Future Generation Computer Systems 89 (2018) 698–712

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A highly cost-effective task scheduling strategy for very large graph computation

Yongli Cheng [a,b,*], Fang Wang [b], Hong Jiang [c], Yu Hua [b], Dan Feng [b], Yunxiang Wu [b], Tingwei Zhu [b], Wenzhong Guo [a,d,e]

[a] College of Mathematics and Computer Science, FuZhou University, China
[b] School of Computer, Huazhong University of Science and Technology, WuHan, China
[c] Department of Computer Science & Engineering, University of Texas at Arlington, USA
[d] Key Lab of Spatial Data Mining & Info. Sharing, Min. of Education, Fuzhou, China
[e] Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou, China

HIGHLIGHTS

- The reduced the number of supersteps due to the high convergence speed.
- The eliminated synchronization overheads due to the pipeline-based task scheduling.
- The high flexibility due to the network ecosystem friendliness.

ARTICLE INFO

Article history:
Received 18 October 2017
Received in revised form 19 June 2018
Accepted 10 July 2018
Available online 18 July 2018

Keywords:
Graph computation
Cost-effectiveness
Very large graphs

ABSTRACT

Existing distributed graph-processing frameworks, e.g., Pregel, GPS and Giraph, handle large-scale graphs in the memory of clusters built of commodity compute nodes for better scalability and performance. While capable of scaling out according to the size of graphs up to thousands of compute nodes, for graphs beyond a certain size, these frameworks would usually require investments of machines that are either beyond the financial capability of or unprofitable for most small and medium-sized organizations, making the deployment of their large-scale graph-computing jobs difficult if not impossible. At the other end of the spectrum of graph-processing frameworks research, the single-node disk-based graph-computing frameworks, such as GraphChi and XStream, handle large-scale graphs on just one commodity computer, leading to high efficiency in the use of hardware but at the cost of low user performance and limited scalability. Motivated by this dichotomy, in this paper we propose a pipeline-based task scheduling strategy with high cost-effectiveness. We use this scheduling strategy to design and implement a distributed disk-based graph-processing framework, called DD-Graph, that can process very large graphs with trillions of edges on a small cluster while achieving the high performance of existing distributed in-memory graph-processing frameworks. The evaluation of DD-Graph prototype, driven by very large graph datasets, shows that it saves 73% of GPS' hardware costs while running 1.34x faster than GPS.

1. Introduction

In recent years, there has been a recent surge of interest in extracting valuable information from graph structures in both academia and industry. Today, in many problem domains that require graph computation, the graphs are becoming larger than ever before. These graphs, such as social networks, can have billions of vertices and up to trillions of edges [1,2].

Due to the fact that many graph algorithms exhibit irregular access patterns [3], most graph-processing frameworks require that the graphs fit entirely in memory [4–8], necessitating either a supercomputer or a very large cluster to process very large graphs [4,9,10]. The excessive investment of a very large cluster or a supercomputer discourages and possibly prevents many small and medium-sized organizations from deploying their large-scale graph-computing jobs.

In order to reduce hardware costs and improve efficiency, several graph-processing frameworks, e.g., GraphChi [11] and XStream [12], have been proposed to process graphs with billions of edges on just one commodity computer, by relying on secondary storage [11,12]. However, the performance of these frameworks is limited by the limited secondary storage bandwidth of a single compute node [13] and the significant difference in the access

* Correspondence to: 2 Xue Yuan Road, University Town, Fuzhou, Fujian 350108.
E-mail address: chengyongli@hust.edu.cn (Y. Cheng).

https://doi.org/10.1016/j.future.2018.07.010
0167-739X/

speeds between secondary storage and main-memory [14]. Furthermore, the limited amount of storage of a single commodity computer can potentially limit the scale of the processed graphs, since graphs continue to grow rapidly in size [10].

The key difference between the in-memory graph-processing frameworks and single-node secondary storage based graph-processing frameworks lies in the trade-off between the hardware cost and performance, with the former trading off hardware cost for performance while the latter doing the exact opposite. In this paper, we propose a *distributed disk-based graph-processing framework*, called DD-Graph that has the salient feature of both the low hardware cost and high performance.

Distributed disk-based graph-processing frameworks target efficient big graph processing with a small cluster of commodity PCs that is affordable to most common users. However, it is challenging to design an efficient distributed disk-based graph-processing system since the total resources of a small cluster are limited. This is also evidenced by most recent research results, such as Chaos and Pregelix [10,15]. GraphD [16] is proposed recently to hide the disk I/O cost by overlapping the disk I/O with the communication inside each compute node of the small cluster, improving the utilization of the resources in each compute node. However, this solution is efficient only for the network ecosystem with low bandwidth [16].

DD-Graph is different from several recently proposed distributed disk-based graph-processing frameworks [10,15–17], which improves the overall runtime of the graph-computing job significantly by using the pipeline-based task scheduling strategy that provides three key features as follows.

1. *High convergence speed*. By pipelining the tasks of the graph-computing job, our scheduling strategy can reduce the number of supersteps of the graph-computing job significantly. Since when computation Ci of task i has done, the computation Ci $+1$ of task i $+1$ can use the computation result immediately. A task is defined as the execution process of a partition in one superstep. This is important since, in distributed disk-based graph-processing frameworks, the runtime of each superstep is usually time-consuming when processing a very large graph.
2. *Eliminated synchronization overheads*. During the execution process of pipeline-based task scheduling strategy, there is not a clear division between any two consecutive supersteps. Furthermore, when the stage of loading partition has finished, the compute node can immediately execute the computation stage of task t currently being launched if the computation stage of task $t-1$ has finished and the computation result of task $t-1$ has arrived, eliminating the costly synchronization overheads.
3. *Network ecosystem friendliness*. The performance of Chaos [10] and Pregelix [15] relies heavily on the assumption that network bandwidth far outstrips storage bandwidth. At the other end of the spectrum of distributed disk-based graph-processing frameworks research, GraphD [16] is designed for the low-bandwidth networks. Thus, the disk I/O can be hidden by the communication. However, DD-Graph is network ecosystem friendly. It can hide almost the entire communication time and the full disk I/O time by overlapping the disk I/O and communication of each compute node with the computations of other compute nodes, if a cluster with an appropriate scale is available. DD-Graph also provides two optimizations to further improve the communication and the disk I/O efficiencies, as detailed in Sections 4.1 and 4.3.

The rest of the paper is structured as follows. Background and motivation are presented in Section 2. Section 3 introduces the pipeline-based task scheduling strategy. System design and optimization are presented in Section 4. Experimental evaluations of the DD-Graph prototype are presented in Section 5. We discuss the related work in Section 6 and conclude the paper in Section 7.

2. Background and motivation

In this section, we first present a brief introduction to distributed in-memory graph-processing frameworks. We then discuss the single-node disk-based graph frameworks. The significantly different characteristic between the two types of graph-processing frameworks help motivate us to propose a distributed disk-based graph-processing framework that has the characteristics of both the low hardware cost and high performance.

2.1. Pregel-like graph-processing frameworks

Pregel-like graph-processing frameworks, such as Pregel [4], GPS [5] and Giraph,[1] adopt a vertex-centric computation model. In these frameworks, an input graph is divided into partitions, each of which resides in the memory of a compute node during the execution process. A graph-computing job proceeds in a sequence of iterations (supersteps), which terminates when all vertices vote to stop the computation. In each iteration, a user-defined **vertex-program(v)** function is invoked for each vertex **v**, conceptually in parallel. Inside the **vertex-program(v)** function, the state of vertex **v** is updated by using the old state and the incoming messages that were sent to **v** in the previous iteration; then **vertex-program(v)** function generates messages based on the new state of **v** and sends them to **v**'s neighbors. At the end of each iteration, all compute nodes synchronize to ensure that all messages have been received successfully.

Due to the "think-like-a-vertex" philosophy, these frameworks are very user friendly for coding and debugging parallel graph algorithms. Furthermore, in recent years, several in-memory distributed graph-processing framework, such as Gemini [18] and BlitzG [19], have improved the system performance significantly by overcoming the performance bottleneck caused by the fine-grained and high-frequency communication. These frameworks are very useful for high-end users to deploy their large-scale and time-sensitive graph-computing job. For high-end users, such as big companies and banks, they usually have a large number of compute nodes necessitated by the combined requirement of large aggregate memory space with respect to the size of the large-scale graph and high system performance. However, high performance is not always the first thing. For most small to medium size companies and most research institutes, they usually need to meet the increasing rapidly needs of processing large-scale graph computations in a reasonable amount of time. In this case, a very large cluster is not easily accessible in most small to medium size companies and most research institutes, possibly preventing many organizations from deploying their large-scale graph-computing jobs.

2.2. Single-node graph-processing frameworks

In recent years, several graph-processing frameworks, such as GraphChi [11] and Xstream [12], have been proposed to process large-scale graphs on just a single commodity computer. However, due to the costly I/O latency, they routinely suffer from poor performance, leading to a long time used by users to wait for the graph-computing results. In order to address this problem, some techniques are proposed to reduce the I/O latency. GridGraph [20] breaks graphs into 1D-partitioned vertex chunks and

[1] Apache Giraph: http://giraph.apache.org.

2D-partitioned edge blocks using a first fine-grained level partitioning in preprocessing. A second coarse-grained level partitioning is applied in runtime. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reduce the I/O amount required for computation. FlashGraph [21] implements a semi-external memory graph engine which stores vertex states in memory and adjacency lists on SSDs, thus reducing the performance gap between in-memory and out-of-core graph processing. [22] provides a general optimization for disk-based graph processing, which removes unnecessary I/O by using a dynamic graph partitioning method.

These techniques improve the performance of single-node graph-processing frameworks by significantly reducing the I/O latency. However, in the era of big data, single-node graph-processing frameworks still have to face three challenges as follows. First, due to the limited amount of storage that can be attached to a single machine, these frameworks can only process graphs with billions of edges, while graphs of interest continue to grow rapidly in size [11,12,23]. Second, in spite of the improved I/O efficiency, the I/O latency is still the main contributor to the overall runtime due to the limited secondary storage bandwidth of a single compute node and the significant difference in the access speeds between the secondary storage and main memory [13,14].

