Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies

Guo, Yong; Hong, Sungpack ; Chafi, Hassan; Iosup, Alexandru; Epema, Dick

DOI
10.1016/j.jpdc.2016.02.003

Publication date
2017

Document Version
Final published version

Published in
Journal of Parallel and Distributed Computing

J. Parallel Distrib. Comput. 108 (2017) 106–121

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Modeling, analysis, and experimental comparison of streaming graph-partitioning policies

Yong Guo [a,*,1], Sungpack Hong [b], Hassan Chafi [b], Alexandru Iosup [a], Dick Epema [a]

[a] *Delft University of Technology, The Netherlands*

[b] *Oracle Labs, USA*

HIGHLIGHTS

- We model the run time of different types of graph-processing systems.
- We design new graph-partitioning policies that address important challenges.
- We report comprehensive results about the performance of partitioning policies.
- We discuss the coverage of our model and method, and the design of future policies.

ARTICLE INFO

Article history:
Received 16 July 2015
Received in revised form
29 January 2016
Accepted 20 February 2016
Available online 3 March 2016

Keywords:
Streaming graph-partitioning policies
Large-scale graphs
Graph-processing systems
Modeling analysis
Performance evaluation

ABSTRACT

In recent years, many distributed graph-processing systems have been designed and developed to analyze large-scale graphs. For all distributed graph-processing systems, partitioning graphs is a key part of processing and an important aspect to achieve good processing performance. To keep low the overhead of partitioning graphs, even when processing the ever-increasing modern graphs, many previous studies use lightweight streaming graph-partitioning policies. Although many such policies exist, currently there is no comprehensive study of their impact on load balancing and communication overheads, and on the overall performance of graph-processing systems. This relative lack of understanding hampers the development and tuning of new streaming policies, and could limit the entire research community to the existing classes of policies. We address these issues in this work. We begin by modeling the execution time of distributed graph-processing systems. By analyzing this model under the load of realistic graph-data characteristics, we propose a method to identify important performance issues and then design new streaming graph-partitioning policies to address them. By using three typical large-scale graphs and three popular graph-processing algorithms, we conduct comprehensive experiments to study the performance of our and of many alternative streaming policies on a real distributed graph-processing system. We also explore the impact on performance of using different real-world networks and of other real-world technical details. We further discuss how to use our results, the coverage of our model and method, and the design of future partitioning policies.

1. Introduction

The scale of graphs is increasing rapidly in recent years, and has already exceeded the processing capabilities of single machines. Distributed graph-processing systems such as Pregel [27], GraphLab [25], and GraphX [12], have been designed and developed to process large-scale graphs by using the computation and memory capabilities of clusters. For such systems, graph partitioning is essential in achieving good performance, because it determines the computation workload of each working machine and the communication between them. Many streaming graph partitioning policies [39,45,42] have been proposed to efficiently partition graphs into balanced pieces for distributed graph-processing systems. *Streaming* graph partitioning treats graph data as an online stream, by reading the data serially and then determining the target partition of a vertex when it is accessed. However, the impact on the overall system performance of these partitioning policies

[*] Corresponding author.
E-mail addresses: Yong.Guo@tudelft.nl (Y. Guo), Sungpack.Hong@oracle.com (S. Hong), Hassan.Chafi@oracle.com (H. Chafi), A.Iosup@tudelft.nl (A. Iosup), D.H.J.Epema@tudelft.nl (D. Epema).
[1] This work was done when the author was doing an internship at Oracle Labs.

http://dx.doi.org/10.1016/j.jpdc.2016.02.003
0743-7315/© 2016 Elsevier Inc. All rights reserved.

has not been thoroughly evaluated on real graph-processing systems, and the understanding of the performance issues raised by such policies when used in real-world graph-processing systems is currently relatively limited. Gaining such knowledge can lead to the design of new policies, to new methods for tuning existing policies, and in general to better system design for distributed graph processing. Addressing this lack of understanding is the goal of our present work, in which we model, analyze and design new policies, and experimentally compare streaming graph-processing policies in real-world environments.

In this paper we address the following five important challenges in partitioning large-scale graphs. The first challenge is partitioning graphs into splits with balanced numbers of vertices while minimizing edge-cuts, which is an NP-complete problem [1]. For graphs with billions of edges [5], the partitioning time can become too long, even when using partitioning heuristics. Second, many graphs of interest are not static but dynamic, with vertices and edges being added all the time. As a consequence, graph partitioning is then an online streaming process rather than an offline process. Third, the performance of partitioning depends on the graph-processing application. Fourth, because they are designed to address the needs of specific communities, each with their own applications and domains of expertise, graph-processing systems are designed around different programming models and generally take different evolutionary paths. The core programming model, which specifies how the system performs computation on vertices and how the distributed components of the system communicate, can affect the performance impact of partitioning. Fifth, the structure and capacity of the cluster used may impact the performance effect of a partitioning policy on the run time of graph-processing systems. For instance, switching the network from relatively low-speed Ethernet to high-speed InfiniBand, or the level of heterogeneity of a cluster [45] may change the relative merits of partitioning policies.

