the changes to the MRBGraph. Then i^{2}MapReduce merges the preserved MRBGraph and the Delta MRBGraph, and obtains the updated MRBGraph. For each affected key in the updated MRBGraph, the merged list of values will be used to invoke the Reduce function to generate the updated final results.

Tornado [31] is an incremental iterative graph processing system that is built on top of Storm. Tornado contains a main loop and several branch loops. The main loop continuously gathers incoming data and approximates the results at the current instant, while the branch loops perform iterations over the snapshot that is taken when a result query is required. KickStarter [38] builds a set of

dependency trees based on the dependency relationship between vertices. When an edge is deleted, KickStarter identifies the set of vertices impacted by the deleted edge. This can be done simply by finding the subtree rooted at the target vertex of the deleted edge. Then it resets the value of the impacted vertices and rebuilds the dependency tree by recomputing the values of the impacted vertices. GraphBolt [26] proposes a dependency graph for tracking dependency relationship between vertices. Vertices in the dependency graph maintain their intermediate values that are produced during the iterative computations. Edges in the dependency graph capture dependencies among intermediate values. As the graph changes, GraphBolt corrects the intermediate vertex states iteration-by-iteration according to the dependency graph.

3.2 Distributed GNN Training

Graph Neural Networks (GNNs) have been emerging as powerful learning tools for graph data. However, it is challenging to train a GNN for real-world large-scale graphs. Most of the existing popular deep learning frameworks run a single machine, which cannot offer much scalability. Therefore, building a scalable GNN training system for large-scale graphs is desirable [16,17,23,46,49].

A common GNN task contains the forward and backward propagation in standard deep learning and the iterative graph propagation in graph mining algorithms. These two distinct computing styles make it difficult to build high performance GNN systems in distributed environment. Neither existing graph processing systems nor deep learning systems can support GNN training well. An intuitive approach is completely partitioning the graph data to avoid communication between subgraphs (subtasks) and leverage the parameter server's data parallel model. Aligraph [46] relies on distributed Tensorflow's parameter server architecture. The graph data are stored in the server side. When training a K-layer GNN task, each worker pulls the training graph data of one batch (including its k-hop neighbours and their features) from the PS-server and locally performs the computation. Euler [1], AGL [49], PSGraph [17] adopt similar method. PSGraph builds the GNN training system on top of Hadoop and Spark ecosystems, which makes it easier to scale.

Acknowledgement. This work was supported by National Key R&D Program of China (2018YFB1003404), National Natural Science Foundation of China (62072082, 61672141, and U1811261) and Fundamental Research Funds for the Central Universities (N181605017 and N181604016), and Key R&D Program of Liaoning Province (2020JH 2/10100037).

References

1. Euler 2.0 (2020). https://github.com/alibaba/euler
2. Attia, O.G., Johnson, T., Townsend, K., Jones, P., Zambreno, J.: CyGraph: a reconfigurable architecture for parallel breadth-first search. Proc. IPDPS **2014**, 228–235 (2014)

