Fast and Optimized Graph Computing System

Cheng Wang
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
cheng2@cs.hku.hk

Jianyu Jiang
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
jyjiang@cs.hku.hk

ABSTRACT

With the growing numbers of data, people desire a fast and scalable framework for computations on them. However, general purpose computation frameworks (e.g., Spark, MapReduce) are not suitable for graph computation algorithms such as PageRank and ConnectedComponent. Theses frameworks generally generate prohibited amounts of network traffic, so they are not scalable to datasets with billions or even trillion number of records.

This paper revisits several graph computing systems and summarizes the similarity and advantages of them. We first give a brief introduction of some several characteristics of graph computing systems and show how all these graph computing system make their design choice. We also analyze how these systems make the efforts to make computation scalable. We have done extensive evaluations on several graph computing framework such as PowerGraph, Spark GraphX and Gigraph.

1. INTRODUCTION

General big-data computing frameworks (e.g., Spark [24], MapReduce [7]) provide flexible and efficient programming primitives for most algorithms. These frameworks split data into multiple partitions and run tasks on these task independently. When necessary, shuffles happen to resolve data dependencies. However, these framework can result in a great amount of network traffic for graph computing algorithms such as PageRank. Even worse, most realistic graphs follows the power law, so joining two nodes may incur a one-to-all communications.

To tackle this problem, specified graph computing framework such as Pregel [15], GraphLab [14], PowerGraph [8], GraphX [21] ,GraphChi [12] and so on [13, 4]. These graph computing frameworks adopt a vertex-edge computing model, and computations are modeled as vertex and edge updates. For example, Pregel adopts the bulk synchronous parallel (BSP) model, so each vertex updates its value according to the edge and vertices (current vertex and neighboring vertices) values, and the update algorithm.

However, the BSP model is found to be inefficient as the synchronous update mechanism is always necessary for algorithms such as PageRank to converge [14, 8]. Moreover, multiple partition schemes (e.g., vertex-centric [15], edge-centric [18] and 3D partition [26]). The connectivity of graph and the partition scheme can greatly affect the communication cost, and further affect performance. PowerGraph [8] is a advanced version of GraphLab which takes the power law distribution of edges.

We have observed that most graph computing system optimize performance by scheduling, new computing abstractions and partition schemes. The optimizations may come from computation locality (e.g., cpu-memory, data-partition), selective scheduling and so on.

In this paper, we first analyze the basic components of a graph computing system. Then, we give a survey of some popular graph computing system such as Pregel, GraphLab, PowerGraph, GraphChi, GraphX and so on. We show how these systems optimize their performance in terms of network traffic, computing models. We further evaluate the performance of several graph computing systems and analyze their performance. Our evaluations show that PowerGraph has the best performance in practice, as it has optimized to work for graph with power law distribution.

The structure of this paper is as follows: §2 gives the definition of graph computing system and its computing procedure. §3 introduce the components of graph computing systems. §4 gives a survey and analysis of several graph computing system. §5 evaluated system performance. §6 gives the conclusion of the paper.

2. PRELIMINARY

We first introduce some definition of graph computing. Suppose there is a graph G = (V, E), V and E the vertex and edges of the graph. Each vertex contains a state set $(s_1, s_2, ..., s_n)$, and a set of neighboring vertices $v_{x1}, v_{x2}, ..., vxn$. Each edge contains a state set (including the weight of edges) of $(s_1, s_2, ..., s_n, w_i)$.

Initially, the graph G is stored in a database or files of HDFS. The system first **load** data from storages. Each worker loads a subset of vertices and edges of the graph G into memory

according to the **partition scheme** and initializes the values. Then, each worker conducts iterative computations to produce an output from vertex and edge values according to the computing models. The output is used for updating values in the second phase of computations. After that, the updated values will be used in the new iteration. When the computation converges, the system stops and the state of the graph will be output to persistent storages.

3. GRAPH COMPUTING SYSTEM

3.1 Computation Abstraction

The computation abstraction is the core of a graph computing system. The most popular computation abstraction is BSP model and GAS model. The advantage of graph computing systems lies in flexibility and performance.

For flexibility, the system should be able to compute on various graph, even the topology of graph may be changing along with time. Traditional MapReduce framework computes data in functional-like APIs, and stateful computations are hard in practice. TODO.

