IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 15, 2018, accepted June 7, 2018, date of publication June 18, 2018, date of current version July 12, 2018.

*Digital Object Identifier 10.1109/ACCESS.2018.2848291*

# GraphDuo: A Dual-Model Graph Processing Framework

**XINHUI TIAN[1,2] AND JIANFENG ZHAN[1,2], (Member, IEEE)**
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
[2]University of Chinese Academy of Sciences, Beijing 100049, China

Corresponding author: Jianfeng Zhan (zhanjianfeng@ict.ac.cn)

This work was supported by the National Key Research and Development Plan of China under Grants 2016YFB1000600 and 2016YFB1000601.

**ABSTRACT** Algorithms for large-scale natural graph processing can be categorized into two types based on their value propagation behaviors: the unidirectional value propagation algorithms and the bidirectional value propagation algorithms. The graph computation in different types exhibits different properties on how vertices interact with their neighbors and has different requirements on system design. Current distributed graph processing systems usually try to support both types in one general-purpose computing model, which can result in suboptimial performance, high communication overhead, and high resource consumption. In this paper, we propose GraphDuo, a new Spark-based graph processing framework that provides two efficient computing models for unidirectional and bidirectional value propagation algorithms, respectively. Combined with a degree-based graph partitioning scheme, a locality-aware graph layout, and other optimization techniques, GraphDuo achieves low computation cost, low communication overhead, small memory footprint, and good communication balance for two types of algorithms. According to the experimental results, GraphDuo can outperform GraphX from 1.76× to 6.56× and from 1.11× to 1.26× for two types of algorithms, respectively, with much less memory consumption and communication cost. The source code of GraphDuo is publicly available from http://prof.ict.ac.cn/GraphDuo.

**INDEX TERMS** Data processing, distributed processing, graph theory, large-scale systems.

## I. INTRODUCTION

Graphs are ubiquitous in many domains, including social networks, recommendation systems, natural language processing, and many scientific researches. The scales of these graphs are often very large. The task to efficiently process massive graphs has attracted a lot of attention and research effort in recent years, which drives the development of lots of specialized distributed graph processing systems [1]–[5]. These frameworks usually follow the *"think like a vertex"* philosophy by providing users a vertex-centric programming model, coupled with an iterative execution engine to process vertices in parallel.

In these frameworks, graph algorithms are coded as vertex programs, which define algorithm behaviors including how each vertex computes new value based on its neighbors, and how it propagates its value along edges. Especially,different algorithms can exhibit varied value propagation behaviors. In many natural graph algorithms, such as PageRank and Single-Source Shortest Path (SSSP), vertices propagate vertex values along only one direction [6], [7]. In other graph algorithms, such as Connected Components and SVD [8], vertices need to accumulate values from all neighbors to compute new values. In those algorithms, value propagation is performed along both edge directions. For ease of discussion, we name algorithms with one consistent value propagation direction as the unidirectional value propagation (UVP) algorithms, and the others as the bidirectional value propagation (BVP) algorithms. For UVP algorithms, only unidirectional communication is required during the computation. For BVP algorithms, bidirectional communication is required. Therefore, algorithms of different types have different communication patterns and neighbor dependencies, which raise different requirements for system design.

On the other hand, properties of graph structures also exhibit great diversity. The vertex degree distributions of different graph datasets vary widely in different domain [9]. Especially, in natural graphs from real world, the vertex degrees follow a power-law distribution, in which a small faction of vertices connect with lots of neighbors [3]. Great challenges are also imposed to distributed graph processing


2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

systems on how to efficiently process those skewed graphs [10], [11].

Existing systems usually employ an "*one-size-fits-all*" design to handle diverse graph algorithms. Those systems often provide computing models and partitioning schemes primarily designed for one type of algorithms, and treat the other as a special case. Such design can results in suboptimial performance and large memory consumption for the other type of algorithms. For example, Pregel [1] performs vertex-centric computation and makes resources locally accessiable to ensure low computation latency by evenly distributing vertices with all their out-neighbors among machines. Pregel restricts the value propagation direction along outgoing edges, and natively support UVP algorithms. For BVP algorithms, Pregel needs to add reverse replicas for each edge, which can lead to large memory footprint and high communication overhead. GraphLab [2] also employs a vertex-centric computing model. However, it blindly creates replicas for edges and vertices spanning machines, which can result in large memory footprint for both UVP and BVP algorithms. Moreover, both Pregel and GraphLab lack efficient support for high-degree vertices (i.e., vertices with a large number of neighbors), which results in imbalanced communication for skewed graphs. In constrast, PowerGraph [3], PowerLyra [6], and GraphX [5] evenly partition edges among machines, and perform multi-step edge-centric computation, which natively supports BVP algorithms, and ensures good communication balance, but can incur high communication and computation overhead for UVP algorithms [6], [7].

Therefore, small memory footprint, computation and communication efficiency can not be guaranteed for both UVP and BVP algorithms by the general-purpose computing models in existing systems. In this paper, we propose GraphDuo, a new graph processing framework on Spark [12] based on a dual-model abstraction. The dual-model abstraction provides two different computing models for UVP and BVP algorithms respectively to perform differentiated and efficient computation. The two computing models include 1) a balanced vertex-centric computing model optimized for UVP algorithms, which only needs one computation step in each iteration to ensure low computation latency, and provides efficient mechanisms for high-degree vertices to ensure balanced communication, and 2) an edge-centric computing model optimized for BVP algorithms, which only needs two computing steps and two communication operations in each iteration. Both computing models rely on the same degree-based graph partitioning scheme to ensure storage and communication balance, and the same locality-aware graph layout to ensure fast data lookup and scan. Other optimization techniques, including dynamic model switch and elastic mirror setup mechanisms are also provided to further reduce communication overhead and memory footprint for specific algorithms. According to our evaluation results, GraphDuo outperforms GraphX from 1.78× to 6.56× and from 1.11× to 1.26× for UVP and BVP algorithms, respectively. Comparing with GraphX, GraphDuo can reduce the memory footprint of graph structures by 8%-35%, and reduce the communication overhead by 60%-83% for PageRank on five large graphs from real-world. Considering graph loading time, GraphDuo (writtern in Scala) also outperforms PowerGraph and PowerLyra (written in C++) for several large graphs by up to 2.36×, despite the language efficiencies. The architecture of GraphDuo is illustrated in Fig. 1.

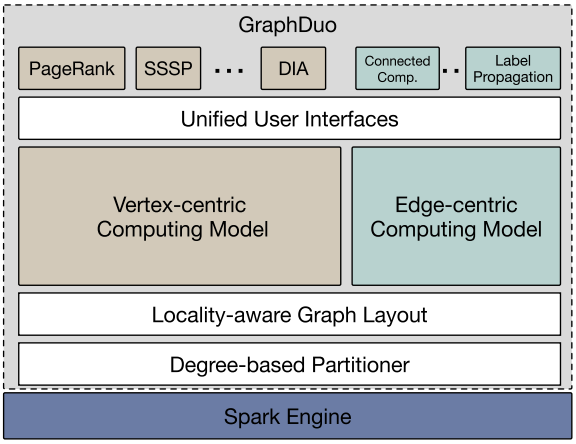


**FIGURE 1.** GraphDuo.

The main contributions of this paper are shown as below:

- We present GraphDuo, a new graph processing framework on Spark based on a dual-model abstraction, which provides two efficient computing models for UVP and BVP algorithms respectively to ensure small memory footprint, computation and communication efficiency. Both computing models rely on an unified degree-based graph partitioning scheme and a locality-aware graph layout for communication balance and fast data lookup (Section III-A, III-B, III-C).
- To further reduce communication overhead and memory footprint, we provide other optimized techniques, including dynamic model switch and elastic mirror setup. Dynamic model switch significantly decreases redundant messages for graph traversal algorithms, while elastic mirror setup eliminates the creation of unnecessary mirrors (Section III-D).
- A detailed comparison evaluation is presented to confirm the efficiency and effectiveness of GraphDuo (Section V).

## II. BACKGROUND AND MOTIVATION

In this section we first give the abstraction of graph computation, then analyze the suitability of current graph processing systems on algorithms with different value propagation behaviors.

### A. GRAPH COMPUTATION ABSTRACTION

In a typical graph system following the *"think like a vertex"* paradigm, an user-defined vertex program runs on the vertices of a/an directed/undirected graph $G = \{V, E\}$ in parallel.

Each vertex $v \in V$ maintains an arbitrary data $D(v)$, and communicates with its neighbors along edges. In each iteration, a vertex is activated by its neighbors or system to perform computation, and votes to halt after computation. A computation job terminates when all vertices vote to halt.

The vertex program defines value computation and propagation behaviors of each vertex, i.e., how a vertex computes its new value based on its neighbors, and how it propagates its value to neighbors. Generally, graph algorithms can be categorized into two classes based on their value propagation behaviors. One is the bidirectional value propagation algorithms (BVP algorithms for short), and the other is the unidirectional value propagation algorithms (UVP algorithm for short). The properties of two types of algorithms are illustrated in Fig. 2.

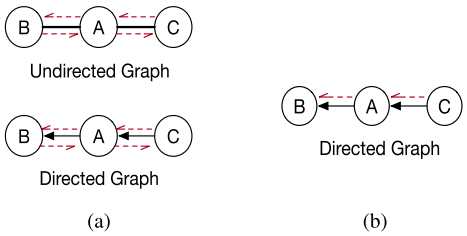


**FIGURE 2.** Different value propagation behaviors in diverse graph algorithms. The red dashed lines represent the direction of value propagation. When the graph is undirected, algorithms running on it are BVP algorithms. BVP algorithms also include algorithms running on directed graphs considering bidirectional value propagation on each edge. UVP algorithms include algorithms on directed graphs with unidirectional value propagation. (a) Bidirectional value propagation. (b) Unidirectional value propagation.