2.3. Motivation

The discussion and analysis above clearly reveal the significantly different features between existing distributed in-memory graph-processing frameworks and the single-node secondary storage based graph-processing frameworks. The former can offer high performance but suffer from high hardware cost. Although the latter has a clear advantage of low hardware cost by processing large-scale graphs on just a single commodity computer, they suffer from the poor performance and the limited capacity of the storage system on single compute node [10].

We further obtain an important observation through an experiment contrasting GPS and GraphChi. GPS runs one iteration of the PageRank graph algorithm on the Twitter-2010 graph on systems of different sizes respectively, with the number of compute nodes ranging from 8 to 32. Note that GPS needs at least 8 compute nodes to run this job. Experimental results indicate that the runtime of GPS is reduced gradually as the number of compute nodes increases and reaches its minimum value at 30 compute nodes. However, the runtime is not improved further when more dedicated compute nodes are added. The limited scalability of GPS stems mainly from its inefficient fined-grained communication model [11,19,24–26], i.e., message-passing. Fig. 1(a) shows execution process of the peak-performance scale case. GPS distributes the vertices of the input graph into 30 compute nodes that are executed in parallel. Firstly, all compute nodes start at time T0 simultaneously. Then, each compute node executes its vertex-programs. The graph-computing job ends at T1 when the slowest compute node (Node 0) has finished. In this case, the runtime measured is 27 s.

GraphChi runs one iteration of the PageRank graph algorithm with the Twitter-2010 graph on one compute node. It divides the input graph into 8 partitions. As shown in Fig. 1(b), the graph-computing job starts at time T0. Then the partitions are executed sequentially. Each execution process of a partition consists of three stages: loading partition, computation, and saving results. The graph-computing job ends at T1 when the results of partition7's execution have been saved, consuming 280 s.

An important observation: We measure the time spent on computation of GraphChi by adding up the computation time of each partition. Similarly, the time spent on disk I/O of GraphChi is measured by adding up the times spent on loading partition and saving results of each partition. An important phenomenon is observed from Fig. 2, i.e., the time spent on computation of GraphChi [11] accounts for only 9% of the overall run time while the time spent on disk I/O is responsible for up to 91% of the overall run time. Furthermore, the computation time (25.2 s) of GraphChi is slightly less than the overall run time (27 s) of GPS. This observation motivates us to ask a question: *Can we avoid the I/O latency of single-node disk-based graph processing frameworks by using a small cluster, and thus almost reduce the overall runtime to the total time spent on the computation stage of each partition, simultaneously achieving cost-saving and high system performance?*

In this paper, we propose a distributed disk-based graph-processing framework, called DD-Graph, that is cost-effective and of high-performance. We describe several important goals of DD-Graph as follows.

Processing Very Large Graphs on a Small Cluster. DD-Graph divides the input graph into partitions, each of which can be loaded in the memory of each compute node. For a very large graph, the number of partitions is usually large. Partitions are assigned to a limited number of compute nodes of a small cluster, each of which manages a number of partitions. At any given time of the execution process, only one partition resides in the memory of any compute node. Hence, DD-Graph can handle a very large graph as long as the aggregate disk capacity of the compute nodes can accommodate the input graph. However, single-node secondary storage based graph-processing frameworks can only process graphs with a limited size due to the limited amount of storage of a single commodity computer. This is important since the graphs are becoming larger than ever before and continue to grow rapidly in size [10]. Moreover, for existing distributed in-memory graph-processing frameworks, they need to load the full input graph into the memory of the compute nodes, necessitating either a supercomputer or a large cluster to process very large graphs.

Reduced Number of Supersteps. The distributed disk-based graph-processing frameworks [10,15–17] are built on a small cluster. Compared with a large one, the total resources of a small cluster are limited. In this case, in order to improve the performance of a distributed disk-based graph-processing framework, one solution is to reduce the number of supersteps that can guarantee the graph algorithm to meet its convergence condition. By pipelining the tasks of the graph-computing job, DD-Graph can reduce the number of supersteps of the graph-processing job significantly. Since when computation Ci of task i has done, the computation $Ci+1$ of task $i+1$ can use the computation result immediately. However, in other distributed disk-based graph-processing systems, such as GraphD [16], the computation results of partitions cannot be used until next superstep.

High Utilization of the Resources of the Whole Cluster. Another solution is to improve the utilization of the resources of the whole cluster. DD-Graph achieves this goal by using four measures as follows. First, by using our pipeline-based task scheduling strategy, there is not a clear division between any two consecutive supersteps. When the stage of loading partition has finished, the compute node can immediately execute the computation stage of task t currently being launched if the computation stage of task $t-1$ has finished and the computation result of task $t-1$ has arrived, eliminating the costly synchronization overheads. Second, DD-Graph can hide almost the entire communication time and the full disk I/O time by overlapping the disk I/O and communication of each compute node with the computations of other compute nodes, if a cluster with an appropriate scale is available. Third, DD-Graph introduces an edge-data-block based data representation to significantly reduce the communication latency. It eliminates the extra communication volume used by each message to carry the vertex name. The size of the vertex names is usually responsible

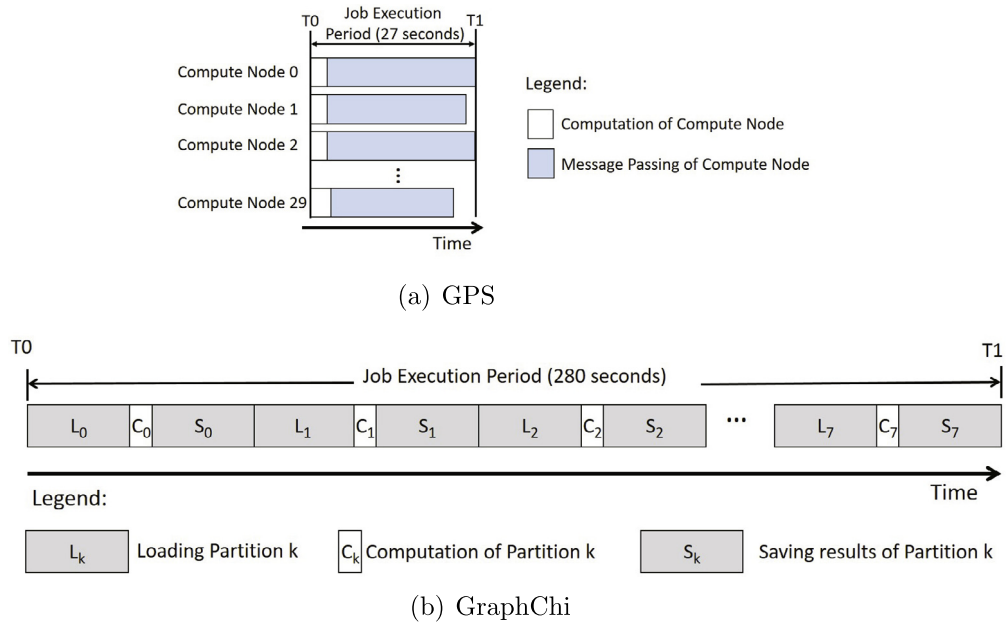


Fig. 1. Execution process of one iteration.

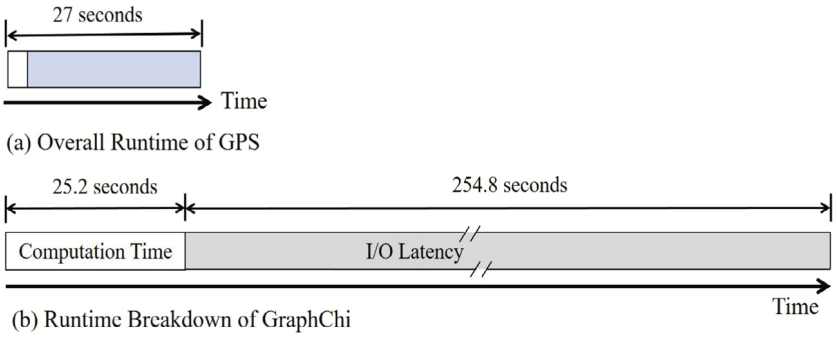


Fig. 2. Computation time of GraphChi VS. Overall runtime of GPS.

for up to 67% of the overall communication volume, as discussed in Section 4. This is important, especially where a high-bandwidth network is not available. Finally, DD-Graph also provides one optimization to further improve the disk I/O efficiency, as detailed in Section 4.3.

3. Pipeline-based task scheduling strategy

In this section, we present the pipeline-based task scheduling strategy that is highly effective for distributed disk-based graph computation, due to the three key features of *high convergence speed*, *eliminated synchronization overheads* and *network ecosystem friendliness*.

3.1. Task assignment

The system architecture consists of a master node and M compute nodes. The master node schedules the tasks of the graph-computing job. Each compute node is responsible for managing and executing its assigned partitions. Without the loss of generality, we assume that there are three compute nodes, a graph is divided into six partitions, and a graph-computing job consists of two iterations. Each partition p is managed by the compute node x, where $x = (p \bmod M)$. As shown in Fig. 3, partitions P_0 and P_3 are managed by compute node 0, partitions P_1 and P_4 are managed by compute node 1, and partitions P_2 and P_5 are managed by compute node 2. In order to avoid an imbalance in the number of partitions assigned to different compute nodes, we impose a constraint condition that the number of partitions P is divisible by M.

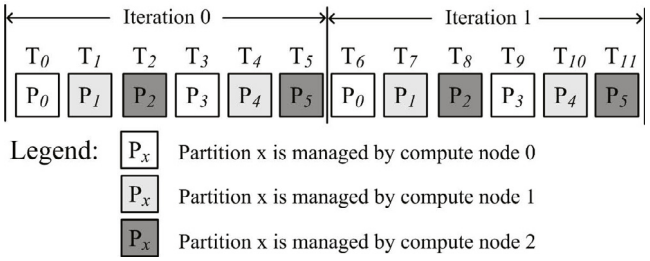


Fig. 3. Relationships among compute nodes, partitions, iterations and tasks.