Many graph-partitioning approaches have been proposed to address these challenges, from offline partitioning heuristics to online, streaming, graph-partitioning policies. These partitioning-centric studies focus on the design of reasonable partitioning policies that are based on heuristics and rely on a limited set of theoretical metrics, such as the edge cut ratio [39], the number of vertices per partition [35,22], etc. The partitions are created online by real-world graph-processing systems, which indicate that empirical metrics, such as partitioning time and algorithm run time are important for system developers and users. However, few partitioning policies have been proposed from the perspective of real systems. In contrast to such policies, the policies designed from a more theoretical perspective lack of simplicity and of considering the relationship between the computation and the communication, because they use relatively complicated heuristics and focus on minimizing the communication. And also, few experiments have been conducted on real graph-processing systems to evaluate the performance of existing partitioning policies. As our own and related studies [15,26,16] of entire graph-processing systems have shown, the results reported from narrow experiments can misreport performance by orders of magnitude, especially when the input workloads and the algorithms change from the conditions tested in the limited studies.

In this work, we address the challenges of streaming graph partitioning and the problem of relative lack of understanding about streaming graph-partitioning policies. In Section 3, we model the run time of distributed graph-processing systems. We set the objective function of partitioning to minimizing the run time. Our model extends related work [45] by including different programming models and implementation of graph-processing systems.

In Section 4, we conduct an experimental analysis of the performance implications of partitioning policies, using our run time model and conducting real-world measurements on a real-world graph-processing system—PGX.D [18]. PGX.D is an Oracle Labs tool and it can be up to ninety times faster than other popular graph-processing systems [18], such as GraphLab [25] and GraphX [12]. We find out what graph characteristics are closely related to the run time. We further propose streaming graph policies based on the run-time-influencing graph characteristics.

In Section 5, we evaluate and compare the performance of our policies, other streaming alternative, and also the start-of-the-art offline partitioner—METIS [19] on PGX.D, by using 3 large-scale graphs and 3 popular graph-processing algorithms. We use a set of metrics to present the partitioning performance, such as run time, partitioning time, edge cut ratio, scalability, etc. We also consider the impact of different real-world networks (Ethernet and InfiniBand) and the impact of a common technique (selective ghost node) used by graph-processing systems.

In Section 6, we further discuss how to use our results, the coverage of our method for different types of real-world graph-processing system, and the design of future partitioning policies based on our comprehensive experimental results.

2. Background and related work

2.1. Graph-processing systems

Single machines with limited resources are unable to handle growing modern graphs. *Generic* distributed data-processing systems, such as Hadoop [44], have first been adapted to analyze and process large-scale graphs on clusters. However, because of the limitation of programming models, generic data-processing systems cannot support iterative graph-processing applications very well. It has been reported that the performance of generic data-processing systems, for graph-processing applications, is much worse than *specific* graph-processing systems [27,25,9]. This has become a common knowledge in the graph-processing community.

Many graph-processing systems adapt the vertex-centric paradigm, in which graph-processing algorithms are implemented from the perspective of each vertex of graphs. The Bulk Synchronous Parallel (BSP) computing model has been used by many graph-processing systems, such as Pregel [27] and Hama [36], mainly because the BSP model simplifies the design and implementation of iterative graph-processing algorithms. A BSP computation of a graph-processing algorithm consists of a series of global iterations (or supersteps). In each iteration, active vertices execute the same user-defined function, generate messages, and transfer them to neighbors that are not located in the same machine. Synchronization is needed between two consecutive iterations to ensure that all vertices have been processed and all messages have been delivered. The cost of synchronization in BSP systems may incur performance degradation, especially when the workload between working machines is not balanced. To improve performance, graph-processing systems, such as GraphLab [25] and GraphHP [6], have used asynchronous models to avoid using barriers for synchronization and to reduce the performance degradation caused by imbalanced workload. The use of asynchronous models increases the complexity of graph-processing systems and, in some cases, creates redundant messages [46] when executing graph algorithms.

Graph-processing systems can be categorized into three main *multi-phase* systems, based on their vertex computation abstractions [28]: *one-phase* [27,9], *two-phase* [18,33,41], and *three-phase* [11,8]. The main computation in graph processing includes processing incoming messages, applying vertex updates, and preparing outgoing messages. In each multi-phase abstraction, the main computation is placed and executed in different

Table 1
Graph-processing systems using different partitioning approaches.

Partitioning approach	Example heuristics	Example systems using the approach
Traditional heuristics	METIS [19], ParMETIS [21]	–
Streaming	Hash, LDG [39]	Giraph [9], HeAPS [45]
Vertex-cut	Random, Balanced p-way [11]	PowerGraph [11], GraphX [12]
Dynamic	Exchange [35], Migration [22]	GPS [35], Mizan [22]
Chunking	File size	Hadoop [44], Stratosphere [43]

computation phases. For example, in *Scatter–Gather*, which is a two-phase abstraction, the scatter phase prepares outgoing messages, and the gather phase collects incoming messages and applies updates to vertex values. We will further analyze and discuss these three abstractions in Sections 3 and 6.