#### 1) BVP ALGORITHMS

For BVP algorithms, communication is bidirectional on each edge, which means two vertices connected with an edge can communicate with each other. If a graph is undirected, each edge can be considered as an edge with two directions, and messages can be propagated along either direction. Therefore, algorithms running on an undirected graph are categorized as BVP algorithms. There are also algorithms running on directed graphs that can be categorized as BVP algorithms, where vertex values need to be propagated along both in-edges and out-edges, such as Strongly and Weakly Connected Components [13]. Properties of such algorithms can be summarized as follows: *(1) The communication on each edge is bidirectional. (2) The vertex value computation relies on all its neighbors.*

#### 2) UVP ALGORITHMS

UVP algorithms include graph algorithms on directed graphs where value propagation is consistent with only one direction. Algorithms such as PageRank, SSSP and Approximate Diameter [14] all fall into this category. Properties of such algorithms can be summarized as follows: *(1) The communication on each edge is unidirectional and consistent with one direction. (2) The value computation relies on neighbors along the other direction.*

Therefore, algorithms of two categories have different neighbor dependencies on value computation and propagation. An efficient graph processing system should consider such different properties and provide flexible and efficient computation and communication mechanisms for both types of algorithms. Meanwhile, low memory footprint should also be guaranteed by the system to ensure efficient computation on few machines.

Another important factor impacting the total communication efficiency is the communication balance, which is highly related to graph partitioning. Many natural graphs follow a skewed degree distribution [9], where some vertices may have significantly much more neighbors than others. Those vertices are called as high-degree vertices, which can become the bottlenecks of communication [3], [15]. Therefore, balanced graph partitioning and communication mechanism should also be considered by the system.

### B. BVP AND UVP ALGORITHMS ON DIFFERENT COMPUTING MODELS

In this subsection, we present a discussion about the suitability of BVP and UVP algorithms on several typical computing models adopted in many state-of-the-practice distributed graph processing systems.

For the ease of discussion, we name neighbors and edges relied by value computation as computation-related neighbors (cr-neighbors) and edges (cr-edges). Neighbors and edges used for value propagation are named as propagation-related neighbors (pr-neighbors) and edges (pr-edges). For UVP algorithms, those two types of sets are disjoint. For example, in PageRank, each vertex relies on the ranks of all its in-neighbors to compute its new rank, so the cr-neighbors are the in-neighbors. Each vertex propagates its rank value to all its out neighbors, so the pr-neighbors are the out-neighbors. For BVP algorithms, those two types of sets are the same, which contains all the neighbors and edges respectively.

#### 1) PUSH-BASED VERTEX-CENTRIC MODEL (PushVM)

Systems such as Pregel [1], Giraph [16] and GraphV [17] adopt a push-based vertex-centric computing model. They employ an edge-cut graph partitioning scheme, which evenly distributes vertices among machines with all their pr-edges (out edges by default). In each iteration, each vertex first computes new value based on received messages, then pushes messages to all its *pr-neighbors* along *pr-edges*.

Sender-side combination is a common optimization used in PushVM systems, where messages with the same target vertices are combined before sent out. This optimization ensures each vertex gets at most one message from each remote partition. The communication overhead of each vertex $v$ is related with the number of partitions having at least one cr-neighbor of $v$.

In PushVM systems, propagation related resources are accumulated locally for each vertex to ensure low

**TABLE 1. A summary of various distributed graph processing systems. U and B Represent UVP and BVP Algorithms Respectively. The number $n_v^{cr}$ denotes the number of partitions having at least one cr-neighbors of vertex $v$, which is the upper bound of communication overhead of $v$.**

| Computing Model | Push-based Vertex-centric | Pull-based Vertex-centric | Edge-centric | Dual-model |
|---|---|---|---|---|
| Systems | Pregel, Giraph, GraphV (with combination) | GraphLab | PowerGraph, GraphX, PowerLyra | GraphDuo |
| Graph Partitioning | Edge-cut | Edge-cut | Vertex-cut | Degree-based |
| Vertex Computation Steps | One step | One step | Multiple steps ($\geq 3$) | **U:** One step<br>**B:** Two steps |
| Communication Overhead | $\leq \#\sum_{v \in V} n_v^{cr}$ | $\leq$ #ghosts | PowerGraph: $\leq 5 \times$ #mirrors<br>GraphX: $\leq 3 \times$ #mirrors<br>PowerLyra: $\leq 4 \times$ #mirrors | **U:** $\leq$ #remote low-degree pr-neighbors + high-degree mirrors<br>**B:** $\leq 2 \times$ #mirrors |
| Load Balance | no | no | yes | yes |

computation latency. The communication is unidirectional, i.e., from each vertex to its *pr-neighbors*. Therefore, such systems naturally support UVP algorithms. For BVP algorithms, however, they have to create a reverse replica for each edge to support bidirectional communication, as shown in Fig. 3, which can result in large memory footprint and high communication overhead.

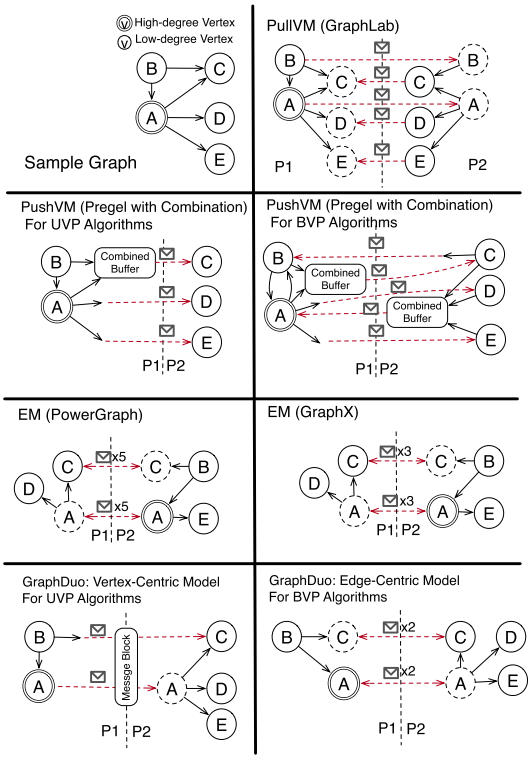


**FIGURE 3. A comparison of different computation models. P1 and P2 represent two different partitions. The red lines represent communication operations between two partitions. In the vertex-centric model of GraphDuo, high-degree vertices propagate their values based on master-mirror synchronization. All the messages sent from each partition with the same target partitions are merged into one message block.**

### 2) PULL-BASED VERTEX-CENTRIC MODEL (PullVM)

In a system employing pull-based vertex-centric computing model, such as GraphLab [2], each active vertex pulls data from its *cr-neighbors* to perform value computation. Each vertex $v$ is assigned to one partition $P$. For each in-neighbor and out-neighbor of $v$ that is not assigned to $P$, a replica (called ghost) is created on P. Ghosts are also created for each edge across machines, as shown in Fig. 3. This ensures all resources can be reachable locally for each vertex, thus to ensure low computation latency. However, all vertices need to synchronize their values with all their remote ghosts after value computation in both UVP and BVP algorithms. The communication overhead of each vertex $v$ is proportional to the number of ghosts of $v$.

### 3) EDGE-CENTRIC MODEL (EM)

Systems such as PowerGraph [3], PowerLyra [6] and GraphX [5] adopt an edge-centric computing model, where the minimum unit of value propagation is each edge. Those systems adopt a vertex-cut partitioning scheme which evenly distributes edges among machines to ensure load and communication balance [3]. Mirrors are created for vertices across machines, of which one is selected as the master.

EM systems decompose the vertex program into different stages, including edge-parallel stages responsible for value propagation and edge value update, and vertex-parallel stages for master vertex value computation and synchronization. Multiple synchronization operations are required in each iteration. For example, in each iteration of PowerGraph [3], mirrors of each vertex gather messages along cr-edges locally, and send them to the master to compute new value. The new value are then synchronized to mirrors. The pr-neighbors are then activated to attend the next iteration. A typical iteration in PowerGraph requires five communication operations [6], as shown in Fig. 3. Three of them are related with vertex activation, and two with value synchronization. In GraphX [5], each iteration requires one activation related synchronization, thus three communication operations in total. PowerLyra [6] provides differentiated computation strategies for low-degree and high-degree vertices. It requires one communication operation for low-degree vertices in each iteration. For high-degree vertices, however, it still needs four communication operations.

In EM systems, each edge has enough information (values of two neighboring vertices) to perform bidirectional value propagation locally, thus to avoid the creation of massive edge replicas for BVP algorithms.

For UVP algorithms, however, such systems fail to explore the unidirectional properties on neighbor dependency and value propagation, which can result in excessive unnecessary messages. For example, as shown in Fig. 3, when running PageRank, vertex $C$ still gets messages from its mirror on P2, and synchronizes its value and activate state with this mirror, even though the mirror is not used by any vertex to compute new rank in P2. Moreover, each iteration requires multiple computation steps, which can lead to high computation latency.

#### 4) GraphDuo

According to the above analysis, PushVM and PullVM systems need to create extra edge replicas for BVP algorithms, which can result in large memory footprint and high communication overhead. For UVP algorithms, PushVM systems can ensure lower computation latency by exploring the locality of pr-neighbors, and achieve smaller memory footprint (no replicas of edges and vertices) than EM and PullVM systems.