For performance, as graph computing generally generate a tremendous amount of network traffics for some operations, it is highly desirable to reduce it. MapReduce framework needs to transfer all states from previous computation to the next one, which causes high computation costs. With graph computing, only a portion of states needs to be transferred.

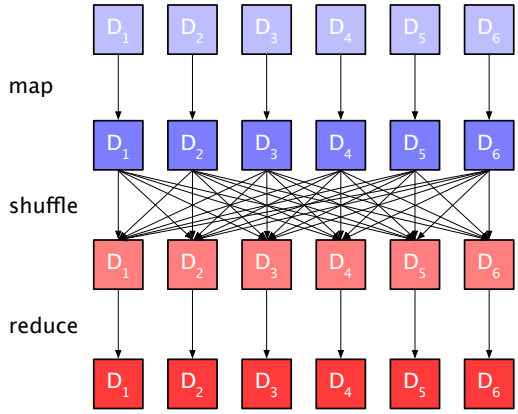

Figure 1: MapReduce

3.1.1 MapReduce

MapReduce [7] computes data in map and reduce operations. In map, a user-defined function on each data record is executed independently. After map, results are shuffled and redistributed into different partitions for reduce. In reduce, it read output from previous map and executes aggregations for specific subsets of data. MapReduce is a functional computation model, so no states are stored across computations. Therefore, states of data are stored in data so all states in previous computations are necessary for computations in next one. Figure 1 shows the idea of MapReduce.

3.1.2 Bulk Synchronous Parallel Model

The BSP model assumes that computation is done on multiple steps. A bulk contains some subset of data, and a computation is done on these subset of data. After computations, the output is used for synchronization.

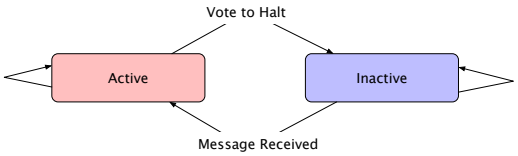

Figure 2: State

For graph computing, Pregel makes the BSP model more specific. Each vertex has two states: active and inactive, and computation will only be conducted on active vertices. Figure 2 shows the state machine of a vertex. Computations are done on multiple supersteps. In the first superstep, all vertices is active and they will vote to halt and convert to inactive. A vertex will be activate when a message for the vertex is received.

In each superstep, a vertex get all messages sent to it and computes result according to the current state of the vertex and sends messages to its neighbors. Figure **??** shows the workflow of BSP.

A Combiner and Aggregator are used for transferring data for message passing. The Combiner can combines states in each partitions to reduce network traffics. For example, both vertex b and c send a same message to vertex a, then this message can be combined into one. On the other hand, Aggregator is used for aggregating output from multiple vertex and even multiple iterations.

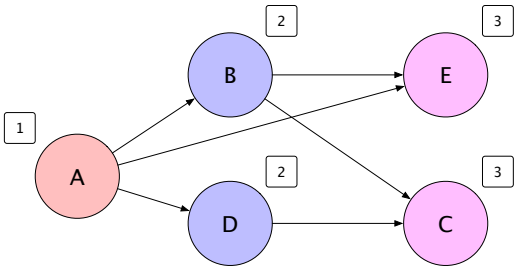

Figure 3: GAS

3.1.3 GAS Model

Gather, apply and scatter model (GAS) is more restricted than the BSP model. Compared with BSP, a vertex in GAS can only interact with neighboring vertices.

In **gather** phase, a vertex gathers information from all adjacent vertices, and set a aggregate function for all the gathered information. Then, it **apply** the function and updates values of vertices and edges. At last, in **scatter**, it can activate it neighboring vertices with information.

Compared with BSP model, a node in GAS only interact with neighboring vertices. Although it hurts expresses of computations, it also enable multiple optimization techniques. For example, vertices can be partitioned and computes in multiple steps (e.g., using the coring algorithm).