2.2. Related work

The study of partitioning policies for graph-processing is based on two main disciplines, graph partitioning and performance analysis. We survey in this section the related materials published in each of these two disciplines, in turn. Overall, ours is one of the few studies combining theoretical work in graph partitioning with experimental comparison of policies using several algorithms and datasets, which, as we indicate in the introduction, is important for the validity of the results. Our main findings from this survey, regarding graph partitioning, are summarized in Table 1.

Graph partitioning. Graph partitioning has been explored and studied for a long time in many research areas [28,20], from scientific workflow scheduling [10] to recent work on large-scale graph processing [12]. Balanced graph partitioning, which aims to balance the number of vertices in each partition while minimizing the communication between partitions, is known as the k-way graph partitioning problem and has been proved to be NP-hard [1]. To achieve an approximate solution, many *traditional heuristics* [19,34] have been proposed. Many of them adapt the *multi-level partitioning* scheme, which typically includes *three main phases* [34], coarsening to reduce the size of the graph, partitioning the reduced graph, and uncoarsening to map back partitions for the original graph. The prominent example of multi-level partitioning, METIS [19] and its family of partitioning policies [7], are used by the community because of their high-quality partitions and relatively fast partitioning speed. However, we identify three main reasons for which these heuristics may be unable to handle the partitioning problem for distributed graph-processing systems. First, most distributed graph processing systems are designed for large-scale graphs, with millions of vertices and billions of edges. For partitioning policies designed explicitly for single-node operation, such as METIS, large-scale graphs and their intermediate partitioning data often do not fit in the main memory of the system, which causes spills to disk and severe performance degradation, and in our experience even system crashes. For multi-node heuristics such as ParMETIS [21], using them in practice may be complex and time consuming, because they need a global view of graphs and slow synchronization for partitioning. Second, these heuristics are designed to operate offline. They need to access the entire graph for every partitioning operation, which makes them relatively inefficient for growing and changing graphs. Third, many of the heuristics are designed for scientific computing workloads. In particular, they have been designed to solve k-way partitioning problem, by recursively executing 2-way partitioning when k is a power of 2. They may not be able to effectively partition real-world graphs representative for other domains, and in particular real-world graphs with arbitrary values of k [2].

To address the problems faced by offline heuristics, online streaming graph partitioning policies have been proposed for distributed graph-processing systems. *Hash partitioning*, a type of streaming graph partitioning, is used in many graph processing systems, such as Pregel-like systems [27,9], because of its simplicity and short partitioning time. The drawbacks of hash partitioning for real large-scale graphs are obvious. For computation, partitions created by hash partitioning policies from highly-skewed real graphs [11] can have an even number of vertices but will often include partitions where vertices have very diverse in-/out-degrees, case in which graph-processing algorithms such as Breadth-First Search (BFS) traversal will incur high computation imbalance. For communication, hash partitioning does not consider any locality of vertices and edges. There may be an inordinate amount of edge-cuts between partitions, which results in intensive network traffic. To conclude, hash partitioning policies have so far not considered highly-skewed graphs, and result when used on real-world graphs in partitions that lead to imbalanced computation and communication.

Many studies make efforts in two main directions to obtain balanced graph partitions. The first direction is to design more complex steaming graph-partitioning policies. Stanton and Kliot [39] propose more than ten streaming policies. Many factors are selected and used in these policies, such as the relationship between the vertex to be assigned and the current vertices in the partition, buffering for assigning a group of vertices, and streaming orders. From their evaluation, a *linear-weighted deterministic greedy policy* (LDG) performs the best. In LDG, a vertex is assigned to the partition with the most neighbors, while using the remaining capacity of partitions as a penalty. Tsourakakis et al. [42] formulate a partitioning *objective function*, considering the costs of edge cut and the size of partitions. Based on this function, they design a streaming graph partitioning, FENNEL, which is a greedy policy using different heuristics to place vertices. Closest to our work, to address heterogeneity of computing hardware and network, Xu et al. [45] build a model for the heterogeneous environment and discuss a time-minimized objective function from the perspective of graph-processing systems. They propose six streaming graph partitioning policies and evaluate their performance in both homogeneous and heterogeneous environments. From their experimental results, the *combined policy* (CB) achieves the best performance in homogeneous environment and reasonably good performance in different settings of heterogeneous environment. They use the analytical method to estimate the workload of the whole computation. In our model, we further divide the whole computation and use real experiments to find out run-time-influencing graph characteristics. Our method can be more precise. Advanced streaming graph partitioning policies can achieve comparable performance of METIS [39,45].

The second direction is to partition graphs by *vertex-cut* [11,12]. Vertex-cut partitioning places edges, instead of vertices, to different partitions. According to percolation theory [40], good vertex-cuts can be achieved in power-law graphs. Evenly placing edges can reduce the workload imbalance and the large communication of high-degree vertices, which are represented as multiple replicas and stored in different partitions. Vertex-cut partitioning has its drawbacks. System-wise, the graph-processing system needs to allow a single vertex's computation to span multiple machines, which increases the complexity of the system. Performance-wise, too many pieces of vertex replicas can still generate high communication, primarily to synchronize vertex status. We summarize our survey of this class of graph-partitioning policies in Table 1, in the row "Vertex-cut". Vertex-cut partitioning is used by few graph-processing systems. In our work, we focus on edge-cut partitioning, which is used by more systems.