Communication balance can not be ensured by traditional PushVM and PullVM systems for natural graphs [3]. Even though there are systems such as GPS [18] and PregelPlus [15] providing efficient mechanism to handle high-degree vertices, they still lack support for efficient bidirectional communication.

Considering different properties of UVP and BVP algorithms, and the suitability of different computing models, we propose a dual-model graph processing framework called GraphDuo, which provides two different efficient computing models specially designed for BVP and UVP algorithms respectively. An optimized edge-centric model is provided for BVP algorithms, which naturally support bidirectional communication, and merges all activation related operations with value synchronization in one phase to reduce the communication overhead. Each iteration requires only two synchronization operations, and two computation steps in this model, which ensures low communication overhead and computation latency, as shown in Fig. 3.

A balanced vertex-centric model is provided for UVP algorithms, which considers the unidirectional properties in such algorithms to avoid unnecessary communication. To achieve good communication balance, pr-edges of each high-degree vertex are spread out across partitions to perform distributed value propagation. Combined with a hybrid message combination mechanism, only one computation step and one communication operation is required in each iteration, thus to achieve low computation latency. Both computing models rely on a same degree-based partitioning scheme, which can ensure load and communication balance for both models, and accumulate propagation related resources locally for vertex-centric model.

### C. MATH ANALYSIS AND QUANTITATIVE COMPARISON

To further prove the suitability of different computing models on UVP and BVP algorithms, and the efficiency of the two computing models in GraphDuo, we provide a math analysis and quantitative comparison in this subsection. We mainly focus on the communication overhead, which is the key factor that impacts the total efficiency of computation and communication.

As discussed above, for EM systems, the communication overhead of each iteration is related with the replicate number of vertices, and the number of synchronization operations. Let $r_v$ denote the number of replicas of vertex $v$, then the communication overhead of $v$ is $k(r_v - 1)$, where $k$ denotes the number of synchronization operations in each iteration. $k$ is 5 in PowerGraph, and 3 in GraphX. Let $r_{avg}$ denote the average replication factor of all vertices, then the expected communication overhead of EM systems is:

$$E[C_{EM}] = \frac{\sum_{v \in V} k(r_v - 1)}{|V|} = k(r_{avg} - 1) \tag{1}$$

For PushVM systems, when considering message combination, the communication overhead of each vertex $v$ is related with the number of remote partitions having at least one cr-neighbor of $v$, which is denoted as $n_v^{cr}$. Therefore, the *expected communication overhead* of PushVM is:

$$E[C_{PushVM}] = \frac{\sum_{v \in V} n_v^{cr}}{|V|} = n_{avg}^{cr}, \tag{2}$$

where $n_{avg}^{cr}$ is the average number of $n_v^{cr}$. When performing UVP algorithms, cr-neighbors of each vertex are its out-neighbors or in-neighbors. When performing BVP algorithms, cr-neighbors include all neighbors.

For PullVM systems, each active vertex sends its value to all its ghosts in each iteration. The communication overhead of each vertex $v$ is equal to the number of ghosts of $v$, which is equal to the number of remote partitions having at least one in-neighbor or out-neighbor of $v$. We denote this number as $n_v$, and the average number is $n_{avg}$. As PushVM and PullVM systems both adopt an edge-cut partitioning scheme, the communication overhead of PullVM systems is similar with the case of PushVM systems performing BVP algorithms. The *expected communication overhead* of PullVM systems is thus:

$$E[C_{PullVM}] = \frac{\sum_{v \in V} n_v}{|V|} = n_{avg} \tag{3}$$

In GraphDuo, when performing UVP algorithms, balanced vertex-centric computing model is used. For each high-degree vertex, the pr-edges are spread out among partitions to perform distributed value propagation. Replicas (mirrors) are created on each partition having pr-edges for each high-degree vertex, of which one is selected as the master. Distributed value propagation is performed via master-mirror synchronization, where the master sends a synchronization message to all its mirrors, then the mirrors propagate values to local pr-neighbors. For low-degree vertices, the value propagation is performed in the same way of PushVM. The number of replicas of a high-degree vertex $v_h$ is denoted as $r_{v_h}$, then the communication overhead of $v_h$ is equal to $r_{v_h} - 1$. For each low-degree vertex $v_l$, the communication

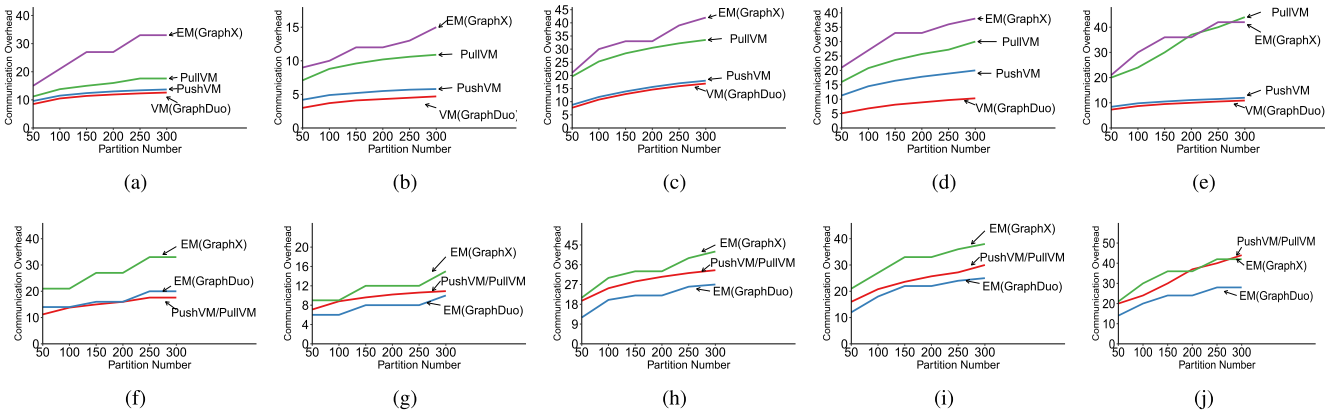


**FIGURE 4.** Quantitative comparison of different computing models in different systems. The optimized edge-centric computing model in GraphDuo is added in the comparison of the case of BVP algorithms. (a) UVP-LJ. (b) UVP-Wiki. (c) UVP-UK. (d) UVP-Twitter. (e) UVP-SK. (f) BVP-LJ. (g) BVP-Wiki. (h) BVP-UK. (i) BVP-Twitter. (j) BVP-SK.

overhead is the same with that of PushVM, which is $n_{v_l}^{cr}$. The *expected communication overhead* of the vertex-centric model in GraphDuo is thus:

$$E[C_{VM_{GraphDuo}}] = \frac{\sum_{v_h \in V_h}(r_{v_h} - 1) + \sum_{v_l \in V_l} n_{v_l}^{cr}}{|V|}, \quad (4)$$

where $V_h$ denotes the set of high-degree vertices, and $V_l$ denotes the set of low-degree vertices.

For BVP algorithms, GraphDuo uses the optimized edge-centric model (EM). Therefore, the communication overhead is also related to the number of replicas, but it only requires two synchronization operations in each iteration. The expected communication overhead is thus:

$$E[C_{EM_{GraphDuo}}] = \frac{\sum_{v \in V} 2(r_v - 1)}{|V|} = 2(r_{avg} - 1) \quad (5)$$

We then study the expected communication overhead of different computing models on five representative real-world graphs. The graph details are shown in Table. 2. We calculate the communication overhead based on above analysis. Since the communication overhead is related to the partitioning scheme, we consider the usual partitioning schemes adopted in real systems. In PushVM and PullVM systems, we consider a hash-based edge-cut partitioning scheme, which is the default partitioning scheme in many systems [1], [2], [16].

**TABLE 2.** Graph details.

| Graph | Description | Vertices | Edges |
|---|---|---|---|
| LiveJournal (LJ) | LiveJournal friendship network [19] | 5.4M | 79M |
| Wikipedia-en (Wiki) | The hyperlink network of Wikipedia [20] | 18.2M | 172.2M |
| uk-2005 (UK) | The network of the .uk domain in 2005 [21] [22] | 39.4M | 936.3M |
| Twitter-2010 (Twitter) | Twitter follower network [23] | 41.6M | 1.4B |
| sk-2005 (SK) | The network of the .sk domain in 2005 [21] [22] | 50.6M | 1.9B |

For EM systems, we consider a grid vertex-cut partitioning scheme, which is also hash-based and widely supported by many EM systems [24]. The replication factor of this scheme is larger than greedy and other complex vertex-cut schemes [6], thus it can be used to compute the upper bound of communication overhead in EM systems. Fig. 4 plots the comparison results for different partition numbers ranging from 50 to 300.

As shown in Fig. 4, PushVM systems can achieve significantly less communication overhead than PullVM and EM systems for UVP algorithms in all five graphs. For BVP algorithms, the communication overhead of current EM systems is still slightly higher than that of PushVM and PullVM, due to multiple synchronization operations. For the edge-centric computing model in GraphDuo, only two synchronization operations are required for BVP algorithms, so the expected communication overhead is the least among different models. The expected communication overhead of GraphDuo is also the least for UVP algorithms, due to the distributed value propagation in vertex-centric model that can help to reduce the number of messages generated by high-degree vertices, which will be discussed in details in the next section.