A task is defined as the execution process of a partition in one iteration, therefore there are 12 tasks in total, ordered as $T_0, T_1, \ldots, T_{11}$. A task is decomposed into three stages: (1) loading partition, (2) computation, and (3) communication and saving results to the

disk. As shown in Fig. 3, the tasks T_0, T_3, T_6, T_9 are executed by compute node 0, the tasks T_1, T_4, T_7, T_{10} are executed by compute node 1, and the tasks T_2, T_5, T_8, T_{11} are executed by compute node 2. Each compute node employs a *task queue* to manage its tasks.

3.2. Job execution process

As shown in Fig. 4, all compute nodes start at time T0. Each compute node launches its first task from its *task queue* and loads the partition of that task. When the stage of loading partition has finished, the compute node can immediately execute the computation stage of task t currently being launched if the computation stage of task $t-1$ has finished and the computation result of task $t-1$ has arrived. Otherwise, task t should wait for a short time period for the arrival of the computation result sent by task $t-1$. The task T_0 is a special case because it is the first one. The short waiting time period indicates that the computation stage of task $t-1$ has not finished. This can lead to the waste of computational resources, resulting in efficiency loss, as discussed in the following subsections.

In the computation stage, a user-defined **Update(v)** function is invoked for each vertex **v** in the partition in parallel. Inside **Update(v)**, the vertex **v** updates its state by its in-edge values and then updates its out-edge values. The in-edge values of vertex **v** were updated by the source vertices of the in-edges in the previous $P-1$ tasks, and the out-edge values of vertex **v** will be used by the destination vertices of the out-edges in the subsequent *P-1* tasks.

Then, the compute node starts the communication and result-saving processes simultaneously. In the communication process, the compute node sends different computation result to the subsequent $P-1$ tasks sequentially in order. When task t begins to execute the computation stage, it has received N different computation results for its partition from the previous N tasks, where $N = \min(t, P-1)$. In the result-saving process, the compute node saves the local computation result of the partition currently being executed to the disk. Finally, the compute node either repeats for the next task, or stops when it is demanded by the master node or the *task queue* is empty.

This task scheduling strategy can run a fixed number of iterations by assigning *NI* a fixed number. *NI* is a variable used to control the number of iterations. However, in most cases, graph-computing jobs stop according to the convergence conditions. In this case, users do not need to input the number of iterations. A very large default value (such as 9999) of *NI* is automatically assigned, which is set in the system configure file. In this case, the graph-computing job proceeds in an uncertain number of iterations until the convergence condition of the graph-computing job is met.

3.3. Improved convergence speed

By pipelining the tasks of the graph-computing job, our scheduling strategy can reduce the number of supersteps of the graph-processing job significantly. Since when computation C_i of task i has done, the computation C_{i+1} of task $i+1$ can use the computation result immediately, as shown in Fig. 4. However, in other distributed disk-based graph-processing frameworks, such as GraphD [16], implement synchronous execution model. In order to guarantee the determinism of the graph-computing job, the computation results of partitions cannot be used until next superstep. This execution process of our scheduling strategy can not only guarantee the determinism of graph-computing jobs but also accelerate the convergence of graph algorithms [11]. Experimental results show that DD-Graph achieves convergence by the 31th superstep when running Pagerank with Twitter-2010 graph. However, convergence is not achieved until the 39th superstep in GraphD. This is important since, in distributed disk-based graph-processing frameworks, the runtime of each superstep is usually time-consuming when processing a very large graph. For example, Chaos uses 3179.5 s to run only one superstep of PageRank with Hyperlink-2012 graph.

Some in-memory distributed graph-processing frameworks, such as PowerGraph [8], allow vertices to read the new status of their neighbors in the same superstep, aiming to improve the convergence speed. In order to guarantee data consistency, Powergraph prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol, resulting in costly lock overheads. However, during the execution process of the pipeline-based task scheduling strategy, the computation stages of the tasks are executed sequentially according to the sequence number of task. In the computation stage of each task, the vertices are executed in parallel. Hence, the data consistency can be guaranteed without lock overheads.

3.4. Network ecosystem friendliness

Some distributed disk-based graph-processing frameworks can obtain high performance only when machines in the cluster are connected by high-speed network (e.g., 40GigE). For example, the performance of Chaos and Pregelix relies heavily on the assumption that network bandwidth far outstrips storage bandwidth [10].

At the other end of the spectrum of distributed disk-based graph-processing frameworks research, GraphD [16] is designed for the network ecosystem with low bandwidth. It assumes the compute nodes connected by a low-bandwidth network. In this case, the communication time is longer than the disk I/O time. Thus, the disk I/O can be hidden by the communication. However, nowadays, high-bandwidth networks are easily accessible. In this case, instead of communication, the disk I/O will become the bottleneck of system performance.

DD-Graph is network ecosystem friendly, due to the reasons as follows. The first one is the *2-Level Hierarchical Overlapping*. Like GraphD, in each compute node, the communication is overlapped with the disk I/O, thus improving the utilization of the resources of each compute node. Unlike GraphD, our pipeline-based task scheduling strategy further overlaps the disk I/O and communication of each compute node with the computations of other compute nodes. This design is network ecosystem friendly. In different network ecosystems, it can hide almost the entire communication time and the full disk I/O time by overlapping the disk I/O and communication of each compute node with the computations of other compute nodes, if a cluster with an appropriate scale is available. By hiding communication time and disk I/O time, the overall runtime of the graph-computing job can be reduced almost to the time spent on the computation stages, as illustrated in Fig. 5. This is efficient for the distributed disk-based graph-processing framework with a small cluster, as shown in experiment contrasting GPS and GraphChi of Section 2.3. The second one is that DD-Graph introduces two optimizations to further reduce the communication and disk costs, thus improving the efficiency of the whole cluster, as detailed in Sections 4.1 and 4.3.

The experimental results show that DD-Graph is effective for not only the high-bandwidth but also the low-bandwidth networks.

3.5. Eliminated synchronization overheads

There is a costly global synchronization between any two consecutive supersteps in existing distributed disk-based graph-processing frameworks [10,16]. For example, GraphD [16] implements Pregel's synchronous execution model. In this case, there is a barrier at the end of each superstep to ensure that all generated

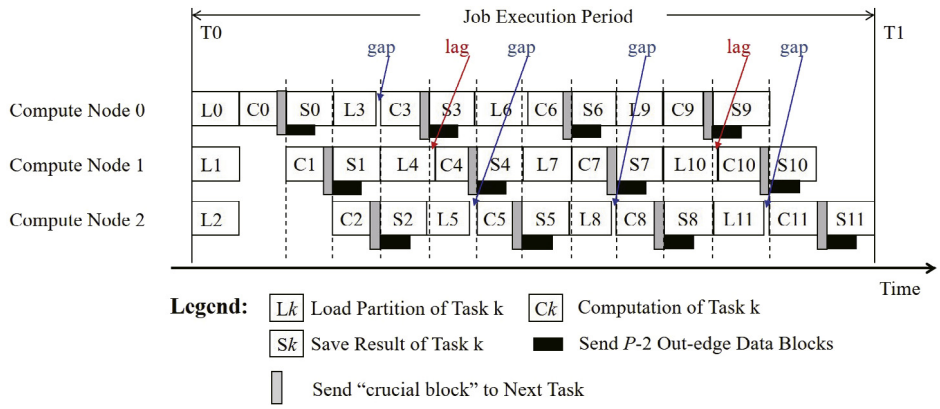


Fig. 4. Job Execution Process. A *gap* represents an idle period between the loading partition stage and the computation stage of a task. A *lag* signifies a short time span between two adjacent computations.

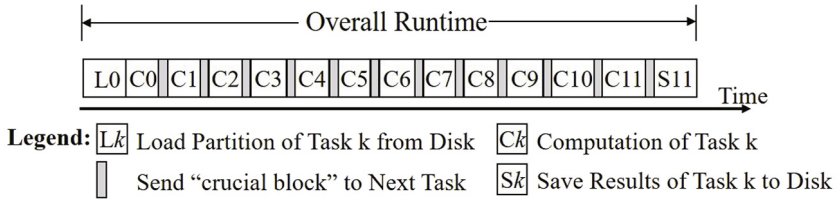


Fig. 5. An illustration of the overall runtime as a result of DD-Graph's minimization of communication cost and hiding of disk I/O latency.

messages are delivered to their destination vertices before the next superstep starts, resulting in costly synchronization overheads. In order to mitigate the costly synchronization overheads, GraphD introduces a method that allows the compute node to start next superstep when this compute node has received all the messages. Chaos [10] applies a work stealing technique to mitigate the workload imbalance among the compute nodes, aiming to reduce the global synchronization overheads. However, this technique allows multiple machines to execute a single partition, resulting in costly extra communication overheads for data migration [10].

Unlike existing distributed disk-based graph-processing frameworks, there is not a clear division between any two consecutive supersteps in our pipeline-based task scheduling strategy. For each task t, when the stage of loading partition has finished, the compute node can immediately execute the computation stage of task t currently being launched if the computation stage of task $t-1$ has finished and the computation result of task $t-1$ has arrived, eliminating the synchronization overheads.

4. System design

In this section, we present the design of DD-Graph, i.e., a new distributed disk-based graph-processing framework. Key components of the DD-Graph framework are detailed in the subsections that follow.

4.1. Edge-data-block based data representation

The high performance of DD-Graph stems mainly from the pipeline-based task scheduling strategy that hides the latencies of communication and disk I/O by overlapping the communication and disk I/O times of each compute node with the times of computations of other compute nodes. In order to further improve communication efficiency, DD-Graph also introduces the edge-data-block based data representation that is one of the key techniques of LCC-Graph [27], one of our previous work accepted by IWQoS conference of 2016. LCC-Graph is a distributed in-memory graph-processing framework that is designed for high-end users to deploy their large-scale and time-sensitive graph-computing job, where a large cluster is easily accessible. The key technique of LCC-Graph is the LCC-BSP computation model that decomposes each superstep into two distinct steps of computation and communication. LCC-Graph deploys edge-block based data representation to speed up the communication process. This data representation is described as follows.