To avoid the workload imbalance incurred by static streaming partitioning and vertex-cut partitioning and also by the execution of algorithms (for example, active vertices vary in each iteration during the process of the BFS algorithm), *dynamic repartitioning* is moving vertices between working machines during the execution of algorithms. The general process of dynamic repartitioning methods can be abstracted as the following sequence of four steps: discover workload imbalance of computing machines, find the pairs of computing machines for migrating vertices, determine which vertices are required to move, and migrate selected vertices from its source to destination. Mizan [22] selects the execution time of each machine as the metric for workload imbalance and maintains a distributed hash table to record the position of vertices. GPS [35] simply uses the outgoing messages as the workload-imbalance metric. When computing machines are paired, they will exchange vertices rather than migrate vertices from one to another. Both Mizan and GPS take a *delay migration strategy* to alleviate the overhead of migration of vertices and their associated data. Shang et al. [37] focus on how much of the workload should be moved between pairs of working machines and on which vertices should be moved. They also propose several constraints to improve the benefit of migration. We show systems that support dynamic repartitioning in Table 1, in the row "Dynamic".

Partitioning performance. Although many graph partitioning methods and policies have been proposed, their performance has not been thoroughly evaluated with various input graphs and algorithms. Theoretical metrics, such as the edge cut ratio and modularity [39,42,3] are generally used to measure the quality of partitions. For real graph-processing systems, these metrics do not directly represent the performance of partitioning [45]. In practice, metrics such as the run time of graph-processing algorithms, partitioning time, and the variance of the run time on different machines/threads represent the performance of bottleneck components in real graph-processing systems. Meyerhenke et al. [29] design their graph partitioning heuristic based on label propagation and size constraints for social networks and web graphs. Guerrieri and Montresor [14] discuss the properties of high quality partitions and introduce a distributed edge-partitioning framework. Both studies lack experimental results from executing algorithms on real graph processing systems, to show the performance of their partitioning methods in practice. Stanton and Kliot [39], FENNEL [42], and Xu et al. [45] compare the performance of many streaming partitioning policies on data-processing systems. The systems they run experiments on are not (advanced) graph-processing systems—Spark's generic data processing for Stanton and Kliot, Hadoop for FENNEL, and a prototype of Pregel for Xu et al., contrast starkly with highly optimized production systems such as GraphLab [25] and Giraph [9]. Their evaluations are also limited to the use of a single algorithm, PageRank; our own and related studies [15,26,16] have shown that the results obtained from a single algorithm do not characterize well the performance expected from the general field of graph processing. In contrast, in this work, we conduct comprehensive experiments on an advanced distributed graph-processing system—PGX.D, using 3 representative algorithms, 3 large-scale graphs with billions of edges from different domains, different practical configurations and in particular different types of network, and many different performance metrics.

3. A model of graph-processing systems and the objective function of graph partitioning

In this section, we model the run time of different types of graph-processing systems and we discuss the objective function of graph partitioning of real graph-processing systems. We focus on graph-processing systems that follow the BSP programming model, that is, for which the graph-processing algorithm is executed in super-steps or iterations. Our model focuses on two-phase systems (described later in this section), but it can also represent single-phase systems such as the Pregel-based Apache Giraph. We consider in our model machine-level and thread-level programming abstractions, and blocking and parallel I/O. Conceptually, our model derives non-trivially from prior work; in contrast to the prior model of Xu et al. [45], which is the closest related work to our present study, our model considers a much larger variety of systems and has a higher granularity of processing units.

Similarly to the model of Xu et al. [45], suppose we have M working machines running N iterations of the same process. If T_i^k is the run time on machine i of the kth iteration of some application, and if T^k denotes the (total) run time of the kth iteration across all machines, then we have:

$$T^k = \max_i\{T_i^k\}, \quad k = 1, 2, \ldots, N. \tag{1}$$

The total run time T_r of the application running on multiple machines can now be presented as:

$$T_r = \Sigma T^k, \quad k = 1, 2, \ldots, N. \tag{2}$$

We assume conservatively that in each iteration all vertices are active (that is, considered for processing) and that messages are sent to all their neighbors, for three reasons. First, many popular algorithms match well this assumption, such as community detection [32] and PageRank [31]. Second, previous policies, and in particular the commonly used family of policies based on METIS, partition the whole graph with all its vertices and edges, so they implicitly follow this assumption. Last, predicting, for different algorithms, which of the vertices and edges become active during an arbitrary iteration is an open and challenging problem, but not a part of real-world graph-processing systems. Currently, no real graph-processing system is able to make prediction-based workload balancing in each iteration. We further discuss how the variety of algorithms complicates prediction in Section 6.3. Under this conservative assumption, the run time of every iteration on each machine can be considered to be equal, say to value $\overline{T}_i$, and so we can simplify Eq. (2) to:

$$T_r = N \times \max_i\{\overline{T}_i\}. \tag{3}$$

From the survey [28], there are three vertex-centric programming abstractions of graph-processing systems: one-phase abstraction, two-phase abstraction, and three-phase abstraction. For each iteration, the one-phase programming abstraction runs a single computation function, which consists of three computation tasks: processing incoming messages, applying vertex values, and preparing outgoing messages. The communication starts after the completion of the single computation function. The one-phase abstraction is often used in practice, for example in Pregel-like systems [27,9]. The two-phase abstraction usually refers to two computation phases: the scatter phase (for preparing outgoing messages) and the gather phase (for processing incoming messages and applying vertex values). The communication happens between the scatter phase and the gather phase. The two-phase abstraction has been implemented in systems such as PGX.D [18]. Importantly, most one-phase systems can be converted to two-phase

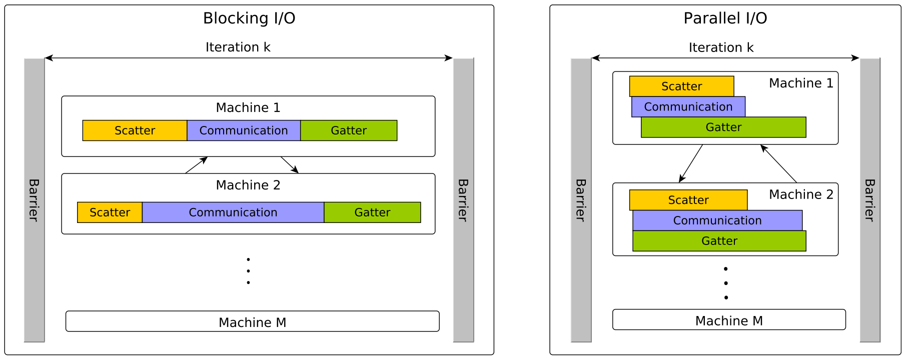

Fig. 1. The computation phases and communication in one iteration of the Scatter–Gather abstraction.

Table 2
Notations for the time of computation and communication of machine *i*.

Symbol	Meaning
$\overline{T}_i^g$	Time spent processing incoming messages and applying vertex values across all threads.
$\overline{T}_{i,l}^g$	Time spent processing incoming messages and applying vertex values in the *l*th thread, $l = 1, 2, \ldots, L$.
$\overline{T}_i^s$	Time spent preparing outgoing messages across all threads.
$\overline{T}_{i,q}^s$	Time spent preparing outgoing messages in the *q*th thread, $q = 1, 2, \ldots, Q$.
$\overline{T}_i^x$	Time spent in communication, data transfers.
L	Number of threads involved in processing incoming messages and applying vertex values.
Q	Number of threads involved in preparing outgoing messages.

systems [28], but the reverse may not be true. We summarize in Table 2 the notation we propose for the time of the computation tasks and for the communication. The three-phase systems usually use the vertex-cut partitioning, which is out of the scope for this work. We further discuss three-phase systems, as a future extension of our modeling work, in Section 6.2.

Graph-processing systems can use one of the following two I/O modes, between computation and communication: *blocking I/O* and *parallel I/O*. With blocking I/O, computation and communication are executed serially. With parallel I/O, computation and communication can execute in parallel, with at least parts of the execution overlapped. For blocking I/O, $\overline{T}_i$ is the sum of the time spent on all computation phases and on communication. For parallel I/O, $\overline{T}_i$ is determined by the longest among the two computation phases and communication. We show in Fig. 1 two computation phases and communication in one iteration of the Scatter–Gather abstraction.

Another important aspect of graph processing that we consider in our model is the granularity of the programming abstraction. In real graph-processing systems, where *multi-threading* has been used to accelerate computation, the run time of a computation phase is determined by the thread with the longest run time.

Table 3 summarizes the run time of a single iteration executed on machine i for different programming abstractions and I/O modes, in coarse-grained *machine-level* and fine-grained *thread-level*. Because the one-phase abstraction uses a single computation function, all computations for a vertex are always executed by the same thread, which means processing incoming messages and applying vertex updates cannot be parallelized with preparing outgoing messages. For the parallel I/O mode of the two-phase abstraction, the threads of a working machine need to be assigned to different computation phases to gain all the possible performance through parallelism. Thus, the assignment of the threads is an important factor for the run time of working machines. Moreover, for threads in the same phase, being able to balance their workload is crucial for achieving high performance.

The models we summarized in Table 3 are used to determine the graph characteristics that may have an impact on the run time of graph-processing systems (see Sections 4.1 and 4.2). However, the models cannot be used to (precisely) predict the run time of graph-processing systems, because the relationships between every time component (such as $\overline{T}_i^g$) of the models and the graph characteristics are not explored. It is non-trivial to formulate uniform relationships for various graphs, datasets, and systems.