## III. GraphDuo

In this section, we provide more details about main components of GraphDuo, which include: (1) A degree-based partitioning scheme, which can help to explore the locality during the computation of UVP algorithms, and guarantee communication balance for both UVP and BVP algorithms. (2) A dual-model abstraction, which provides two different but efficient computing models for UVP and BVP algorithms respectively to achieve efficient graph computation with low communication cost and load balance. (3) A locality-aware graph layout for fast lookup and data scan.

Moreover, GraphDuo also provides two optimization techniques to further reduce the communication overhead and memory footprint, including dynamic model switching and elastic mirror construction.

### A. DEGREE-BASED GRAPH PARTITIONING

Before running a graph algorithm, the graph data are first loaded from HDFS [25] or other file systems and split into different partitions by a graph loader in GraphDuo. GraphDuo employs a degree-based graph partitioning scheme, which accumulates all pr-edges of each low-degree vertex in one partition, and spreads out pr-edges of each high-degree vertex among partitions. The partitioning scheme needs to take a propagation direction parameter from users, and takes two steps to split a graph dataset. We assume the value propagation is along the outgoing direction. Vertices are first evenly assigned to partitions with all their out-edges based on a consistent hash function on vertex id. Vertices having degree larger than a threshold are treated as high-degree vertices. Another repartitioning step is performed to reassign their out-edges across different partitions based on the out-neighbor id. By default, we set the high-degree threshold to 1000, which is an empirical value according to our experiments. A comparsion among traditional edge-cut, grid vertex-cut [26] and our degree-based partitioning scheme is shown in Fig. 5.

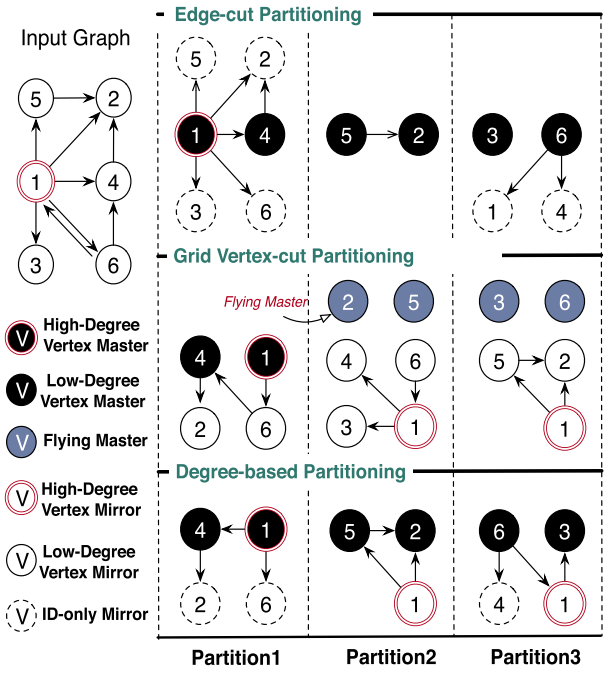


**FIGURE 5.** Comparsion among edge-cut, grid vertex-cut and degree-based partitioning scheme.

This partitioning scheme is also suitable for BVP algorithms. The reason is that such partitioning scheme can guarantee lower replication factor than traditional vertex-cut schemes, and can ensure low synchronize cost and good communication balance [6], [27], [28]. In this way, we can unify the graph partitioning scheme for two computing models, which enables seamlessly model switching for different types of algorithms with no extra cost.

The degree-based graph partitioning is inspired by the hybrid-cut of PowerLyra [6], which also provides differentiated partitioning for low-degree and high-degree vertices. The original hybrid-cut partitioning in PowerLyra assigns low-degree vertices along with all in-edges to partitions, and distributes in-edges of high-degree vertices across different partitions. In such way, only high-degree vertices need to create replicas. Different from PowerLyra, GraphDuo considers the locality of out-neighbors by default, which are the pr-neighbors in most UVP algorithms. This partitioning strategy accumulates all the propagation related resources together locally for low-degree vertices. Integreted with our hybrid communication mechanism, which will be described later, we enable the vertex-centric computing model to perform only one computation step in each iteration for UVP algorithms, with only one communication operation. For UVP algorithms propagating values along incoming direction, such as Approximate Diameter (DIA) [14], users can also set the partitioning related direction to the incoming direction in their programs.

### B. DUAL-MODEL ABSTRACTION

Considering the different system requirements of UVP and BVP algorithms, GraphDuo employs a dual-model abstraction, which involves a balanced vertex-centric computing model specially designed for UVP algorithms, and an optimized edge-centric computing model for BVP algorithms. Combined with other communication optimization mechanisms, the vertex-centric model can merge all operations required in each iteration to just one computation step, with only one communication operation, thus to achieve low computation latency and low communication overhead. The edge-centric model requires only two steps in each iteration, which is also less than that in existing edge-centric systems. For model selection, by default, system selects the computing model based on a direction parameter (in, out or both direction) provided by user. A low level interface is also provided for users to select model according to the specific workload.

#### 1) BALANCED VERTEX-CENTRIC COMPUTING MODEL

We propose a balanced vertex-centric computing model for UVP algorithms, which considers differentiated and efficient computation and communication strategies for both low-degree and high-degree vertices.

Generally, in each iteration, all vertices first perform the user-defined vertex computation program to update values based on received messages. Vertices with new values are activated, and continue to propagate values along edges. To achieve low computation and communication cost, our vertex-centric computing model merges vertex activation, vertex value update, message generation and other communication optimizations in one computation step. To achieve good communication balance, the vertex-centric computing model performs different communication strategies for low-degree and high-degree vertices, to eliminate the imbalanced communication caused by the high-degree vertices.

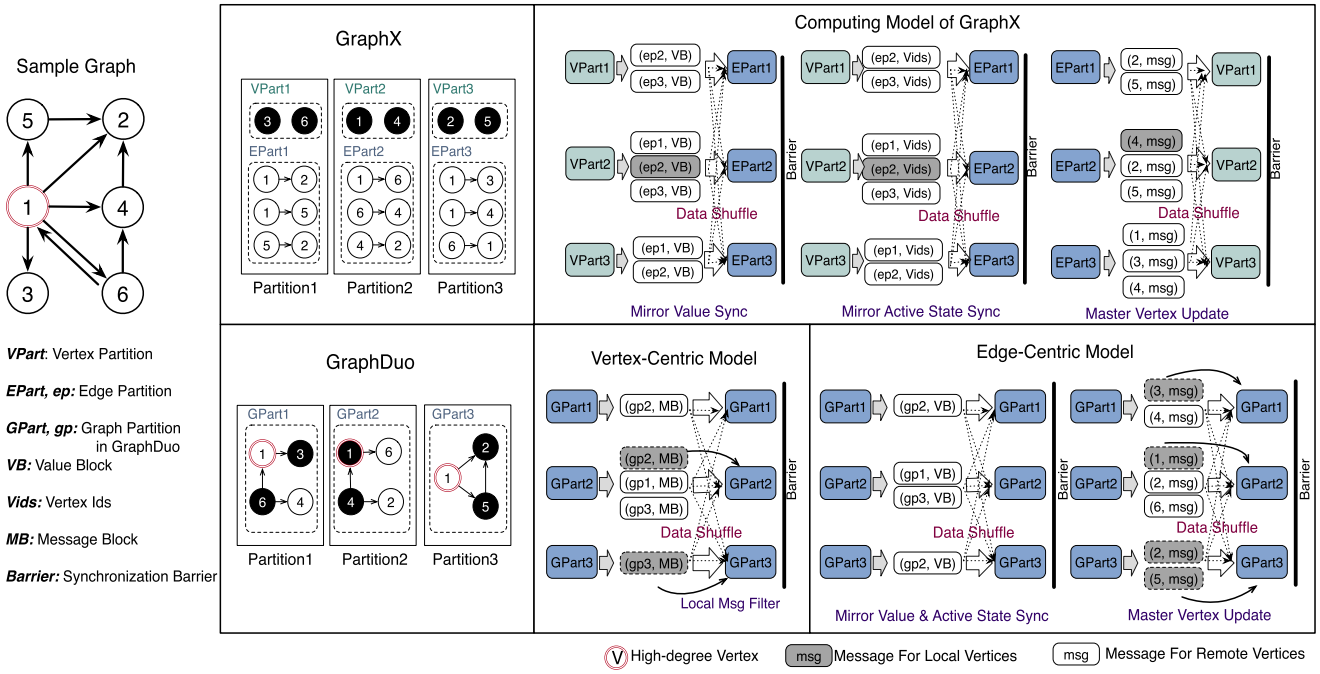


**FIGURE 6.** Comparsion between our differentiated computing models and the original computing model of GraphX. For GraphX and GraphDuo's vertex-centric computing model, we suppose it runs a PageRank workload.

As discussed above, the partitioning scheme can accumulate pr-edges locally for low-degree vertices, and spread out the pr-edges across different partitions for high-degree vertices. For low-degree vertices, each active vertex computes new value and generates messages for all its pr-neighbors. For each high-degree vertex, mirrors are created on each partition having its pr-edges to perform distributed value propagation. In each iteration, each active high-degree vertex first generates synchronization messages to all its mirrors. Each mirror then performs a local scatter operation to forward values to their pr-neighbors.

The balanced vertex-centric computing model ensures that each high-degree vertex generates at most one message for each remote partition. Therefore, the number of messages generated by each high-degree vertex is at most P - 1, where P is the number of partitions. In the original push-based vertex-centric model, the message number is proportional to the number of remote pr-neighbors, which can be much larger than the number of partitions. For this reason, our balanced vertex-centric computing model can achieve a lower communication overhead than those systems.