In the preprocessing phase, the input graph is divided into partitions. The edge values of each partition are organized into a local-edge data block, $P-1$ remote in-edge data blocks and $P-1$ remote out-edge data blocks intelligently, where P is the number of partitions. Each remote out-edge data block includes the values of all the out-edges whose destination vertices reside in one of the remote partitions. Each remote in-edge data block includes the values of all the in-edges whose source vertices reside in one of the remote partitions. Since the out-edges of a given vertex are the in-edges of its neighbors, each remote out-edge data block is an exact copy of a remote in-edge data block of one of the other partitions. The local-edge data block is a special case because both the source vertices and destination vertices of the edges are in the same partition.

This data representation provides three salient advantages. First, each out-edge data block is an exact copy of an in-edge data block of a remote partition, that is, each edge value has the same position in the two copies. Thus, the edge values indexed by vertices can be used by the vertices directly in both sender side and the receiver side, eliminating the high extra volume of communication. The extra volume of communication is used by Bulk Synchronous Parallel (BSP) computation model [28] based distributed graph-processing frameworks to carry the destination vertex name on each message. For many graph algorithms, the extra communication volume due to the destination vertex names is responsible for 67% of the overall communication volume. Second, since the edge values in the edge data blocks are indexed by vertices directly,

the received out-edge data blocks can be used by the receiver side directly, avoiding the parsing overhead that is used by the *message parser* to parse the message batches in BSP-based distributed graph-processing frameworks [5]. Third, like the *message buffering* technique of BSP-based distributed graph-processing frameworks [5], by sending full remote out-edge data blocks, it enables the network bandwidth capacity to be efficiently utilized.

4.2. Communication efficiency

DD-Graph introduces two methods to minimize its communication costs. First, DD-Graph overlaps the communication of a compute node with the computations of other compute nodes. When a task k has finished its computation stage, it will send $P-1$ out-edge data blocks to the other $P-1$ partitions, including $P \times (M-1)/M$ remote partitions and $P/M-1$ local partitions. The out-edge data block to the partition of task $k+1$ should be sent first, because this data block triggers the execution of the computation stage of task $k+1$. This out-edge data block is defined as the "crucial block". DD-Graph overlaps the remaining $P \times (M-1)/M-1$ block transfers with the computations of subsequent tasks, effectively eliminating $(P \times (M-1)-M)/(P \times (M-1))$ communication latency.

Second, our efficient communication model speeds up the transfer of each "crucial block" greatly, further improving the system performance. As discussed in Section 4.1, the edge-data-block based data representation reduces communication cost between any pair of compute nodes by avoiding extra communication volume and eliminating the extra overhead at the receiver side. Furthermore, like BSP-based distributed graph-processing frameworks, our coarse-grained communication model can fully utilize the network bandwidth, resulting in very high communication efficiency.

The efficiency of edge-block based data representation is reduced for the graph-processing algorithms with a few inter-vertex interactions. In order to address this problem, we also have implemented the message passing communication model. However, we abandoned this scheme finally, due to its effectiveness. The reasons are twofold. First, this data representation is effective enough for most graph algorithms, which can avoid 67% of the overall communication volume. Second, even for the graph algorithms with a few inter-vertex interactions, the effectiveness of message passing communication model also is limited, due to extra communication volume and the extra overhead at the receiver side.

4.3. Disk I/O optimization

DD-Graph also introduces one measure to further improve the disk I/O efficiency. In DD-Graph, the input graph is partitioned a number of subgraphs. Each subgraph with vertices and edges can be fully loaded to the memory of one compute node. Thus, each compute node can send computation results to other compute nodes in memory, without involving disk I/O for communication. However, GraphD [16] loads vertices into memory only. During the process of computation, the generated messages are first sent to local disk. In the communication process, the messages need to been loaded into memory where they are then sent to network. This results in extra disk I/O for communication.

4.4. Balancing efficiency and performance

There are *gaps* and *lags* during the execution process in DD-Graph, as shown in Fig. 4. A *gap* represents an idle period between the loading partition stage and the computation stage of a task. For example, while L_3, the partition loading for the next task (Task 3), has finished, C_2, the computation of the current task (Task 2) has not finished, meaning that compute node 0 will be idle for a short period in which the computation stage of Task 3 waits for the computation stage of Task 2 to complete and the crucial block from Task 2. A *lag* signifies a short time span between two adjacent computations. For example, C_3, the computation of the current task (Task 3), has finished, but L_4, the partition loading for the next task (Task 4), has not finished, delaying the start of C_4. While *lags* bring extra latency between two adjacent computations and thus lengthen the overall run time, *gaps* result in the waste of computational resources and lead to efficiency loss.

More compute nodes bring more and longer *gaps* but fewer and shorter *lags*, resulting in performance improvement but efficiency loss. The reason is that the longer disk I/O time of each task in each compute node can be hidden by the more computations of other compute nodes as the system scale increases. In this case, the disk I/O latency is reduced before the compute node executes the computation stage of its next task. For example, as shown in Fig. 6, the disk I/O (S_0 and L_5) time of compute node 0 can be overlapped completely with the computation times of Task 1, Task 2, Task 3 and Task 4. If there are only four compute nodes, the disk I/O (S_0 and L_5) time of compute node 0 cannot be hidden completely by the computations of Task 1, Task 2 and Task 3. The *lags* can be eliminated completely when the number of compute nodes is sufficiently large since the saving-results stage of task t and the loading-partition stage of task $t+M$ can be fully overlapped with a sufficient number $M-1$ of computation stages of tasks following task t, each executed by a dedicated compute node, before the computation stage of task $t+M$ starts in this case. Note that the tasks t and $t+M$ are assigned to the same compute node, M is the number of compute nodes. In this case, DD-Graph reaches its peak performance. When the system scale continues to increase, there are more and longer *gaps*. As mentioned before, a *gap* represents an idle period between the loading partition stage and the computation stage of a task, leading to efficiency loss. Conversely, fewer compute nodes result in fewer and shorter *gaps* but more and longer *lags* leading to performance loss but efficiency improvement.

DD-Graph can increase or decrease the *system scale* (i.e., the number of compute nodes) to tradeoff between performance and efficiency. In general, while the performance of DD-Graph increases with the *system scale*, its efficiency is conversely correlated to the *system scale*. The efficiency reaches a maximum value when *system scale* $=$ 1. In this case, DD-Graph is degraded into a GraphChi-like single-node disk-based graph-processing framework.

The *system scale* that achieves the peak performance depends on the specific graph algorithm since the computation time and load/store time vary from one graph algorithm to another. To easily demonstrate DD-Graph, in Fig. 4, the disk I/O time of each task is only ∼2x longer than its computation time, i.e., the time spent on loading the partition or saving the results is shown to be only as long as that on computing the partition. Hence, in this case, DD-Graph with three compute nodes can nearly reach it peak performance. In fact, for most graph algorithms, the difference between the disk I/O time and the computation time is usually more than the 2x factor implied in Fig. 4. In such cases, DD-Graph requires more compute nodes to reach its peak performance, instead of only three compute nodes as the case of Fig. 4. Fig. 6 demonstrates an example of such a case where the time spent on the saving-results stage of Task 0 and the loading-partition stage of Task 5 is overlapped with time spent on the computations of Task 1, Task2, Task 3, and Task 4. DD-Graph uses 5 compute nodes to nearly reach peak performance in this example. Even so, system scales achieving the peak performance of DD-Graph are usually much smaller than those of existing distributed in-memory graph-processing frameworks while achieving the high performance of the latter. For example, as

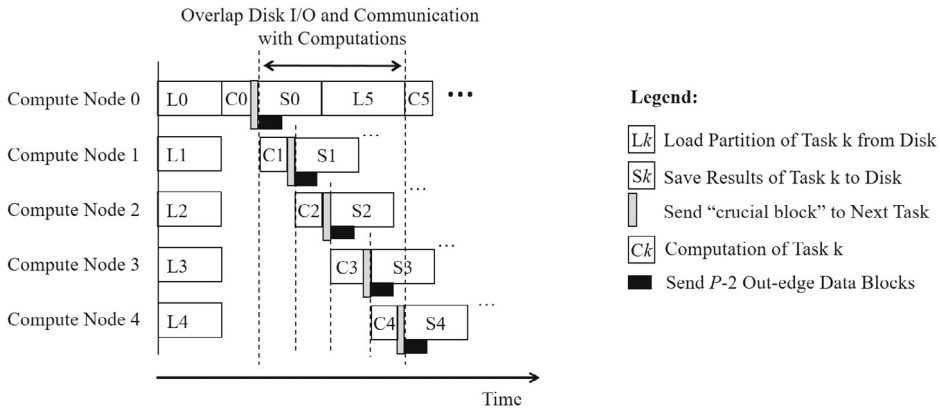


Fig. 6. DD-Graph reaches peak performance in case of each task with longer disk I/O times.

Table 1
Summary of graph datasets.

DataSets	Vertices	Edges	Type
Twitter-2010[a]	41×10^6	1.4×10^9	Social network
UK-2007-05[a]	106×10^6	3.7×10^9	Web
Hyperlink-2012[b]	3.5×10^9	128×10^9	Web
RMAT27[c]	128×10^6	2×10^9	Graph500
RMAT28[c]	256×10^6	4×10^9	Graph500
RMAT29[c]	512×10^6	8×10^9	Graph500
RMAT30[c]	1×10^9	16×10^9	Graph500
RMAT31[c]	2×10^9	32×10^9	Graph500
RMAT32[c]	4×10^9	64×10^9	Graph500
RMAT36[c]	64×10^9	1024×10^9	Graph500

[a] http://law.di.unimi.it/datasets.php.
[b] http://webdatacommons.org/hyperlinkgraph/.
[c] http://www.graph500.org.

shown in Section 4, the DD-Graph with 12 compute nodes reaches its peak performance when running PageRank on the Twitter-2010 graph, in contrast to GPS that requires 30 nodes.

Given a limited number of compute nodes, the numbers and lengths of *laps*/*gaps* also depends on another factor, namely, graph partitioning, that is discussed in the next subsection.