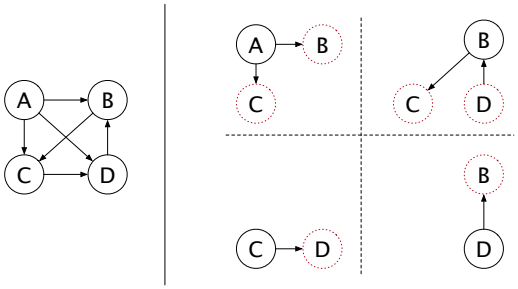

Figure 4: Edge Cut

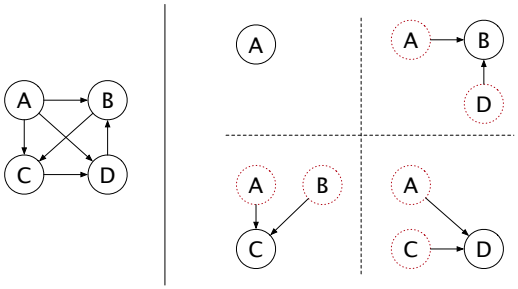

Figure 5: Vertex Cut

Therefore, messages passing and computation can be done at the same time, which greatly improve performance. Furthermore, a flexible consistency model can be adopted. For example, a $VertexConsistency$ only guarantees consistency of read-writes of vertices. In this cases, races of edges read/writes can happen and no locking or synchronization are needed, which improve performance.

3.2 Partition Scheme

The way to partition the graph G can greatly affect performance. The simplest way to partition data is by its vertex id or edge id. However, this approach incurs too much network traffic. Therefore, multiple partition schemes have been proposed including **Vertex Cut**, **Edge Cut** and **3D Cut**. This algorithm is enough for randomly generated graphs, but for real-world graphs that follow the power law, a 1D partitioner usually leads to considerable skewness.

Vertex Cut Vertex cut partition vertices into multiple random partitions. Figure 5 shows the idea of Vertex Cut.

Edge Cut As the Vertex Cut is not suitable for real-life graph which follows power law, some work [18] propose to partition data based on edges. In Edge Cut, edges are put into multiple partitions equally, synchronization of vertices in different nodes incurs network communications. Figure 4 shows the idea of Edge Cut.

3D Cut For more, PowerLyra [3] use a heristic to combine them. Cube proposes a new 3D partition of graph, which consider dimensions of data. Cube [26] considers the hidden dimensions of vertex partition. For Machine Learning and Data Mining (MLDM) algorithms, values of vertex may be linked to values of different vertices. Therefore, different value of a vertex can be partitioned into different nodes for improved computations. Figure 6 shows the idea of 3D Cut.

Dynamic Repartitioning Different processes may has imbalance workload, this can hurt performance as some processes are waiting for result from others. A Dynamic Repartitioning move data from one node to another so that workload is balanced across nodes. However, this technique can incur extra network traffics. Moreover, performance of this technique is not stable, as workload of nodes in next iterations may change, causing imbalance workloads.

Mirroring To reduce network traffic, Mirroring creates mirroring vertices (or edges) for a cut edge. Therefore, access of neighboring vertices does not incur synchronous network messages.

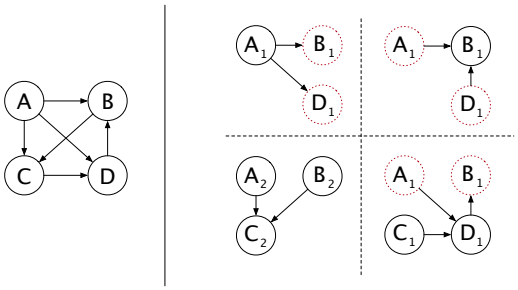

Figure 6: 3D Cut

4. SURVEY OF EXISTING SYSTEMS

In this section, we will give a short survey of existing graph computing systems. We will explain the techniques adopted in these system and show some unique techniques for them.

4.1 Pregel

Pregel [15] is a C++ implementation of Google's graph computing system. Pregel adopts the BSP computation abstract and uses the simple vertex cut partition scheme. A $compute()$ function should be programmed for graph computations on a vertex v. It accepts the messages from previous supersteps and sends messages to next supersteps.

Pregel uses simple vertex cut for graph partition, this is practical choice as a vertex in Pregel can send messages to arbitrary vertices in a step. Therefore, communication patterns should be known for a better partition of graph, which may not be possible in practice. Pregel provides abstraction for changing topologies of graph. It adopts two mechanisms: partial ordering and handlers, to guarantee determinism of computation for changing of graphs.

In practice, a open-sourced implementation of google's Pregel is Giraph [1]. It provides multi-threading programming and memory optimization to get optimal performance.

4.2 GraphLab / PowerGraph

GraphLab [14] chooses GAS computation model for graph computing. With GAS, GraphLab open several optimizations for performances. For example, it can separate computations into multiple steps, and computations and message passing can be working in the same time.