Furthermore, GraphDuo's vertex-centric model merge2 the high-degree mirror synchronization operation together with the regular message transmit from low-degree vertices to their neighbors. In this way, GraphDuo ensures only one communication happens for each vertex computation round. To achieve this, we propose a hybrid message combiner, where the messages from low-degree vertices and mirroring messages from high-degree vertices for the same remote partitions are combined together in one hybrid message block. After the communication, each partiton receives message blocks from other partitions, then extracts the mirroring messages out to perform the local scatter operation. Local messages are generated by this operation for the pr-neighbors which are on this partition. A local message combination is proceeded to combine all the remote messages and local messages for same vertices. The vertex computation program is then executed for each vertex to update new value based on the final message. A comparison between our vertex-centric model and the original computing model of GraphX is shown in Fig. 6.

To prove the efficiency of the balanced vertex-centric computing model, we perform a comparison between GraphDuo and GraphV [17], which is a Pregel-like PushVM graph processing framework on Spark. We run the PageRank workload on the Twitter follower graph on both systems, and measure the communication cost of each partition in the first iteration. The partition number is set to 96, and the graph details are shown in Section V. The results are illustrated in Fig. 7, which demonstrates that GraphDuo can achieve better communication balance and lower communication overhead across partitions than GraphV.

The computation strategy for high-degree vertices in GraphDuo is similar as the mirroring technique in PregelPlus [15]. However, PregelPlus requires an extra communication operation for mirror synchronization, which can introduce extra computation cost.

### 2) EDGE-CENTRIC COMPUTING MODEL

We propose an optimized edge-centric computing model for BVP algorithms, which inherits from GraphX with fewer

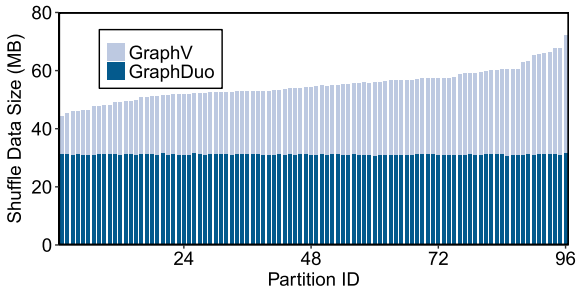


**FIGURE 7.** Comparison of communication balance and overhead between GraphDuo and GraphV. The shuffle data size is got from Spark logs.

computation steps and communication operations. The original edge-centric computing model in GraphX needs two passes over the vertex data for mirror synchronization and activation respectively, as shown in Fig. 6. Comparing with GraphX, the edge-centric computing model in GraphDuo merges the mirror synchronization and activation together in one step, to ensure only one pass over the vertex and edge data in each iteration.

In our edge-centric computing model, each iteration only takes two computation steps, including one edge-parallel step for message generation and one vertex-parallel step for vertex value update, mirror synchronization and activation. In the edge-parallel step, the remote mirrors of each vertex are first synchronized and activated by mirroring messages sent from its master in the previous iteration. Edges having at least one active neighbor produces messages according to an user-defined message generation function. This function defines how the messages are generated based on two neighboring vertex values and edge value, and the target of each message. Sender-side combination is also considered in this step to promise only one message is sent to each remote vertex. In the vertex-parallel step, each vertex receives messages from remote partitions, updates new value, and produces mirroring messages to all its remote mirrors. The main phases of the optimized edge-centric computing model is also illustrated in Fig. 6.

### C. LOCAL GRAPH STRUCTURE WITH LOCALITY-AWARE GRAPH LAYOUT

GraphDuo constructs a local graph structure in each partition for all the vertices and edges assigned to it. Each partition creates a master value copy for every vertex assigned to it. By default, high-degree mirrors are created for remote high-degree masters having propagation related edges assigned to this partition. Low-degree mirrors are also created for pr-neighbors whose masters have low degrees and are not on this partition. After this phase, vertices on each partition can be categorized into four classes, including high-degree masters, low-degree masters, high-degree mirrors and low-degree mirrors. The edge field is maintained on the same partition where the edge locates. To support fast data lookup and sequential data scan, we organize the local graph in each partition based on a locality-aware layout. After partitioning, vertices on each partition are first grouped based on their classes and ordered by the sequence of high-degree masters, low-degree masters, high-degree mirrors, and low-degree mirrors. Vertices are then transformed into local ids, and the order can ensure that vertices of same class are assigned with sequential local ids. Moreover, for the reason that all the masters and mirrors are put together respectively, the masters and mirrors both have their local ids sequential. Therefore, we can use a simple array to store master and mirror values respectively, which can ensure fast vertex data lookup and scan.

The local graph layout also considers the locality of edges. Edges in each partition are clustered by their corresponding vertices, which can be low-degree masters, high-degree masters or high-degree mirrors. Based on our vertex layout discussed above, all vertices having edges are put together, which can ensure fast edge scan for vertex-centric computing model. The start positions of neighboring edges of each vertex are maintained in a compressed sparse row (CSR) structure, which is also an array structure since all vertices having edges are put together with sequential local ids.

### D. OPTIMIZATION TECHNIQUES FOR MESSAGE AND MEMORY FOOTPRINT REDUCTION

To further decrease the communication overhead and memory footprint, we provide two techniques, including a dynamic model switching mechanism to reduce redundant messages during the computation, and elastic mirror construction to avoid the creation of unnecessary mirrors.

#### 1) DYNAMIC MODEL SWITCHING MECHANISM FOR REDUNDANT MESSAGE REDUCTION

The computation and communication cost of each iteration is highly related to the size of active edge set, which implies how many edges are involved in this computation round. The active edge set can become sparse or dense during the execution. For algorithms such as PageRank and HashMin [29], [30], the active edge set is dense in the first few iterations, and become increasingly sparse as the computation proceeds. For graph traversal algorithms, such as BFS and SSSP, the active edge set starts from sparse, and becomes denser as more vertices are activated by their in-neighbors. The active edge set then becomes sparse again as more and more vertices reach convergence.

An iteration with dense active edge set may generates extensive messages. Meanwhile, as the active edge set becomes denser in graph traversal algorithms, the fraction of vertices having been visited increases rapidly. Some vertices can even approach converge after only a few iterations. Messages sent to those visited vertices may become redundant, which introduces unnecessary communication overhead. For example, in SSSP, each active vertex sends its new shortest distance to all its pr-neighbors. If the neighbor has already converged, or the new distance is larger than current value of the neighbor, this message will not be used and become redundant. Such redundant messages can not be

avoid with the our vertex-centric model, because each active vertex is not aware of the current values of its pr-neighbors.

To handle the redundant message problems in graph traversal algorithms, or other algorithms with the similar activity distribution, we consider a combination of two computing models. At the beginning, vertex-centric computing model is used. When the active edge set becomes dense enough (i.e., larger than a threshold), GraphDuo switches to the edge-centric computing model, in which the pr-neighbors of each vertex are first synchronized with their remote masters to get their latest values. A local message generate function is then performed on each propagation related edge of each active vertex. Since the values of pr-neighbors are now reachable by the function, redundant messages can be directly filtered and removed. In such way, we can significantly reduce the communication overhead for iterations with dense active edge set.

To support this, we add a dynamic model switching mechanism, where the computing model switches between vertex-centric and edge-centric according to the size of active edge set. For the fact that both models rely on the same partitioning scheme and same local graph structure, the switching operation is straightforward and easy to be performed. Such mechanism is similar as the dual update propagation model in Gemini [31]. Different from Gemini, we do not need to create extra edge replicas to support propagation on two directions. Same as Gemini, we set the threshold between sparse and dense active edge set as |E|/20, where |E| means the number of total edges. Fig. 8 shows the difference of message number distributions when the SSSP workload runs on GraphDuo with or without dynamic model switching mechansim. The graph data is the Twitter follower graph, whose details can be found in Section II. As illustrate in Fig. 8, when dynamic model switching is enabled, the number of redundant messages can be significantly decreased, which can result in less communication overhead.

### 2) ELASTIC MIRROR CONSTRUCTION

For many UVP algorithms, such as PageRank, values of pr-neighbors are not required by vertex computation related functions. Therefore, it is not necessary to maintain mirrors for their pr-neighbors, which provides a chance to further reduce the memory footprint. We propose an elastic mirror construction mechanism, which constructs mirrors for the pr-neighbors of each vertex only when the values of pr-neighbors are required by the graph algorithms. For UVP algorithms that do not involve pr-neighbors in vertex computation, each local graph does not construct mirrors for remote pr-neighbors. For BVP algorithms and graph traversal algorithms that requires the support of dynamic model switching, mirrors are created for all remote vertices before the computation proceeds. In other words, the construction of mirrors for pr-neighbors only when the edge-centric model is used during the whole computation period, which is decided by the propagation direction and dynamic model switching option given by the user-defined function. As illustrated in Fig. 9, the elastic mirror construction mechanism can reduce the memory footprint by 6%-35% for the five graphs used in this paper when running PageRank, comparing with the default case when mirrors are indiscriminately created for all remote masters. The details of five graphs are shown in Section II.