4.5. Impact of graph partitioning

Given a limited number of compute nodes, DD-Graph can obtain a higher system performance by reducing the number of *laps*/*gaps* and shortening the *laps*/*gaps*. In other words, for optimal performance, tasks should have approximately the same disk I/O time spent on loading their partitions and saving the results, and roughly the same amount of computation time. To achieve this, two critical conditions must be met: (1) the cluster is built of homogeneous/identical compute nodes; (2) the workloads of partitions are balanced. The first condition can easily be met since the number of compute nodes of DD-Graph is usually small. For the second condition, the workloads of partitions can be well balanced by using a proper graph partitioning method, such as Round-Robin and Steaming Graph Partitioning [29].

5. Experimental evaluation

In this section, we conduct extensive experiments to evaluate the performance of DD-Graph. Experiments are conducted on a 50-node cluster. Each node has two quad-core Intel Xeon E5620 processors with 32 GB of RAM and eight 15,000RPM SAS disks (1.2 TB in total). DD-Graph uses *round-robin* as the default graph partitioning method.

Applications and Datasets: We implement five graph algorithms to evaluate DD-Graph: PageRank (PR) [30], Community Detection (CD) [11], Connected Components (CC) [11], RandomWalks (RW) [31] and Breadth First Search (BFS) [32]. We evaluate DD-Graph by using one social network graph, two web graphs and seven graph500 graphs. These datasets are summarized in Table 1.

Baseline Frameworks: We compare DD-Graph with GPS, Giraph, Chaos, Pregelix, GraphD and GraphH. GPS is an open-source Pregel implementation from Stanford InfoLab [5]. It is a representative Bulk Synchronous Parallel (BSP) computation model [28] based distributed graph-processing framework. We also choose Apache Giraph.[2] as it is a well-known open source implementation of Pregel. Chaos, Pregelix, GraphD and GraphH are distributed secondary storage based graph-processing frameworks that have been proposed in recent years [10,15–17]

5.1. Performance analysis

In order to have a clear understanding about how DD-Graph achieves a comparable performance to existing distributed in-memory graph-processing frameworks but with much lower hardware costs, experiments are conducted to investigate the runtime breakdowns of DD-Graph and GPS. Each framework runs 10 iterations of PR on the Twitter-2010 graph repeatedly, with the number of compute nodes ranging from 1 to 32. We decompose the runtime of DD-Graph into two parts: (1) computation latency and (2) disk I/O & communication latency. The computation latency is measured by adding up the computation time of each task. The disk I/O & communication latency is measured by adding up the time gap between any two adjacent computations. The runtime of GPS consists of two parts: (1) computation latency and (2) communication latency. The computation latency is measured by adding up the computation time of each iteration. In each iteration, the computation time is used by the slowest worker to loop through its assigned vertices. The overall communication latency is measured by adding up the communication latency of each iteration. In each iteration, since the computation of GPS is overlapped with the communication, the communication latency is calculated by subtracting the computation time from the communication time that is the time gap between the first sent message and the last received message.

Experimental results, as shown in Fig. 7, indicate that the computation latency of DD-Graph maintains a constant value when the system scale ranges from 1 to 32. The reason is that, recall from Section 3.2 and Fig. 5, the computations of the physically distributed tasks are actually executed sequentially in time by DD-Graph. Thus,

[2] Apache Giraph: http://giraph.apache.org.

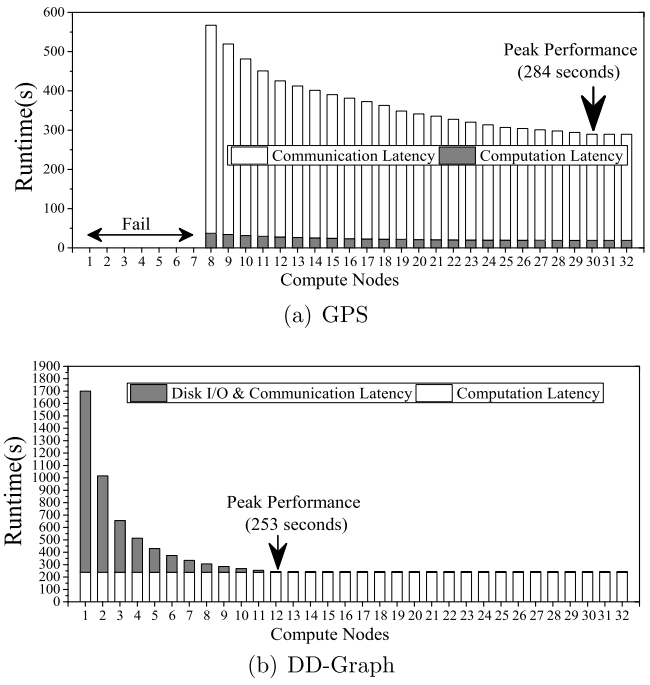


Fig. 7. Runtime breakdown. Each framework runs PR on the Twitter-2010 graph repeatedly, with the system scale ranging from 1 to 32.

for the same algorithm and graph dataset, the computation latency of DD-Graph is roughly constant and independent of the system scale. Due to parallel execution of computations, the computation latency and communication latency of GPS are reduced gradually as the system scale increases from 8 to 30. However, the runtime of GPS maintains a constant value when the system scale ranges from 30 to 32. The reason is the limited scalability of GPS. Note that GPS fails to execute the graph-computing job when system scale is less than 8. However, the disk I/O & communication latency of DD-Graph is reduced significantly when the system scale ranges from 1 to 12 and reaches the peak performance at the system scale of 12. The reasons are twofold. First, more disk I/O and communication time can be overlapped with the computation time by using more compute nodes. Second, most of the communication and disk I/O time has been overlapped when 12 compute nodes are used. In fact, as indicated by Fig. 7, DD-Graph and GPS reach their respective peak performance at 12 and 30 compute nodes respectively. Although the computation latency of DD-Graph is longer than that of GPS, the disk I/O & communication latency of DD-Graph is much shorter than the communication latency of GPS, leading to slightly shorter overall runtime of DD-Graph (i.e., 253 s vs. 284 s).

Fig. 7(b) indicates that the runtime of DD-Graph remains constant when the system scales from 12 nodes to 32 nodes. The reason are twofold. First, although the computation times and disk I/O times of each compute node continue to be reduced with the increasing system scale, the computations of the physically distributed tasks are actually executed sequentially in time by DD-Graph, as mentioned before. Thus, the computation time (white bar) of DD-Graph remains constant when the system scale ranges from 1 to 32. Second, when the system scales from 12 nodes to 32 nodes, almost the entire communication latency and disk I/O latency have been hidden by overlapping the disk I/O and communication times of each compute node with the computation times of other compute nodes.

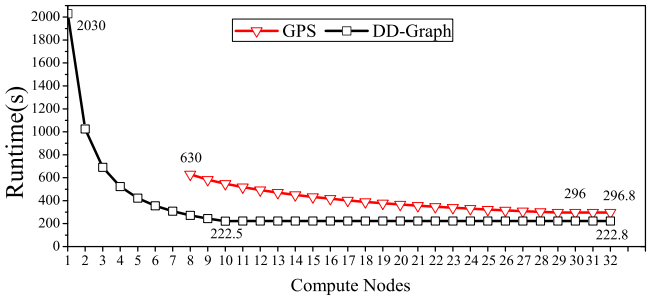


Fig. 8. Optimal performance of the CC graph algorithm.

These experimental results show that although there is a significant difference in the numbers of compute nodes between DD-Graph and GPS, DD-Graph obtains slight performance improvements over GPS. The reasons are twofold. First, although the computations of DD-Graph are executed sequentially, the costly disk I/O and the communication latencies have been hidden. The computation time of each task is much shorter than its disk I/O time. Second, although GPS is a distributed in-memory graph-processing framework, the limited performance of GPS is caused mainly by the costly communication and limited scalability [7,10].

5.2. Hardware cost, performance and efficiency

DD-Graph is compared with GPS in terms of the hardware cost, performance and efficiency. We conduct two sets of experiments to evaluate DD-Graph against GPS. For each set of experiments, each graph-processing framework runs CC on the Twitter-2010 graph repeatedly, with the system scale ranging from 1 to 32. As illustrated in Fig. 8, DD-Graph achieves its peak performance (222.5 s) when 10 compute nodes are used. Due to the out-of-memory problem, GPS simply crashes when the system scale ranges from 1 to 7. GPS achieves its peak performance (296 s) when 30 compute nodes are used. We use the peak performances of DD-Graph and GPS for an easy two-way comparison. Although there is a big difference in system scale, i.e., $30 - 10 = 20$ compute nodes, between DD-Graph and GPS, the runtime of DD-Graph is 1.34x shorter than that of GPS.

Experiments are also conducted on BFS, PR, CD and RW. We show the experimental results with the peak performances of the graph-computing jobs to analyze DD-Graph against GPS in terms of the hardware cost, performance and efficiency. Fig. 9(a) shows that the system scale of DD-Graph is only 27%∼40% of that of GPS. Further, the peak-performance system scale for DD-Graph varies because the computation time and load/store time vary from one graph algorithm to another. Experimental results, illustrated in Fig. 9(b), indicate that although the peak-performance system scale of DD-Graph is consistently and significantly smaller than that of GPS, DD-Graph has a slight performance advantage. Combining the results of Fig. 9(a) and 9(b), Fig. 9(c) shows the efficiency of DD-Graph and GPS, i.e, CC = (Number of Compute nodes) × Runtime. A smaller CC value, as defined in Section 3, which indicates that the efficiency of DD-Graph is 2.9x–4x higher than that of GPS. In summary, results of these experiments show that DD-Graph saves 60%–73% of GPS' hardware costs while running 1x–1.34x faster than GPS.

5.3. Very large graphs

We illustrate the key feature of the DD-Graph, that is, it can handle very large graphs on a small cluster while achieving a comparable performance with existing distributed in-memory graph-processing frameworks.