The main target of partitioning graphs for real graph-processing systems is to achieve the shortest run time. Similarly to Xu et al. [45], we set the objective function for finding a graph partitioning that minimizes the total run time T_r:

$$\min\{T_r\} = N \times \min\{\max_i\{\overline{T}_i\}\}. \quad (4)$$

In the following section, we investigate what are the interesting graph characteristics that affect the run time of the computation phases and communication, and we use this information to design new partitioning policies.

4. Design of graph partitioning policies

The aim of this section is to design good graph-partitioning policies. In order to do so, we want to identify the graph characteristics that have significant impact on the run time of graph-processing systems. In Section 4.1, we propose a method for identifying such graph characteristics, and in Section 4.2 we empirically validate this method in the PGX.D graph-processing system. Then in Section 4.3 we design new streaming graph-partitioning policies according to the graph characteristics we identified.

4.1. A method for identifying the run-time-influencing graph characteristics

As many popular graph-processing systems [27,9] can only process directed graphs, we consider without loss of generality graph-processing systems that use a directed graph representation. In Table 4 we distinguish a number of characteristics of a partition of a directed graph that may have an impact on the run times of graph processing algorithms. Our target is to identify the graph characteristics that actually have the strongest such impact. We propose the

Table 3
The time for one iteration ($\overline{T}_i$) for different programming abstractions and I/O modes.

System	I/O block, machine-level	I/O block, thread-level	I/O parallel, machine-level	I/O parallel, thread-level
One-phase	$\overline{T}_i^g + \overline{T}_i^s + \overline{T}_i^x$	$\max(\overline{T}_{i,l}^g + \overline{T}_{i,l}^s) + \overline{T}_i^x$	$\max(\overline{T}_i^g + \overline{T}_i^s, \overline{T}_i^x)$	$\max(\max(\overline{T}_{i,l}^g + \overline{T}_{i,l}^s), \overline{T}_i^x)$
Two-phase	$\overline{T}_i^g + \overline{T}_i^x + \overline{T}_i^s$	$\max(\overline{T}_{i,l}^g)+\overline{T}_i^x+\max(\overline{T}_{i,q}^s)$	$\max(\overline{T}_i^g, \overline{T}_i^s, \overline{T}_i^x)$	$\max(\max(\overline{T}_{i,l}^g), \max(\overline{T}_{i,q}^s), \overline{T}_i^x)$

Table 4
The characteristics of a partition of a graph.

Characteristic	Symbol	Definition
Number of vertices	$\#V$	Vertex count
Remote in-degree	D_{ri}	The number of in-edges from other partitions
Remote out-degree	D_{ro}	The number of out-edges to other partitions
Local in-degree	D_{li}	The number of in-edges in the partition
Local out-degree	D_{lo}	The number of out-edges in the partition (equal to local in-degree)
Total in-degree	D_{ti}	The sum of remote in-degree and local in-degree
Total out-degree	D_{to}	The sum of remote out-degree and local out-degree
Remote degree	D_r	The sum of remote in-degree and remote out-degree
Local degree	D_l	The sum of local in-degree and local out-degree
Total degree	D_t	The sum of remote degree and local degree

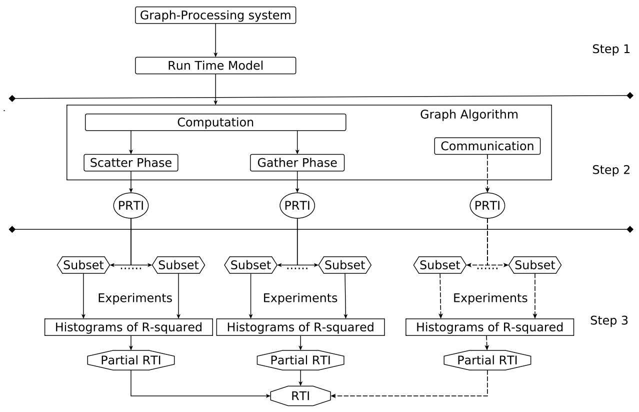

Fig. 2. Our 3-step method for identifying the run-time-influencing graph characteristics of a two-phase system. (The selection from the communication component may not be performed if the communication is overlapped by the computation, depicted as dashed lines and further discussed in Section 4.2.)

following three-step method to achieve this, which is illustrated for a two-phase system in Fig. 2.

Step 1: Determine the run time model of the graph-processing system from the possibilities listed in Table 3.

Step 2: Determine the Potential Run-Time-Influencing (PRTI) graph characteristics that may have an impact on the run time given the model determined in Step 1. These characteristics represent the candidate set for Step 3 of our method. The PRTI graph characteristics may vary for different graph-processing algorithms and perhaps even for the different components (e.g., computation and communication) of the same graph-processing algorithm, and for the model of the graph-processing system. For each component (even each phase if the model includes multiple phases) of the algorithm, we select a set of PRTI graph characteristics according to the graph entities operated by the graph algorithm. For example, the number of vertices ($\#V$) is always selected for the scatter phase (one phase of the computation component) because vertices are processed during the computation, and the remote out-degree (D_{ro}) is selected for the communication component if the algorithm sends messages by remote out-going edges.