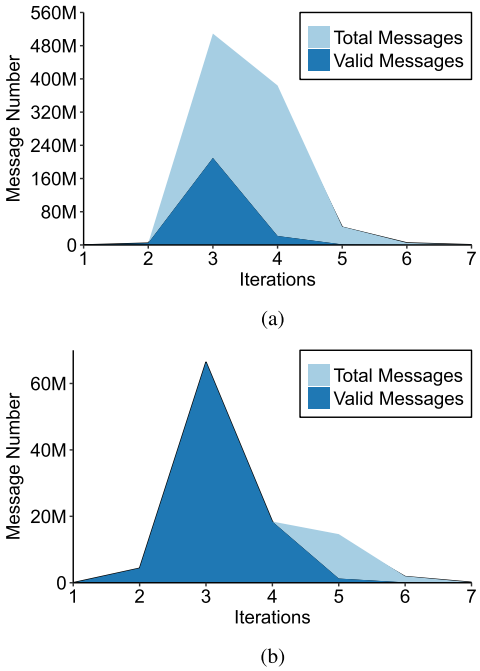


**FIGURE 8.** Message number distribution of GraphDuo with or without dynamic model switching when running SSSP. A message is valid only if its value is smaller than that of its target, which means it contributes to the computation of its target's new value. In vertex-centric model, the message number is the sum of messages contained in all message blocks. (a) GraphDuo without dynamic model switching. (b) GraphDuo with dynamic model switching.

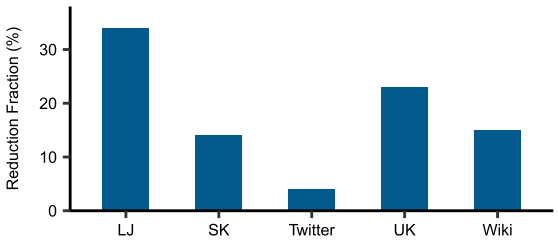


**FIGURE 9.** The reduction fraction of memory footprint for PageRank on five graphs when the elastic mirror construction mechansim is enabled.

## IV. PROGRAMMING MODEL

GraphDuo is built on a general-purpose big data analytics system Spark [12]. The programming model of GraphDuo is built on top of the Resilient Distributed DataSets (RDD) abstraction of Spark. Spark provides developers with a variety of dataflow operators, such as group by, join and filter. We choose to implement GraphDuo on Spark because of the good scalability, the high-performance in-memory execution mechanism and the rich set of dataflow operators. However, the dual-model abstraction is independent of Spark, and can be implemented on other software stacks.

GraphDuo inherits the integrated graph and collection abstraction from GraphX, where graph data can be viewed both as a graph and as a table. By employing such abstraction, graph structures can be seamlessly recast to collections to support other complex analytics operations in the graph analysis pipeline.

The graph abtraction in GraphX represents a graph as a pair of vertex and edge collections, which both contains many graph-specific structures. This abstraction makes it easy to analyze the vertex data and edge data separately. However, such design introduces extra memory storage for global vertex states, which introduces extra memory consumption and graph loading overhead.

Different from GraphX, GraphDuo uses only one collection to abstract the whole graph which contains all the structures and operators needed for graph computation. The vertex computation procedures of two computing models in GraphDuo are both represented as a sequence of map, join and reduce operations, which is simliar as GraphX.

The basic user interfaces of the graph collection in GraphDuo are shown in Table. 3. Similar as GraphX, GraphDuo provide interfaces to generate the collections of vertex and edge, and map functions to change the vertex and edge values. GraphDuo also provides a Pregel-style graph computation API for users, which takes three user-defined functions and a propagation direction as parameters. The graph computation performs a computation loop to iteratively generate messages and update vertex values.

**TABLE 3. The basic interfaces of GraphDuo.**

| Operator | Description |
|---|---|
| vertices: Collection[(Id, V)] | Get the collection of vertices |
| edges: Collection[(Id, Id, E)] | Get the collection of edges |
| outDegrees: Collection[(Id, Int)] | Get the out degrees of all vertices |
| inDegrees: Collection[(Id, Int)] | Get the in degrees of all vertices |
| mapV (f: (Id, V) ⇒ V): Graph[V, E] | Perform a map function on all vertices |
| mapE (f: (Id, Id, E) ⇒ E): Graph[V, E] | Perform a map function on all edges |
| compute(sendMsg: (Id, V, E) ⇒ M, mergeMsg: (M, M) ⇒ M, updateV: (Id, V, M) ⇒ V, direction, ...): Graph[V, E] | Perform the main vertex computation based on three user-defined functions and a given propagation direction |

## V. EVALUATION

We have implemented GraphDuo on Spark 2.1.2, and evaluate GraphDuo against many state-of-the-art systems including GraphX (on Spark 2.1.2), GraphV (on Spark 2.1.2), Giraph (1.2.0 on Hadoop 2.7.1), PowerGraph 2.2, and PowerLyra (implemented based on PowerGraph 2.2). Giraph is an open source implementation of Pregel on Hadoop. PowerGraph and PowerLyra are two open source distributed graph computation engine which are implemented in C++ and based on MPI framework for fast communication. We exclude GraphLab as it is an older version of PowerGraph, and is less performant than PowerGraph and Giraph [3], [32]. Three benchmarks are used for evaluation, including PageRank, Single-Source Shortest Path (SSSP), and HashMin [29], [30].

The partitioning schemes used in GraphX, PowerGraph and PowerLyra are 2D edge partitioning, grid vertex-cut and hybrid-cut respectively. Giraph uses the default hash partitioning scheme. We run a push-based PageRank implementation [33] on Giraph, GraphDuo, GraphV and GraphX, which considers dynamic computation, and has the similar activity distribution as that in the pull-based PageRank implemented in PowerGraph and PowerLyra. In push-based PageRank, active vertices push the changes of their ranks (delta ranks) to their neighbors, and vertices accumulate all the delta contributions from their neighbors and sum them up in each iteration. Vertices only update their values and become active if the sum distribution is larger than a threshold. For SSSP, the weight of each edge is normalized as 1.0. HashMin is used to compute weakly connected components of a directed graph or connected components of an undirected graph, where each vertex first maintains its own id as its value, then iteratively sends its id to all its neighbors, and updates its value with the smallest id it receives. If the smallest id is bigger than its current value, the vertex does not updates its value and turns to convergence.

The cluster we used for evaluation contains 9 Huawei RH2285 servers connected with a 1GigE network. Each server is equipped with two Intel Xeon E5645 processors, including 12 physical cores. The memory is 32GB, and the disk size is 1TB. The graphs we use for evaluation have already been described in Table. 2 in Section II.

### A. COMPARING WITH OTHER SPARK-BASED GRAPH PROCESSING FRAMEWORKS

First of all, we present a comparsion evaluation between GraphDuo and other two Spark-based graph processing frameworks: GraphV and GraphX. All the frameworks run on Spark 2.1.2, with the configurations shown in Table. 4.

**TABLE 4. Spark configuration.**

| Property | Value |
|---|---|
| spark.executor.memory | 27 GB |
| spark.executor.cores | 12 |
| spark.memory.storageFraction | 0.5 |
| spark.default.parallelism | 96 |
| spark.reducer.maxSizeInFlight | 100 MB |

We first show the comparison results among GraphDuo, GraphV and GraphX when running delta PageRank, on the aspects of graph loading time, memory consumption and communication cost. The results are shown in Table. 5. The memory footprint is the total size of local graph structures in all partitions, which is recorded in the storage summary of Spark. The GC time records the total time used for garbage collection during the computation, and the data

**TABLE 5. Detailed comparison among GraphDuo, GraphV and GraphX.**

| DataSet | Graph Loading Time(s) | | | Memory Footprint(GB) | | | GC Time(min) | | | Data Shuffle(GB) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | GraphX | GraphV | GraphDuo | GraphX | GraphV | GraphDuo | GraphX | GraphV | GraphDuo | GraphX | GraphV | GraphDuo |
| LJ | 42 | *26* | 30 | 4.3 | 3.0 | *2.8* | 11 | 4.1 | *2.9* | 32.5 | 18.6 | *13.4* |
| Wiki | 80 | *39* | 40 | 6.5 | 6.5 | *6.0* | 24 | 9.5 | *7.2* | 40.6 | 13.7 | *9.9* |
| UK | 348 | *164* | 175 | 32 | 26.6 | *25.5* | 816 | 72 | *60* | 422.9 | 100 | *70.9* |
| Twitter | 556 | *247* | 248 | 35.2 | 33.6 | *26.2* | 462 | 138 | *59* | 269.6 | 150 | *79.4* |
| SK | 1032 | *311* | 320 | 41.4 | 39.3 | *38* | >642 | *90* | *90* | >1300 | 78.5 | *53.7* |

shuffle records the size of data exchanged among machines. All records are extracted from the execution logs of Spark.

As shown in Table 5, GraphDuo can achieve the smallest memory footprint, least GC time and lowest communication cost on all five datasets. GraphV gets the least graph loading time, due to its simple hash-based graph partitioning. The graph loading time of GraphDuo is slightly longer than that of GraphV, for the reason that its graph partitioning scheme requires an extra step to repartition edges of high-degree vertices. GraphX takes the longest graph loading time, for the reason that GraphX needs to build an extra structure to store the partition infos of all the mirrors for each vertex, which are used for synchronization. To build this, a data exchange operation is needed during the graph loading, and GraphX needs to maintain a huge index to records the partitions having replicas for each vertex. The smallest memory footprint in GraphDuo is mainly attributed to the degree-based graph partitioning and the elastic mirror construction technique, which can eliminate the creation of unnecessary mirrors.

The lowest communication cost and least GC (garbage collection) time in GraphDuo are attributed to the efficient vertex-centric computation model and hybrid communcation mechanism. In each iteration, GraphDuo's vertex-centric model requires only one communication operation for each iteration, whereas GraphX needs three, as discussed in the previous section. Comparing with GraphV, GraphDuo also considers efficient support for high-degree vertices, which can significantly reduce the number of messages sent from high-degree vertices and achieve good communication balance.