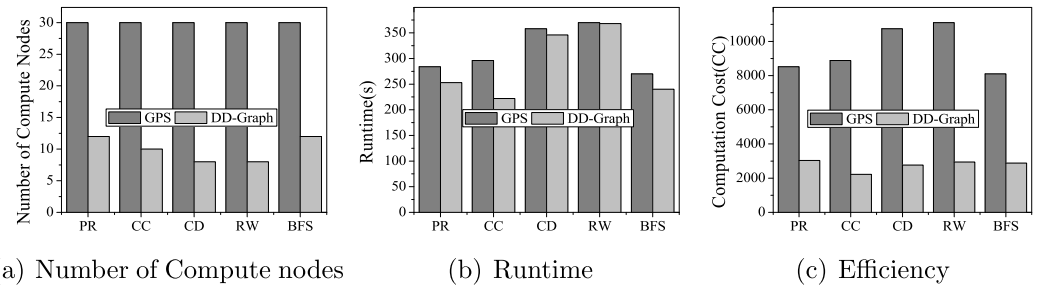


Fig. 9. Hardware cost, performance and efficiency. The computation cost (CC) is defined as the number of compute nodes times the runtime.

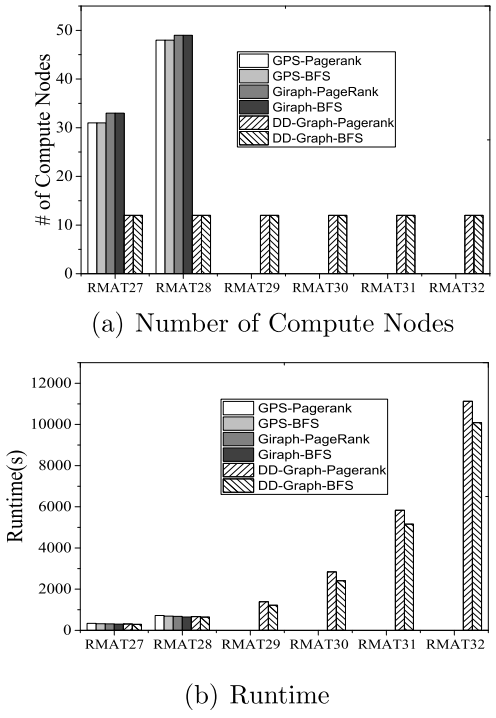


Fig. 10. Large graphs.

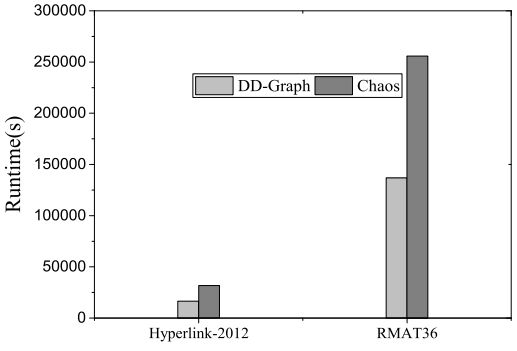


Fig. 11. Very large graphs.

We first compare DD-Graph with GPS and Giraph using a set of graphs, i.e., RMAT27, RMAT28, RMAT29, RMAT30, RMAT31 and RMAT32. As shown in Table 1, these graphs have a large size. Since the experimental results, shown in Section 4.2, indicate that DD-Graph can achieve its peak performance on 12 compute nodes for the PR and BFS graph algorithms. DD-Graph runs PR and BFS with these graphs respectively on the 12-node cluster.

GPS runs PR and BFS on the RMAT27 graph repeatedly, by increasing the system scale. GPS simply crashes when the system scale falls below 11, but as the system scale increases its runtime is reduced gradually and reaches the minimum values (340.6 s and 321 s) when 31 compute nodes are used. Giraph also runs PR and BFS on the RMAT27 graph repeatedly, by increasing the system scale. Its runtime is reduced gradually and reaches the minimum values (317 s and 304 s) when 33 compute nodes are used. In these experiments, Giraph obtains 4.7%∼5.5% performance improvement over GPS while there is a small difference in system scale, i.e., $33 - 31 = 2$ compute nodes, between Giraph and GPS. However, DD-Graph with only 12 compute nodes executes the same graph-computing jobs in 312.5 and 289 s. Experiments are repeated on RMAT28. Similarly, GPS reaches its peak performances (723.9 s and 690 s) when 48 compute nodes are used. Giraph reaches its peak performances (680 s and 649 s) when 49 compute nodes are used. DD-Graph with only 12 compute nodes executes the same graph-computing jobs in 673.5 and 634 s. As shown in Fig. 10(a), DD-Graph saves more compute nodes when handling RMAT28 than RMAT27. Furthermore, it can obtain slight performance improvements when handling both RMAT28 and RMAT27, as shown in Fig. 10(b). These experimental results indicate that DD-Graph is more cost-effective when handling a larger graph than a smaller one.

As shown in Fig. 10, GPS and Giraph simply crash when running on the 50-node cluster with RMAT29, RMAT30, RMAT31 and RMAT32, due to the out-of-memory problem. However, DD-Graph can process the RMAT29, RMAT30, RMAT31 and RMAT32 respectively on a cluster of 12 compute nodes.

We then compare DD-Graph with Chaos using two larger graphs, i.e., Hyperlink-2012 and RMAT36. Chaos is a distributed secondary storage based graph-processing framework that has been proposed recently. Each framework first runs 10 supersteps of Pagerank with 12 compute nodes on the Hyperlink-2012. This graph covers 3.5 billion web pages and 128 billion hyperlinks between these pages. To the best of our knowledge, it is the largest hyperlink graph that is available to the public outside companies. As shown in Fig. 11, DD-Graph is 1.87x faster than Chaos (i.e., 16560 s vs. 31795.2 s) when running on the Hyperlink-2012. Each framework then runs 10 supersteps of Pagerank on the RMAT36 graph, using 24 compute nodes to provide sufficient aggregate disk capacity. RMAT36 is 16x as large as the RMAT32 graph and requires 16 TB of input data. Experimental results show that DD-Graph is 1.96x faster than Chaos (i.e., 136 800 s vs. 255816 s) when running on the RMAT36. The higher performance of DD-Graph stems mainly from the two reasons. First, the communication load of Chaos is heavy since the technique of work stealing leads to data migrations among compute nodes [10]. However, the communication of DD-Graph is efficient. Especially, due to the edge-data-block based data representation, it eliminates the high

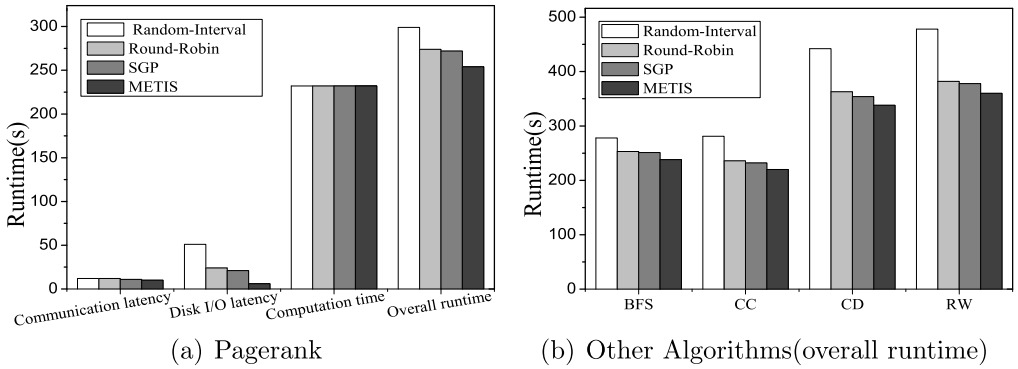


Fig. 12. Impact with graph partitioning.

extra volume of communication in existing BSP-based distributed graph-processing frameworks. As mentioned in Section 3.1, the extra communication volume is usually responsible for up to 67% of the overall communication volume. Second, during the whole execution process, DD-Graph overlaps the communication and disk I/O of each compute node with the computations of other compute nodes. Once the graph-computing job starts, the three key components of computation (CPUs), communication and disk I/O in each compute node of the cluster are busy all the time, enabling these components to be utilized efficiently. In order to improve the computation resources utilization, Chaos overlaps the computation and the disk I/O in each compute node. However, the gain from this method is limited since the computation time is much shorter than the disk I/O time in each compute node.

5.4. Performance impact of graph partitioning

In order to study DD-Graph in terms of the performance impact of graph partitioning, we conduct four sets of experiments according to the four graph partitioning methods of *Random-interval*, *Round-robin*, SGP and METIS in terms of the communication latency, disk I/O latency and computation time. For each set of experiments, DD-Graph first divides the Twitter-2010 graph into partitions and then runs the pagerank graph algorithm on a cluster with system scale of 11.

Experimental results, as shown in Fig. 12(a), indicate that the communication latency is responsible for 4.7%, 4.6%, 4.1% and 3.8% of the overall runtime respectively with Random-Interval, Round-robin, SGP and METIS. The reason is that, as mentioned in Section 3.2, DD-Graph hides almost the entire communication time, the short communication latency of each experiment comes from the "crucial blocks". Since the computation stages are executed sequentially, the computation times of DD-Graph are similar when running on different graph partitioning methods. Compared with computation times, the disk I/O latencies are also short. The reason is that almost the entire disk I/O time of each task is hidden by the computation times of other compute nodes. Considering the Random-Interval case as the baseline, the communication latency is reduced at 47%, 26.4% and 11.7% respectively with the Round-robin, SGP and METIS. The reason is that Round-robin, SGP and METIS well balance the workload among the partitions, significantly reducing the number of *laps*/*gaps* and shortening the *laps*/*gaps*. In these experiments, DD-Graph with round-robin can obtain 21% performance improvements over Random-Interval. DD-Graph with SGP can only obtain less than 3% performance improvements over round-robin. The reason is that the system performance of DD-Graph slightly depends on the edge-cut ratios reduced by SGP. Compared with SGP, DD-Graph with METIS can obtain less than 5.7% performance improvement over SGP. We also conduct similar experiments on the graph algorithms of BFS, CC, CD and RW. The experimental results are similar to that of pagerank. Fig. 12(b) shows the experimental results of overall runtime cases.