3. Ben-Nun, T., Sutton, M., Pai, S., Pingali, K.: Groute: an asynchronous multi-GPU programming model for irregular computations. In: ACM SIGPLAN Notices, vol. 52, no. 8, pp. 235–248 (2017)
4. Chang, D., Zhang, Y., Yu, G.: MaiterStore: a hot-aware, high-performance key-value store for graph processing. In: Han, W.-S., Lee, M.L., Muliantara, A., Sanjaya, N.A., Thalheim, B., Zhou, S. (eds.) DASFAA 2014. LNCS, vol. 8505, pp. 117–131. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43984-5_9
5. Chen, R., Shi, J., Chen, Y., Zang, B., Guan, H., Chen, H.: PowerLyra: differentiated graph computation and partitioning on skewed graphs. ACM Trans. Parallel Comput. (TOPC) **5**(3), 1–39 (2019)
6. Dai, G., Huang, T., Chi, Y., Xu, N., Wang, Y., Yang, H.: ForeGraph: exploring large-scale graph processing on multi-FPGA architecture. Proc. FPGA **2017**, 217–226 (2017)
7. Fan, W., et al.: Application driven graph partitioning. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020), pp. 1765–1779 (2020)
8. Fan, W., et al.: Adaptive asynchronous parallelization of graph algorithms. ACM Trans. Database Syst. (TODS) **45**(2), 1–45 (2020)
9. Fan, W., et al.: Parallelizing sequential graph computations. ACM Trans. Database Syst. (TODS) **43**(4), 1–39 (2018)
10. Floratos, S., Zhang, Y., Yuan, Y., Lee, R., Zhang, X.: SQLoop: high performance iterative processing in data management. In: Proceedings of ICDCS 2018, pp. 1039–1051 (2018)
11. Gong, S., Zhang, Y., Yu, G.: Accelerating large-scale prioritized graph computations by hotness balanced partition (online). IEEE Trans. Parallel Distrib. Syst. **32**, 746–759 (2020)
12. Gong, S., Zhang, Y., Yu, G.: HBP: hotness balanced partition for prioritized iterative graph computations. In: Proceedings of the 36th International Conference on Data Engineering (ICDE 2020), pp. 1942–1945 (2020)
13. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI 2012, pp. 17–30 (2012)
14. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: Proceedings of OSDI 2014, pp. 599–613 (2014)
15. Ham, T.J., Wu, L., Sundaram, N., Satish, N., Martonosi, M.: Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2016), pp. 1–13 (2016)
16. Jia, Z., Lin, S., Gao, M., Zaharia, M., Aiken, A.: Improving the accuracy, scalability, and performance of graph neural networks with ROC. In: Proceedings of Machine Learning and Systems (MLSys 2020), pp. 187–198 (2020)
17. Jiang, J., et al.: PSGraph: how Tencent trains extremely large-scale graphs with spark? In: Proceedings of ICDE 2020, pp. 1549–1557 (2020)
18. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system implementation and observations. In: Proceedings of ICDM 2009, pp. 229–238 (2009)
19. Karypis, G., Kumar, V.: METIS: a software package for partitioning unstructured graphs. Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4(0) (1998)

20. Kim, M., Candan, K.S.: SBV-Cut: vertex-cut based graph partitioning using structural balance vertices. Data Knowl. Eng. **72**, 285–303 (2012)
21. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of OSDI 2012, pp. 31–46 (2012)
22. Li, J., Zhang, Y., Gong, S., Yu, G., Gao, L.: Streamlined asynchronous graph processing framework. J. Softw. **3**, 528–544 (2018)
23. Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., Dai, Y.: NeuGraph: parallel deep neural network computation on large graphs. In: Proceedings of USENIX ATC 2019, pp. 443–458 (2019)
24. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: Proceedings of SIGMOD 2010, pp. 135–146 (2010)
25. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. Proc. VLDB Endow. **8**(12), 1478–1489 (2015)
26. Mariappan, M., Vora, K.: GraphBolt: dependency-driven synchronous processing of streaming graphs. In: Proceedings of EuroSys 2019, pp. 1–16 (2019)
27. Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., Iacoboni, G.: HDRF: stream-based partitioning for power-law graphs. In: Proceedings of CIKM 2015, pp. 243–252 (2015)
28. Reittu, H., Norros, I., Rty, T., Bolla, M., Bazsó, F.: Regular decomposition of large graphs: foundation of a sampling approach to stochastic block model fitting. Data Sci. Eng. **4**(1), 44–60 (2019)
29. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-Stream: edge-centric graph processing using streaming partitions. In: Proceedings of SOSP 2013, pp. 472–488 (2013)
30. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed socialite: a datalog-based language for large-scale graph analysis. Proc. VLDB Endow. **6**(14), 1906–1917 (2013)
31. Shi, X., Cui, B., Shao, Y., Tong, Y.: Tornado: a system for real-time iterative analysis over evolving data. In: Proceedings of SIGMOD 2016, pp. 417–430 (2016)
32. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big data analytics with datalog queries on spark. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD 2016), pp. 1135–1149 (2016)
33. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of PPoPP 2013, pp. 135–146 (2013)
34. Slota, G.M., Madduri, K., Rajamanickam, S.: PuLP: scalable multi-objective multi-constraint partitioning for small-world networks. In: Proceedings of 2014 IEEE International Conference on Big Data, pp. 481–490 (2014)
35. Slota, G.M., Rajamanickam, S., Devine, K., Madduri, K.: Partitioning trillion-edge graphs in minutes. In: Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017), pp. 646–655. IEEE (2017)
36. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". Proc. VLDB Endow. **7**(3), 193–204 (2013)
37. Tsourakakis, C., Gkantsidis, C., Radunovic, B., Vojnovic, M.: FENNEL: streaming graph partitioning for massive scale graphs. In: Proceedings of WSDM 2014, pp. 333–342 (2014)
38. Vora, K., Gupta, R., Xu, G.: KickStarter: fast and accurate computations on streaming graphs via trimmed approximations. In: Proceedings of ASPLOS 2017, pp. 237–251 (2017)
39. Wang, H., Geng, L., Lee, R., Hou, K., Zhang, Y., Zhang, X.: SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In: Proceedings of PPoPP 2019, pp. 38–52 (2019)