Step 3: Identify from the PRTI graph characteristics the actual Run-Time-Influencing (RTI) graph characteristics that are strongly related to the run time of (a phase or component of) an algorithm. In order to do so, we first create different candidate subsets from each set of PRTI graph characteristics for the partial set of RTI graph characteristics. We will show how to create these subsets in Section 4.2. We take an experimental approach to pick the appropriate subset. For each experiment, we measure the run time of each working machine and we calculate the values of the graph characteristics of the partition stored on it shown in Table 4. For each candidate subset, we conduct a linear regression [30] between the run times of the working machines and the values of the graph characteristics in that subset of the partitions assigned to them. In this way, we obtain a value of the R-squared (R^2) coefficient from every experiment.

We perform multiple experiments using different setups (in terms of system configurations, datasets, and graph-partitioning policies) and we build a histogram with the numbers of occurrences of the R^2 value in given ranges (for an example, see Table 7). We select as the partial set of RTI graph characteristics the subset of PRTI with the most occurrences in the highest range of R^2 values. After having obtained the partial sets of RTI characteristics of multiple phases/components of an algorithm, they can be combined to form the set of RTI characteristics of the whole algorithm. The RTI graph characteristics are strongly determined by the behavior of graph algorithms and the model of graph-processing systems. Using different datasets may affect the coefficients of the linear regression between the run times and the values of the subset of PRTI graph characteristics, but not affect the distribution of the R^2 values. So, the obtained RTI graph characteristics are also applicable for other graphs that are not used in the experiments.

4.2. Empirical results validating the method

We will now empirically validate the method from the previous section for the PGX.D graph-processing system.

Step 1: We use Table 3 to identify the run time model corresponding to PGX.D. As PGX.D is a multi-threaded graph-processing system with two-phase abstraction and parallel I/O, its run time model is:

$$\overline{T}_i = \max(\max(\overline{T}_{i,l}^g), \max(\overline{T}_{i,q}^s), \overline{T}_i^x). \tag{5}$$

Step 2: We seek to understand the operation of PGX.D in order to select the PRTI characteristics. In PGX.D, the threads assigned to the scatter phase and the gather phase are called worker threads and copier threads, respectively. PGX.D balances the workload across its worker threads with the edge-chunking technique and across its copier threads with the max-slot first strategy. So, $\max(\overline{T}_{i,l}^g)$ and $\max(\overline{T}_{i,q}^s)$ are equal to the average run time of worker threads and copier threads, respectively. PGX.D uses the continuation mechanism to buffer and combine messages between working machines to reduce communication. A dedicated poller thread is maintained in each working machine for sending and receiving messages. PGX.D implements a commonly used technique, called Selective Ghost Node (SGN), to further reduce the network traffic. SGN duplicates the high-degree vertices (ghosts) in each partition. A vertex is selected as a ghost if the sum of its in-degree and out-degree is larger than a pre-defined threshold. The use of SGN is optional for users.

We apply Step 2 of our method on different components of the PageRank algorithm. The scatter phase in PageRank reads all vertices and prepares messages to remote neighbors through the out-edges of each vertex. Therefore, the PRTI graph characteristics of the scatter phase are the number of vertices, the remote out-degree, the local out-degree, and the total out-degree. The gather phase in PageRank processes all incoming messages and updates each vertex, and so, the PRTI graph characteristics of the gather phase are the number of vertices and the remote in-degree. The only PRTI graph characteristic of the communication component in the PageRank algorithm is the remote out-degree.

Step 3: In order to identify the RTI graph characteristics for PGX.D, we perform experiments with PageRank (maximum 10 iterations) with PGX.D deployed on a 16-machine cluster in Oracle Labs with properties as in Table 5. We explain the experimental setup in terms of the system configurations, the datasets, and the partitioning policies that we employ.

We use four system configurations of worker and copier threads: w24c2, w18c8, w12c14, and w6c20 [18], where the notation w24c2 means that we set 24 worker threads and 2 copier threads in each working machine, etc. We conduct experiments with or without using the SGN technique.

We use three large-scale graphs, Twitter, Scale_26, and Datagen_p10m (see Table 6). The Twitter dataset is one of the largest publicly available real-world datasets and consists of a graph of its users with the follower relationships between them. Scale_26 is a synthetic graph generated by the Graph500 generator, with a scale factor of 26. Graph500 is the de-facto standard for comparing hardware infrastructures for graph processing systems. Datagen_p10m is created by the Linked Data Benchmark Council (LDBC) generator, which aims to produce graphs with structures and properties similar to those of real-world social networks, such as Facebook. The LDBC generator is used by the Graphalytics project [4], which is an active big data benchmark for graph-processing systems. The two generated graphs contain roughly 1 billion edges, and are comparable in size to the Twitter dataset. We set the threshold for ghost selection of SGN to 50,000 for Twitter and Scale_26, and to 600 for Datagen_p10m.