5.5. Performance impact of networks

Experiments are conducted to compare DD-Graph with Chaos, Pregelix, GraphD and GraphH in terms of the impact of the network ecosystem. In these experiments, each system is deployed on 12 compute nodes, and runs 10 supersteps of PR on the RMAT30 graph.

We first compare DD-Graph with Chaos, Pregelix, GraphD and GraphH over the 1 Gbps Ethernet. In these experiments, the network bandwidth is the performance bottleneck in Chaos and Pregelix. For example, as shown in Fig. 13(a), Chaos is 3.9x, 4.8x and 6.79x slower than GraphD, GraphH and DD-Graph respectively. We then evaluate DD-Graph against Chaos, Pregelix, GraphD and GraphH over the 40 Gbps Infiniband. As shown in Fig. 13(b), Chaos with the 40 Gbps network is 4.89x faster than 1 Gbps Ethernet case. Similarly, Pregelix also gains significant performance improvement when a 40 Gbps network is available. These experimental results indicate that the performance of Chaos and Pregelix relies heavily on the network bandwidth [10]. Unlike Chaos and Pregelix, GraphD with the 40 Gbps network only obtains 17% performance improvement over the 1 Gbps Ethernet case. When running on the 40 Gbps network, Chaos is 1.1x faster than GraphD. The reason is that GraphD assumes a small cluster of commodity PCs connected by a low-bandwidth network. In this case, the communication time is longer than the disk I/O time. Thus, the disk I/O can be hidden by the communication. However, the disk I/O will become the bottleneck when system running on the 40 Gbps network. The reason of the costly disk I/O is that GraphD loads vertices into memory only, resulting in extra disk I/O for communication [16]. GraphH employs an edge cache mechanism to reduce the disk I/O overhead, gaining 27% performance improvement over GraphD, as shown in Fig. 13(b). When running on the 40 Gbps network, DD-Graph is 1.63x, 1.52x, 1.76x and 1.38x faster than Chaos, Pregelix, GraphD and GraphH respectively. The performance improvements stem from the three key features of the pipeline-based task scheduling strategy and the two optimizations of DD-Graph, as detailed in Sections 3 and 4.

5.6. Convergence speed

We conduct experiments to compare DD-Graph with GraphD in terms of convergence speed. As mentioned in Section 3.3, by pipelining the tasks of the graph-computing job, our scheduling strategy can reduce the number of supersteps of the graph-computing job significantly. This is important since the total resources of a small cluster are limited, compared with a large one.

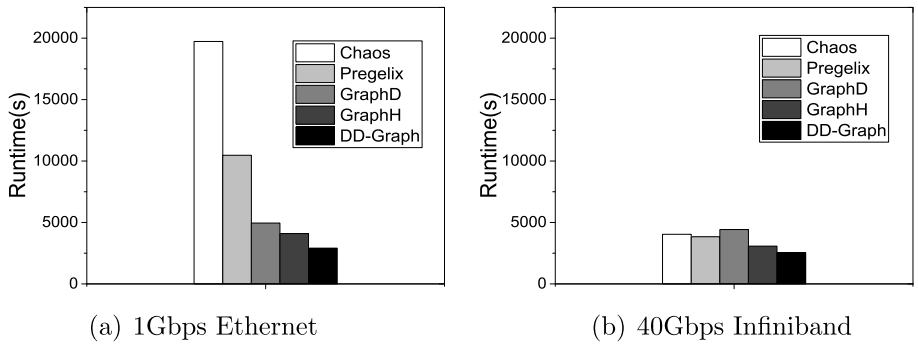


Fig. 13. Impact with network ecosystem.

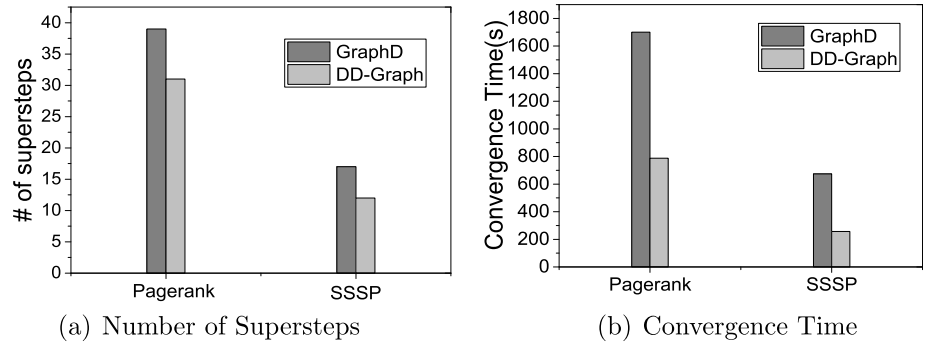


Fig. 14. Convergence speed.

However, the reduced number of supersteps can speed up the execution process of graph-computing jobs, without needing to increase the system scale.

Each graph-processing system runs SSSP and PR over the Twitter-2010 graph on 12 compute nodes respectively. Experimental results, as shown in Fig. 14, indicate that DD-Graph can accelerate the convergence of graph algorithms by gaining notable reductions in the number of supersteps, which contribute to the shorter run times. For example, the Pagerank graph-algorithm job achieves convergence by the 31th superstep in DD-Graph. However, convergence is not achieved until the 39th superstep in GraphD. In this case, DD-Graph is 2.13x faster than GraphD. The high performance stems from the faster convergence and the high utilization of the resources of the whole cluster that stems from the pipeline-based task scheduling strategy.

6. Related work

In this section, we briefly discuss the work on graph-processing frameworks most relevant to our DD-Graph.

Distributed in-memory graph-processing frameworks [4,5,7,33,33,34] can process large graphs with high performance by using a large cluster of commodity machines. For example, Pregel is deployed on clusters of thousands of machines to process very large graphs [4]. Checconi F. et al. [35] explore graphs on large-scale parallel machines in real time. Andrea Clementi et al. [36] use large-scale clusters to detect communities in dynamic graphs. In general, scaling out a cluster of commodity machines can improve scalability and performance. However, these systems are expensive for most common users due to the excessive investment of compute nodes.

Roger Pearce et al. use the IBM BG/P Intrepid supercomputer with up to 131 K processors to traverse large scale-free graphs. Their system performance outperforms the best known Graph500 performance on BG/P Intrepid by 15% [9]. GRAM [37] uses 64 multi-core servers to process large graphs. Each server runs Windows Server 2012 R2 on dual 2.6 GHz Intel Xeon E5-2650 processors (16 physical cores and 32 logical cores), 256 GB of memory. The success of these systems lies in that they leverage the powerful processors and the sufficient aggregate memory capacity efficiently.

Single-node disk-based graph-processing frameworks can handle large scale graphs on just one commodity computer. However they usually suffer from poor performance due to the costly disk I/Os. GridGraph [20] is an excellent single-node disk-based graph-processing framework that breaks graphs into 1D-partitioned vertex chunks and 2D-partitioned edge blocks using a first fine-grained level partitioning in preprocessing. A second coarse-grained level partitioning is applied in runtime. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reducing the I/O amount required for computation.

GraphChi [11] introduces a novel mechanism called Parallel Sliding Windows (PSW) to alleviate the issue of random accesses to improve the disk I/O performance. X-Stream [12] avoids the random accesses by streaming completely unordered edge lists, aiming to improve the disk I/O performance. Another potentially more efficient solution would be one that uses flash or PCI attached non-volatile memory (NVRAM) to reduce the I/O latency. However, as evidenced by the SSD-based GraphChi, the performance improvement is limited by the following factors, First, single-node's limited aggregate I/O capacity limits the performance improvement over disk. Second, NVRAM has significantly slower access speeds than main-memory (DRAM) [14]. Besides the limited single-node

I/O bandwidth, commodity computer usually has the limited disk space that limits the size of the processed graph, especially for processing graphs with trillion edges [10].

Several distributed disk-based graph-processing frameworks have been proposed recently to process large-scale graphs with a small cluster of commodity PCs that is affordable to most common users. However, there are two key challenges in the design of a distributed secondary storage based framework. First, the communication is heavier. Since each compute node needs to process a larger partition and generate heavier communication workload in each superstep, compared with distributed in-memory graph-processing frameworks. The second challenge is the costly disk I/O experienced by each compute node. These are also evidenced by most recent research results, such as Chaos [10] and Pregelix [15].

In order to reduce the costly communication and disk I/O, GraphD [16] is proposed recently to hide the disk I/O cost by overlapping the disk I/O with the communication in each compute node of the small cluster. This is effective when a high-speed network is not available. However, DD-Graph is effective for not only the high-bandwidth but also the low-bandwidth networks. GraphD implements Pregel's synchronous execution model, that is, each compute node is executed in parallel. Unlike GraphD, DD-Graph executes the graph-computing job by pipelining the tasks, reducing number of supersteps of the graph-computing job significantly.

GraphH [17] is a recent research that aims to improve the performance of distributed disk-based graph-processing frameworks by using a two-stage graph partition scheme, GAB (Gather-Apply-Broadcast) computation model and an edge cache mechanism. This work was done by the team from Nanyang Technological University and published in arxiv.org. The authors claim that GraphH could be up to 100x faster than Chaos and GraphD. We think this comparison is inappropriate and unfair for Chaos and GraphD, due to the well-designed experiments. For example, they claim that GraphH could outperform Chaos and GraphD by at least 350x when running one superstep of SSSP. As we know, in the first superstep of SSSP, only the neighbors of the source vertex are executed. This is an unfair case for Chaos since it needs to load all edges in each superstep.

InfiniMem [38] is proposed to transparently support disk-resident versions of object collections so that they can grow to large sizes without causing programs to crash. The solution of InfiniMem is to support size oblivious programming for C++ programs via the InfiniMem C++ library and runtime. InfiniMem can be used to generate large RMAT graphs and process large graphs in a single machine. It is also integrated with distributed shared memory (DSM) to process very large graphs in a small cluster. This is very useful for programmers, due its simple programming interfaces. However, InfiniMem is a general-purpose framework that has been designed to support disk-resident versions of object collections, by using the InfiniMem C++ library and runtime. This generality does, however, come at a performance cost. For a distributed disk-based graph-processing system, the performance relies on not only the convergence speed of the graph-processing job, but also the utilization of the resources of the whole cluster.