Furthermore, we compare the execution time of three workloads among GraphDuo, GraphV and GraphX. As discussed in the previous sections, GraphDuo employs vertex-centric computing model to execute PageRank and SSSP, and edge-centric computing model to execute HashMin. The comparsion results are shown in Fig. 10. As illustrated in Fig. 10, GraphDuo can outperform GraphX by 3.40×-5.90× and outperform GraphV by 1.10× to 1.90× for PageRank. For sk-2005, GraphX cannot finish in 8000 seconds, so we compute the speedup between GraphV and GraphX. The speedup of GraphDuo over GraphX on SSSP and HashMin ranges from 1.78× to 5.43× and 1.11× to 1.26× respectively. For HashMin, GraphV can not run on sk-2005, for the reason that it needs to create reverse replicas for each edge to support bidirectional communication. The large memory footprint can exceed the memory capacity of our cluster. Meanwhile, the performance of HashMin

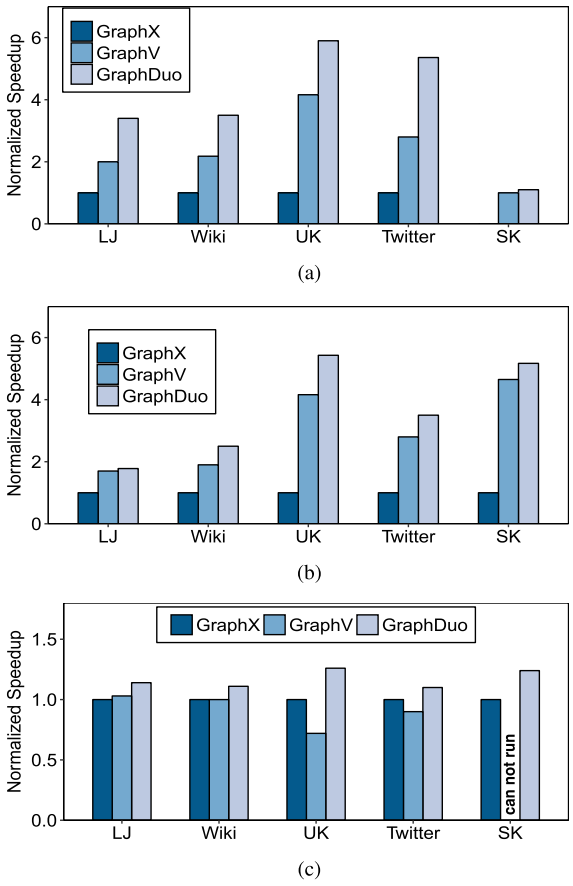


**FIGURE 10. Performance comparison among GraphX, GraphV and GraphDuo. Higher is better. For PageRank on SK graph, GraphX can not finish in 8000 seconds, so we compute the speedup of GraphDuo against GraphV. (a) PageRank (b) SSSP. (c) HashMin.**

in GraphV on LJ, Wiki and Twitter graph is close to that of GraphX. The reason that all these graphs have small diameters (28, 12, 18 respectively), so HashMin can converge fast and terminates after few iterations (less than 30). The differences of computing models are not obvious in these cases. However, GraphDuo can still achieve the best performance on these graphs.

We further perform another comparison between GraphDuo and GraphX on larger clusters. The cluster size increases from 8 to 24 machines (a factor of 8). We run the PageRank workload on the Twitter graph for two systems and the normalized speedup of GraphDuo is shown in Fig. 11. As illustrated in Fig. 11, GraphDuo can outperform GraphX from 5.36× to 6.56× as the number of machines increasing

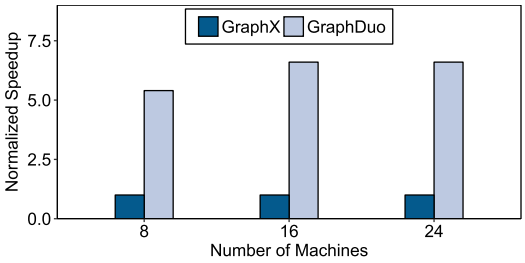


**FIGURE 11.** The speedup of GraphDuo compared with GraphX on clusters with different number of machines when running PageRank.

from 8 to 24. The result shows that GraphDuo can also achieve good performance on large cluster.

### B. COMPARING WITH OTHER GRAPH PROCESSING SYSTEMS

We further perform another expriment to compare the performance among Giraph, GraphDuo, PowerGraph and PowerLyra. In this comparison, we also consider the graph loading time, as it also impacts the total execution time. The result is illustrated in Fig. 12. All the systems have been carefully tuned. All the experiments run five times and we get the average value to plot.

For PageRank, GraphDuo can achieve better performance than Giraph, PowerGraph and PowerLyra on Wiki, UK, Twitter and SK graphs when considering the graph loading time. For SSSP and HashMin, GraphDuo can also achieve better performance than PowerGraph and PowerLyra on Twitter and SK graphs, for the reason that the graph loading time is much less in GraphDuo than that of these two systems. The low graph loading cost of GraphDuo is mainly due to the efficient parallel operators on Spark and our two-step degree-based graph partitioning scheme. Giraph gets the worst performance in all cases, even though the message combiner is used. Meanwhile, it can not handle SK graph on our local cluster. The main reason that Giraph does not provide efficient mechanisms to handle high-degree vertices. Moreover, Hadoop lacks efficient support for iterative computation.

Meanwhile, the execution performance of GraphDuo is close to that of PowerGraph and PowerLyra, despite the fact that GraphDuo is written in a JVM-based language, and relies on common network stacks for communication, which can be much more inefficient than the MPI communication framework used in PowerGraph and PowerLyra [4]. On the other hand, GraphDuo need to write intermediate data into disks to promise good fault tolerance and scalability. In PowerGraph and PowerLyra, however, the intermedia data are maintained

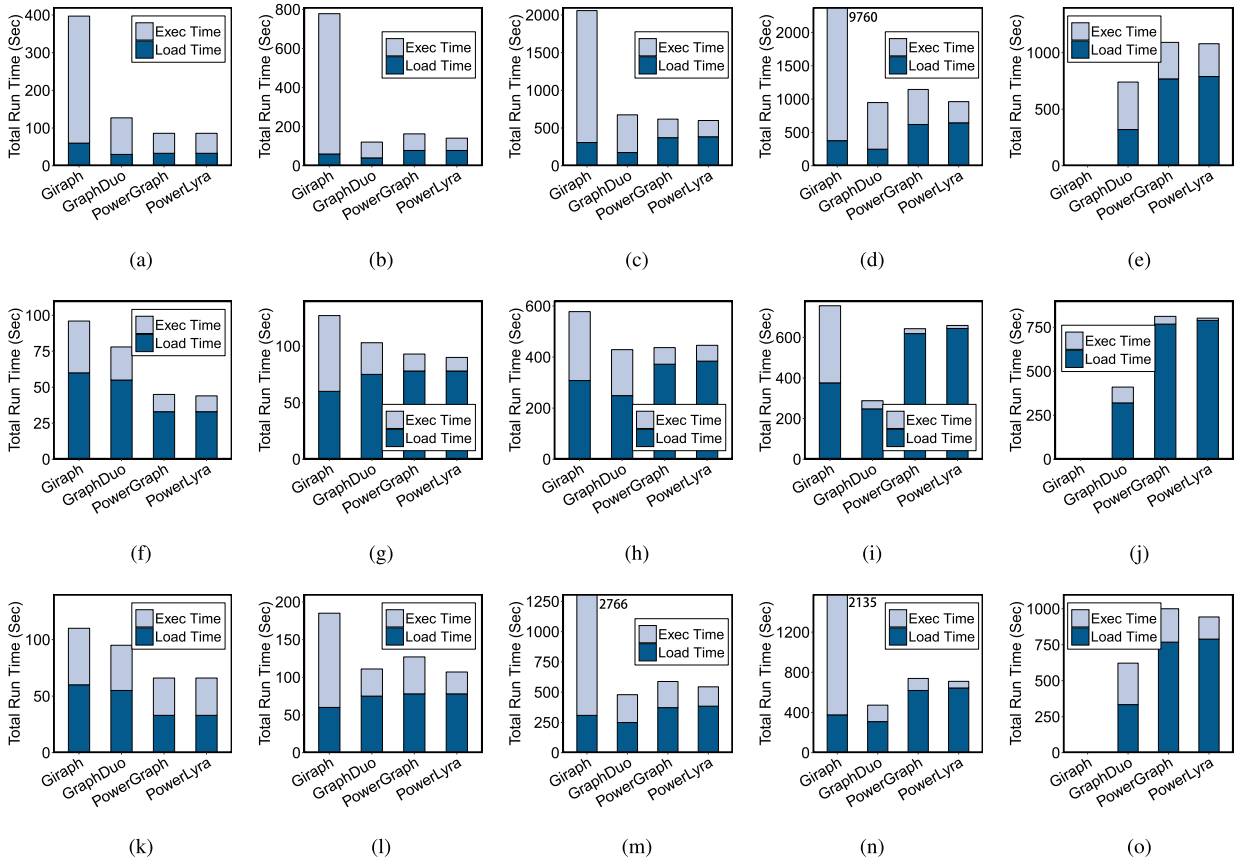


**FIGURE 12.** Performance comparison among Giraph, GraphDuo, PowerGraph and PowerLyra. For Giraph, it can not process the SK graph in our local cluster, so the time is not shown in this figure. (a) PageRank-LJ. (b) PageRank-Wiki. (c) PageRank-UK. (d) PageRank-Twitter. (e) PageRank-SK. (f) SSSP-LJ. (g) SSSP-Wiki. (h) SSSP-UK. (i) SSSP-Twitter. (j) SSSP-SK. (k) HashMin-LJ. (l) HashMin-Wiki. (m) HashMin-UK. (n) HashMin-Twitter. (o) HashMin-SK.

in memory. Such design can lead to low communication overhead, but lacks the support of fault tolerance. PowerLyra gets better performance than PowerGraph in most cases, due to its hybrid-cut graph partitioning and differentiated computing model.