40. Wang, Q., et al.: Automating incremental and asynchronous evaluation for recursive aggregate data processing. In: Proceedings of SIGMOD 2020, pp. 2439–2454 (2020)
41. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of PPoPP 2016, pp. 1–12 (2016)
42. Wang, Z., Gu, Y., Bao, Y., Yu, G., Yu, J.X.: Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In: Proceedings of SIGMOD 2016, pp. 479–494 (2016)
43. Xie, C., Chen, R., Guan, H., Zang, B., Chen, H.: SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In: ACM SIGPLAN Notices, vol. 50, no. 8, pp. 194–204 (2015)
44. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. Proc. VLDB Endow. **7**(14), 1981–1992 (2014)
45. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of WWW 2015, WWW 2015, pp. 1307–1317 (2015)
46. Yang, H.: AliGraph: a comprehensive graph neural network platform. In: Proceedings of KDD 2019, pp. 3165–3166 (2019)
47. Yuan, P., Xie, C., Liu, L., Jin, H.: PathGraph: a path centric graph processing system. IEEE Trans. Parallel Distrib. Syst. **27**(10), 2998–3012 (2016)
48. Zhang, C., Wei, F., Liu, Q., Tang, Z.G., Li, Z.: Graph edge partitioning via neighborhood heuristic. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2017), pp. 605–614 (2017)
49. Zhang, D., et al.: AGL: a scalable system for industrial-purpose graph machine learning. arXiv preprint arXiv:2003.02454 (2020)
50. Zhang, Q., et al.: Optimizing declarative graph queries at large scale. In: Proceedings of SIGMOD 2019, pp. 1411–1428 (2019)
51. Zhang, Y., Chen, S., Wang, Q., Yu, G.: i^{2}MapReduce: incremental mapreduce for mining evolving big data. IEEE Trans. Knowl. Data Eng. **27**(7), 1906–1919 (2015)
52. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: a distributed framework for prioritized iterative computations. In: Proceedings of SOCC 2011, pp. 1–14 (2011)
53. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012)
54. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Trans. Parallel Distrib. Syst. **25**(8), 2091–2100 (2013)
55. Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C.E., Szalay, A.S.: FlashGraph: processing billion-node graphs on an array of commodity SSDs. In: Proceedings of FAST 2015, pp. 45–58 (2015)
56. Zhong, J., He, B.: Medusa: a parallel graph processing system on graphics processors. ACM SIGMOD Rec. **43**(2), 35–40 (2014)
57. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: a computation-centric distributed graph processing system. In: Proceedings of OSDI 2016, pp. 301–316 (2016)
58. Zhu, X., Han, W., Chen, W.: GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of USENIX ATC 2015, pp. 375–386 (2015)