7. Conclusions

This paper proposes DD-Graph, a distributed disk-based graph-processing framework, to support very large scale graph computing jobs. DD-Graph is cost-effective and of high-performance. By scheduling the tasks of a graph-computing job on several compute nodes, DD-Graph is capable of processing very large graphs on a small cluster while achieving the high performance of existing distributed in-memory graph-processing frameworks. Extensive prototype evaluation of DD-Graph, driven by very large graph datasets, indicates that the cost-effective advantage of DD-Graph makes it notably superior to the existing distributed graph-processing frameworks, particularly when processing the very large graphs. For example, DD-Graph saves 73% of GPS' hardware costs while running 1.34x faster than GPS.

Acknowledgments

This work is supported by NSFC, China 61772216, 61672159 and U1705262. This work is also supported by Key Laboratory of Information Storage System, China, Ministry of Education and State Key Laboratory of Computer Architecture, China (No. CARCH201505), Technology Innovation Platform Project of Fujian Province, China under Grant (No. 2014H2005), Fujian Collaborative Innovation Center for Big Data Application in Governments, China, Fujian Engineering Research Center of Big Data Analysis and Processing, China.

References

[1] J. Sun, D. Zhou, H. Chen, C. Chang, Z. Chen, W. Li, L. He, Gpsa: A graph processing system with actors, in: 44th International Conference on Parallel Processing, ICPP, IEEE Computer Society Press, Beijing, China, 2015, pp. 709–718.
[2] D. Nguyen, A. Lenharth, K. Pingali, A lightweight infrastructure for graph analytics, in: ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 456–471.
[3] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (01) (2007) 5–20.
[4] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: ACM SIGMOD International Conference on Management of Data, ACM Association, Indianapolis, Indiana, 2010, pp. 135–146.
[5] S. Salihoglu, J. Widom, Gps: a graph processing system, in: International Conference on Scientific and Statistical Database Management, ACM Association, Baltimore, MD, USA, 2013, pp. 22–32.
[6] B. Shao, H. Wang, Y. Li, Trinity: a distributed graph engine on a memory cloud, in: ACM SIGMOD International Conference on Management of Data, ACM Association, New York, USA, 2013, pp. 505–516.
[7] C. Zhou, J. Gao, B. Sun, J.X. Yu, Mocgraph: scalable distributed graph processing using message online computing, Proc. VLDB Endowment 8 (4) (2014) 377–388.
[8] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: Usenix Conference on Operating Systems Design and Implementation, 2012, pp. 17–30.
[9] R. Pearce, M. Gokhale, N.M. Amato, Faster parallel traversal of scale free graphs at extreme scale with vertex delegates, in: International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE Computer Society Press, New Orleans, LA, USA, 2014, pp. 197–207.
[10] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, Chaos:scale-out graph processing from secondary storage, in: Symposium on Operating Systems Principles, ACM Association, Monterey, California, USA, 2015, pp. 410–424.
[11] A. Kyrola, G. Blelloch, C. Guestrin, Graphchi: large-scale graph computation on just a pc, in: Usenix Conference on Operating Systems Design and Implementation, Usenix Association, Hollywood, CA, USA, 2012, pp. 31–46.
[12] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: edge-centric graph processing using streaming partitions, in: Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM Association, Pennsylvania, USA, 2013, pp. 472–488.
[13] J. Malicevic, A. Roy, W. Zwaenepoelin, Scale-up graph processing in the cloud: Challenges and solutions, in: International Workshop on Cloud Data and Platforms, ACM, 2014, pp. 1–6.
[14] R. Pearce, M. Gokhale, N.M. Amato, Scaling techniques for massive scale-free graphs in distributed (external) memory, in: IEEE International Parallel and Distributed Processing Symposium, 2013, pp. 315–324.
[15] Y. Bu, V. Borkar, J. Jia, M.J. Carey, T. Condie, Pregelix: Big(ger) graph analytics on a dataflow engine, Proc. VLDB Endowment 8 (2) (2014) 129–137.
[16] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, C. Zhang, Graphd: Distributed vertex-centric graph processing beyond the memory limit, IEEE Trans. Parallel Distrib. Syst. 12 (99) (2018) 1–14.
[17] P. Sun, Y. Wen, T.N.B. Duong, X. Xiao, Graphh: High performance big graph analytics in small clusters, 2017, pp. 256–266. arXiv.
[18] Z. Xiaowei, C. Wenguang, Z. Weimin, M. Xiaosong, Gemini: A computation-centric distributed graph processing system, in: USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 301–316.

[19] Y. Cheng, H. Jiang, F. Wang, Y. Hua, D. Feng, Blitzg: Exploiting high-bandwidth networks for fast graph processing, in: IEEE International Conference on Computer Communications, IEEE Computer Society Press, Atlanta, GA, USA, 2017, pp. 2340–2348.
[20] X. Zhu, W. Han, W. Chen, Gridgraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning, in: Usenix Conference on Usenix Technical Conference, 2015, pp. 375–386.
[21] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C.E. Priebe, A.S. Szalay, Flashgraph: processing billion-node graphs on an array of commodity ssds, in: Usenix Conference on File and Storage Technologies, 2015, pp. 45–58.
[22] K. Vora, G. Xu, R. Gupta, Load the edges you need: a generic i/o optimization for disk-based graph processing, in: Usenix Conference on Usenix Technical Conference, 2016, pp. 507–522.
[23] J. Cheng, Q. Liu, Z. Li, W. Fan, J.C.S. Lui, C. He, Venus: Vertex-centric streamlined graph computation on a single pc, in: IEEE International Conference on Data Engineering, 2015, pp. 1131–1142.
[24] M. Han, K. Daudjee, Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems, VLDB J. 8 (9) (2015) 950–961.
[25] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, V. Prasanna, Goffish: A sub-graph centric framework for large-scale graph analytics, in: Euro-Par 2014 Parallel Processing, Springer, 2014, pp. 451–462.
[26] J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: ACM SIGPLAN Annual Symposium Principles and Practice of Parallel Programming, 2013, pp. 135–146.
[27] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, X. Wang, Lcc-graph: A high-performance graph-processing framework with low communication costs, in: IEEE/ACM International Symposium on Quality of Service, 2016, pp. 91–100.
[28] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[29] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: KDD, 2012, pp. 1222–1230.
[30] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: bringing order to the web, Stanford Digit. Libr. Working Pap. 9 (1) (1999) 1–14.
[31] L. Lovasz, Random walks on graphs: A survey, Combinatorics 2 (1) (1993) 1–46. Paul erdos is eighty.
[32] H. Liu, H.H. Huang, Enterprise: breadth-first graph traversal on gpus, in: The International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Austin, TX, USA, 2015, pp. 1–12.
[33] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, P. Kalnis, Mizan: a system for dynamic load balancing in large-scale graph processing, in: ACM European Conference on Computer Systems, ACM Association, Prague, Czech Republic, 2013, pp. 169–182.
[34] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, Proc. VLDB Endowment 5 (8) (2012) 716–727.
[35] F. Checconi, F. Petrini, Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines, in: IEEE International Parallel & Distributed Processing Symposium, 2014, pp. 425–434.
[36] A. Clementi, M.D. Ianni, Gambosi, et al., Distributed community detection in dynamic graphs, in: Structural Information and Communication Complexity, Springer, 2013, pp. 1–12.
[37] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, L. Zhou, Gram: scaling graph computation to the trillions, in: ACM Symposium on Cloud Computing, 2015, pp. 408–421.
[38] S.C. Koduru, R. Gupta, I. Neamtiu, Size oblivious programming with infinimem, in: International Workshop on Languages and Compilers for Parallel Computing, 2015, pp. 3–19.

Yongli Cheng He received the B.E. degree from the Chang'an University, Xi'an, China, in 1998, the M.S. degree from the FuZhou University, FuZhou, China, in 2010, and Ph.D. degree from Huazhong University of Science and Technology, Wuhan, China, 2017. He is a teacher of College of Mathematics and Computer Science at FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in international conferences and major journals, including HPDC, IWQoS, INFOCOM and FCS.

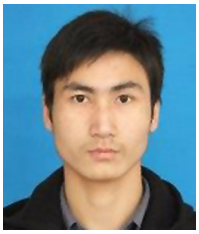
Fang Wang She received her B.E. degree and Master degree in computer science in 1994, 1997, and Ph.D. degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 50 publications in major journals and international conferences, including FGCS, ACM TACO, SCIENCE CHINA Information Sciences, Chinese Journal of Computers and HiPC, ICDCS, HPDC, ICPP.

Hong Jiang He received the B.E. degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MA.Sc. degree from the University of Toronto, Canada, in 1987, and the Ph.D. degree from the Texas A&M University, College Station, in 1991. He is Wendell H. Nedderman Endowed Professor & Chair of Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems and parallel/distributed computing. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACM-TOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD and the State of Nebraska. Dr. Jiang is a Fellow of IEEE.

Yu Hua He received the B.E. and Ph.D. degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing and network storage. He has more than 80 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS and IWQoS. He is a senior member of the IEEE and CCF, a member of ACM, and USENIX.

Dan Feng She received the B.E., M.E., and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE- TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.

Yunxiang Wu He received the B.E. degree in computer science and technology from the Wuhan University of Science and Technology (WUST), China, in 2009. He is currently a Ph.D. student majoring in computer architecture in Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and storage systems.



Tingwei Zhu He received the B.E. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012. He is currently working toward the Ph.D. degree in school of computer science and technology in HUST. His interests include software-defined networking and distributed storage systems. He has several publications in international conferences, including IWQoS and ICPP.

Wenzhong Guo He received the B.S. and M.S. degrees in computer science, and the Ph.D. degree in communication and information system from Fuzhou University, Fuzhou, China, in 2000, 2003, and 2010, respectively. He is currently a full professor with the College of Mathematics and Computer Science at Fuzhou University. His research interests include intelligent information processing, sensor networks, network computing, and network performance evaluation. He is a member of the IEEE.