GraphLab also supports two kinds of execution: asynchronous execution and synchronous execution. For asynchronous execution, data is committed immediately without communications to the neighboring vertices. The change of value

will only be visible in the next step of computation. Although this approach can result in inconsistency of data, it greatly improve performances for algorithms such as PageRank. These algorithms can incur large amounts of network traffics, but values in consecutive iterations are really close.

For synchronous execution, it add a barrier in each iteration for data exchanges by message passing. This barrier guarantee consistency of values across iterations, but the barrier can greatly degrade performance.

PowerGraph [8] is a advance version of GraphLab. It adopts vertex cut for graph partition and uses mirroring of vertices to reduces network traffics for graph with power law distribution. A network message are needed for data consistency of vertices across nodes.

4.3 GraphX

GraphX [21] is a graph commuting framework based on Spark. It introduces two RDD: VertexRDD and EdgeRDD for vertex and edge. representations. Although GraphX is based on a general-purpose dataflow commuting system, it adopted several optimizations for graph computing.

GraphX adopts a Vertex Mirroring, Multicast Join and Incremental View Maintenance for network traffic optimizations. Incremental View Maintenance uses to reduce unnecessary move of unchanged data. As reported in the paper [21], the performance of GraphX is close to GraphLab, an C++ implementation system.

4.4 GraphChi

GraphChi [12] is a graph computing system on a single machine. It adopts Out-of-core computation for reducing accesses of second storages. It also uses GAS model and proposes a selective scheduling scheme. The selective scheduling scheme prioritizes computation with significant values changing. The idea of this technique is that most computations are not necessary for convergences of the whole graph, most of the graph changes only a little or even does not change at all. Therefore, this technique is good for convergences of the whole graph.

4.5 Other Graph Computing system

There are many other graph computing systems [25, 27, 11, 22, 16] aiming to improving performance of graph computing system. Some work [23, 20, 2, 5] aims to scale computations on graph up to trillions of edges. In practice, the graph size is up to trillions, so it is emergent to handle such a large graph.

Some systems [17, 6] propose new dataflow Computing models, and it can process graph in streaming style. These systems compute data in a iterative way, so difference across iterations can be observed for optimizations.

In practice, graph may change with time. Therefore, it is emergent to process temporal graph. Several systems [19, 9, 4, 10] are aimming to tackle this problem. They either adopt a adaptive snapshot or propose new computation model.

5. EVALUATION

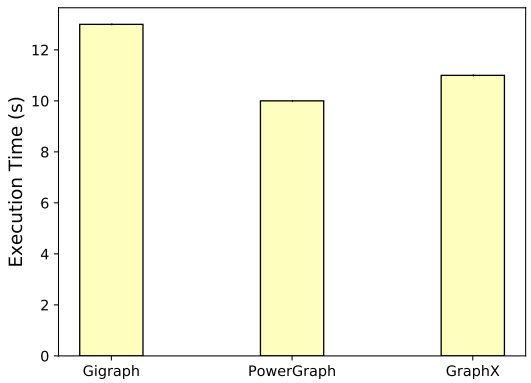

Figure 7:

The section shows the evaluation of several graph computing systems: GraphX, GraphLab (PowerGraph) and Giraph. We conduct the evaluations of these systems in a cluster with 4 machines. Each machine is equipped with a Xeon CPU with 24 hyper cores. We evaluate PageRank with 2 datasets, the datasets we used are showed in Table 1.

Dataset Name	Vertex	Edges
Youtube Network	10k	10m
Twitter Network	1k	1m

Table 1: Dataset.

Our evaluations mainly focus on performance of these system and evaluates Gigraph, SparkX and PowerGraph. Our evaluations mainly answer the following questions:

§5.1: How much time needed for each framework to converge a PageRank?
§5.2: Are these frameworks scalable to large graphs or large number of machines?
§5.3: Are the techniques in each framework effective to provide fast computations?

5.1 Computation Time

We first run the PageRank program with two datasets and compare the time it takes to converge. Figure 7 and Figure 8 shows the result. We used the asynchronous computation mode for PowerGraph.

Among two datasets for PageRank, PowerGrapb has the best performance, while GraphX is a little bit slower, Gigraph performs the worst. PowerGraph is a C++ implementation with asynchronous execution, which results in the best performance.