## VI. LIMITATIONS AND FUTURE WORK

According to the experiment results, we prove that GraphDuo can outperform other Spark-based graph processing frameworks such as GraphX and GraphV. However, the performance is still lower than that of MPI-based systems. The main reason is the high communication overhead of data shuffle operation on Spark. The time taken by communication can be more than 60% of the total execution time in our experiments. In the worst case, the ratio can raise up to 90%. In our future work, we plan to extend our dual-model abstraction on MPI and high performance network hardwares, such as RDMA.

Moreover, the partitioning strategy adopted in GraphDuo requires two partitioning phases, which is not suitable for dynamic and fast-evolving graphs. In our future work, we plan to add an efficient streaming graph partitioning scheme to handle such graphs.

## VII. OTHER RELATED WORK

LFGraph [34] is a lightweight graph processing framework, which relies on cheap hash-based graph partitioning, a publish-subscribe and fetch-once communication model, a single-pass computation model and in-neighbor storage to provide efficient graph computation. It only supports unidirectional communication. Gemini [31] adopts a sparse-dense signal-slot abstraction, a chunk-based partitioning scheme, a dual representation scheme and other intra-node optimizations to build a scalable graph processing system on top of efficiency. It provides different execution strategies for computation rounds with sparse or dense edge set. PowerSwitch [35] provides a dynamic switching mechanism between synchronous and asynchronous execution models according to the properties of algorithms. GPS [18] is another system with the properties including a dynamic repartitioning scheme, an optimization that distributes adjacency lists of high-degree vertices across all computing nodes. PregePlus [15] also provides a mirroring technique to perform differentiated computation for low-degree and high-degree vertices, however, both of them does not consider efficient support for BVP algorithms. LightGraph [7] is based on an edge direction-aware graph partitioning strategy to isolate edges of different directions of each vertex, to eliminate avoidable communication in PageRank-like algorithms. GraphH [36] is another graph processing framework that considers diverse vertex traffic and heterogeneous network costs, and employs an adaptive edge migration strategy to avoid frequent communication over expensive network links. GraphD [37] considers out-of-core support in distributed graph processing, to enable large scale graph processing in small PC clusters. LazyGraph [38] employs a lazy data coherency approach called LazyAsync, which maintains vertex replicas to share the same global view only at some selected data coherency points, thus to reduce the number of global synchronization.

There are also lots of researches about parallel graph processing on single machine. GraphIn [39] is a dynamic graph analytics framework, which incrementally processes graphs using fixed-sized batches of updates. GraphP [40] is a novel software/hardware co-designed graph processing system based on HMC(Hybrid Memory Cube), with low communication overhead and energy consumption. Grazelle [41] is hybrid graph processing framework with a scheduler-aware interface for write traffic elimination, and a Vector-Sparse format to eliminate unaligned memory accesses during the graph computation. Wonderland [42] is a novel out-of-core graph processing system based on abstraction, which enables users to extract effective abstractions from the original graph, and provides abstraction-based propagation and scheduling schemes to speedup out-of-core graph processing.

For system designing, Gao *et al.* [43] suggested that a graph processing system should consider five keys, including graph distribution, on-disk graph organization, programming model, message model and synchronization policy, which provides good for us when designing GraphDuo.

## VIII. CONCLUSION

In this paper, we propose a new Spark-based graph processing framework called GraphDuo, which provides two different computing models for algorithms with different value propagation behaviors. A balanced vertex-centric computing model is provided for UVP algorithms, which only needs one computation step in each iteration, and provides efficient and balanced communication mechanisms. An optimized edge-centric computing model is provided for BVP algorithms, which only needs two computing steps and two communication operations in each iteration. Combined with a cheap degree-based graph partitioner, a locality-aware graph layout and other optimization techniques, GraphDuo ensures low communication overhead, low computation overhead, small memory footprint and good load balance for two types of algorithms. According to the experiment results, GraphDuo outperforms GraphX by 1.11×-6.56×, with much less resource consumption. Moreover, when processing large scale graphs, such as Twitter and SK, GraphDuo even outperforms PowerGraph and PowerLyra by at most 2.36×, due to the low cost graph partitioning and efficient computation models.

## REFERENCES

[1] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[2] 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," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. OSDI*, vol. 12, 2012, p. 2.

[4] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 215–226.
[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. Oper. Syst. Design Implement.*, 2014, pp. 599–613.
[6] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.
[7] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "LightGraph: Lighten communication in distributed graph-parallel processing," in *Proc. IEEE Int. Congr. Big Data*, Jun./Jul. 2014, pp. 717–724.
[8] Y. Koren, "Factorization meets the neighborhood: A multifaceted collaborative filtering model," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 426–434.
[9] S. N. A. Project. *Stanford Large Network Dataset Collection*. Accessed: 2018. [Online]. Available: http://snap.stanford.edu/data/
[10] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, no. 1, pp. 29–123, 2009.
[11] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proc. 20th Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2006, p. 124.
[12] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Networked Syst. Design Implement.* Berkeley, CA, USA: USENIX Association, 2012, pp. 1–14.
[13] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endowment*, vol. 7, no. 7, pp. 577–588, 2014.
[14] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "HADI: Mining radii of large graphs," *ACM Trans. Knowl. Discovery from Data*, vol. 5, no. 2, 2011, Art. no. 8.
[15] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015, pp. 1307–1317, doi: 10.1145/2736277.2741096.
[16] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proc. Hadoop Summit*. Santa Clara, CA, USA, vol. 11, no. 3. 2011, pp. 5–9.
[17] X. Tian, Y. Guo, J. Zhan, and L. Wang, "Towards memory and computation efficient graph processing on spark," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 375–382.
[18] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 25th Int. Conf. Sci.Stat. Database Manage.*, 2013, Art. no. 22.
[19] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 219–228.
[20] (Apr. 2017). *Wikipedia Links, English Network Dataset—KONECT*. [Online]. Available: http://konect.uni-koblenz.de/networks/wikipedia_link_en
[21] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf. (WWW)*. Manhattan, NY, USA: ACM Press, 2004, pp. 595–601
[22] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web*. New York, NY, USA: ACM Press, 2011, pp. 587–596.
[23] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. Int. Conf. Weblogs Social Media*, 2010, pp. 10–17.
[24] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, "An experimental comparison of partitioning strategies in distributed graph processing," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 493–504, 2017.
[25] *Hadoop*. Accessed: 2018. [Online]. Available: http://hadoop.apache.org/
[26] N. Jain, G. Liao, and T. L. Willke, "GraphBuilder: Scalable graph ETL framework," in *Proc. Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 4.
[27] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-based partitioning for power-law graphs," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, 2015, pp. 243–252.
[28] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1673–1681.
[29] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma, "Finding connected components in map-reduce in logarithmic rounds," in *Proc. IEEE Int. Conf. Data Eng.*, Apr. 2013, pp. 50–61.
[30] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.
[31] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. Oper. Syst. Design Implement.* , 2016, pp. 301–316.
[32] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.
[33] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *Proc. Eur. Conf. Parallel Process.*, 2015, pp. 438–450.
[34] I. Hoque and I. Gupta, "LFGraph: Simple and fast distributed graph analytics," in *Proc. 1st ACM SIGOPS Conf. Timely Results Oper. Syst.*, 2013, Art. no. 9.
[35] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to fuse for distributed graph-parallel computation," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.* New York, NY, USA: ACM, 2015, pp. 194–204, doi: 10.1145/2688500.2688508.
[36] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, "GrapH: Traffic-aware graph processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1289–1302, Jun. 2018.
[37] D. Yan *et al.*, "GraphD: Distributed vertex-centric graph processing beyond the memory limit," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 99–114, Jan. 2018.
[38] L. Wang *et al.*, "Lazygraph: Lazy data coherency for replicas in distributed graph-parallel computation," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 276–289.
[39] D. Sengupta *et al.*, "GraphIn: An online high performance incremental graph processing framework," in *Proc. Eur. Conf. Parallel Process.* Basel, Switzerland: Springer, 2016, pp. 319–333.
[40] M. Zhang *et al.*, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. 24th IEEE Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 544–557.
[41] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2018, pp. 246–260.
[42] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proc. 23rd Int. Conf. Architectural Support for Program. Lang. Oper. Syst.* New York, NY, USA: ACM, 2018, pp. 608–621.
[43] Y. Gao, W. Zhou, J. Han, D. Meng, Z. Zhang, and Z. Xu, "An evaluation and analysis of graph processing frameworks on five key issues," in *Proc. 12th ACM Int. Conf. Computing Frontiers*, New York, NY, USA, 2015, Art. no. 11, doi: 10.1145/2742854.2742884.

**XINHUI TIAN** received the B.S. degree from the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, in 2011. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing.

His research interests include big data, distributed system, and large-scale graph processing.

**JIANFENG ZHAN** received the B.S. degree in civil engineering and the M.S. degree in solid mechanics from Southwest Jiaotong University, Chengdu, China, in 1996 and 1999, respectively, and the Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2002.

Since 2012, he has been a Full Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. He has authored over 150 articles. He recently focuses on different aspects of datacenter computing. He was the Founder of the BPOE Workshop, focusing on big data benchmarks, performance optimization, and emerging hardware.