5.2 Scalability

5.2.1 Scalability of Workers

A good graph computing system should make better use of computing resources. Therefore, when the number of computing resources increases, the computing time should be shorten.

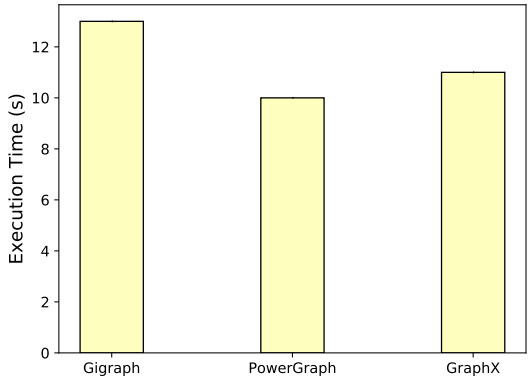

Figure 8:

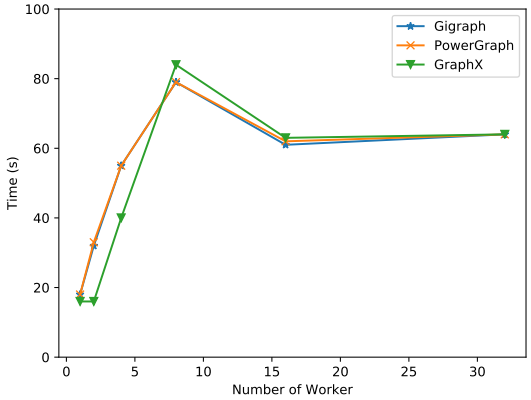

Figure 9:

We run all systems with different numbers of workers using the same dataset, and calculate the time for it to converge. Figure 9 shows the result. As the result shows, it takes less time when the number of workers increases. However, the time does not decrease linearly as it takes much larger costs for task scheduling and message transferring.

5.2.2 *Scalability of Datasets*

In practice, these graph computing system should be able to processing graphs with large numbers of vertices and edges. The scale of graph in practice is growing, and the graph size is billion and even trillion in industrial applications. Therefore, it is highly desirable to scale to large graphs.

We run all systems with different sizes of datasets using the same number of workers. Figure **??** shows the time for them to converge the result. As the result shows, all of these system have good scalability. Some of them increase sub-linearly while PowerGraph has the best scalability.

5.3 Technique Effectiveness

In the section, we show the effectiveness of techniques in different systems. For the short of time, we only evaluated

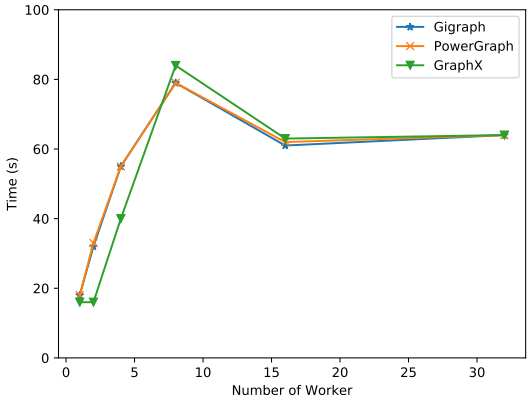

Figure 10:

execution mode	time
Synchronous Execution	10s
Asynchronous Execution	6s

Table 2:

the effectiveness of asynchronous execution in this section.

Table 2 shows the result of running with and without asynchronous execution. The result shows that it takes less time with asynchronous execution. With asynchronous execution, it takes less time for synchronization, which results in much better performance.

6. CONCLUSION

In this paper, we have given a short summery of some popular graph computing frameworks. We have conducted extensive evaluations for these frameworks for analyze the performance gained in each techniques. Understanding the effectiveness of each techniques and the design choices make us better understand how to design a practical graph computing system.

7. REFERENCES

[1] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[2] F. Checconi and F. Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 425–434. IEEE, 2014.

[3] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[4] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM*

european conference on Computer Systems, pages 85–98. ACM, 2012.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[6] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment*, 9(12):1137–1148, 2016.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[9] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[10] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[11] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[12] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[14] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

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

[16] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):14, 2015.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[19] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *arXiv preprint arXiv:1309.1049*, 2013.

[20] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. G ra m: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[21] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[22] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 227–238. ACM, 2014.

[23] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big graph analytics systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2241–2243. ACM, 2016.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[25] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.

[26] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[27] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.