We use three streaming graph-partitioning policies incorporated in PGX.D, viz. the in-degree balanced policy (I), the out-degree balanced policy (O), and the total degree balanced policy (IO). All policies assign vertices to partitions by balancing the in-degree, the out-degree, or the total degree across partitions. To this end, each policy first determines the average (in-/out-/total) degree per partition, and then assigns vertices sequentially to the partitions, going from one partition to the next when the (in-/out-/total) degree of the former exceeds the corresponding average.

We obtain 72 executions of PageRank by combining all system configurations with or without SGN (4x2), all datasets (3), and all partitioning policies (3). We find that the run time of PageRank is dominated by either the scatter phase or the gather phase. As PGX.D optimizes the network traffic and is deployed on the high-speed InfiniBand network, the communication time in PageRank is overlapped by both the scatter and gather phases. In Fig. 3, we show the run time of a single iteration, the run time of the longest worker thread, and the run time of the longest copier thread of PageRank for Twitter when using the w12c14 configuration. We notice that for all policies the run time of a single iteration is approximately 50 ms higher than the maximum run time of the longest worker and copier threads, which is due to the overhead of the system. We have similar findings for other system configurations. Thus, we only need to consider the scatter and gather phases to select the RTI graph characteristics.

Table 5
The environment of our experiments.

Category	Item	Detail
CPU	Type	Intel Xeon E5-2660
	Frequency	2.20 GHz
	Parallelism	2 socket * 8 core * 2 HT
Network	Card	Mellanox Connect-IB
	Switch	Mellanox SX6512
	Raw BW	56 Gbit/s (per port)
Software	OS	Linux 2.6.32 (OEL 6.5)
	Compiler	gcc 4.9.0

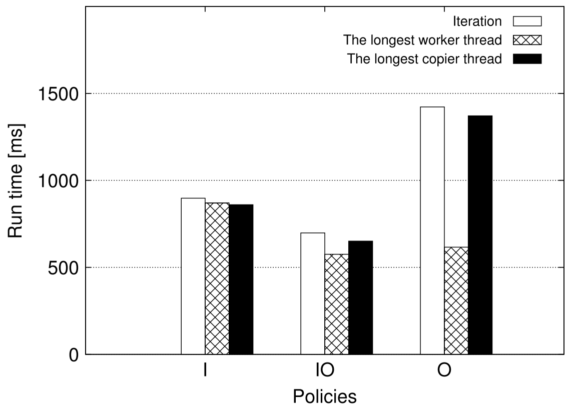

Fig. 3. The run time of an iteration, of the longest worker thread, and of the longest copier thread of PageRank for Twitter using system configuration w12c14.

For each execution, we measure the run times of the scatter and gather phases, respectively, of the sixteen working machines and we calculate for each machine the values of the graph characteristics of its graph partition. We consider different candidate subsets of RTI characteristics from the PRTI set, which we evaluate empirically in order to determine the RTI graph characteristics. We could consider all subsets of the PRTI set as candidates, but in practice, we can use previous knowledge to consider fewer subsets. For example, we derive three subsets of characteristics from the PRTI set of the scatter phase: the number of vertices ($\#V$) and the total out-degree (D_{to}), only D_{to}, and the local out-degree (D_{lo}). We consider the subset consisting of only D_{to} because many previous partitioning policies focus on it. The subset of $\#V$ and D_{to} is considered because during our experiments we found that some partitions with similar D_{to} but different $\#V$ have (significantly) different run times. The subset of D_{lo} is randomly created as a control subset in order to show how weak the relationship between the run time of the scatter phase and a randomly selected graph characteristic can be.

For every candidate subset for each phase we create a histogram of the R^2 values for all 72 experimental setups. In Table 7 we show the histograms for the scatter phase. From this table, we identify as the RTI graph characteristics of the scatter phase the number of vertices ($\#V$) and the total out-degree (D_{to}) because they have the highest number of values of R^2 in the range of [0.9, 1]. Unlike previous policies, which focus on the communication component by minimizing edge-cuts, our results show that the number of vertices is also an important factor. Similarly, for the gather phase, we identify the number of vertices ($\#V$) and the remote in-degree

Table 6
Summary of datasets.

Dataset	V	E	d	$\bar{D}$	Q1	Q2	Q3	Max D	Type & Source
Twitter	41,652,230	1,468,365,182	8	35	5	13	34	3,081,112	Real-world, Public [23]
Scale_26	32,804,978	1,073,741,824	10	33	1	4	17	1,710,236	Synthetic, Graph500 [13]
Datagen_p10m	9,749,927	687,174,631	72	70	30	87	204	648	Synthetic, LDBC [24]

V and **E** are the vertex count and edge count of the graphs. **d** is the link density ($\times 10^{-7}$). $\bar{\mathbf{D}}$ is the average vertex out-degree. **Q1**, **Q2**, and **Q3** are the first quartile, median, and the third quartile of vertex total-degree, respectively. **Max D** is the largest vertex total-degree.

Table 7
The numbers of experiments with PageRank that have the value of R-squared (R^2) for the scatter phase in the indicated ranges.

Range	$\overline{T}_i^g$ with $\#V$ and D_{to}	$\overline{T}_i^g$ with D_{to}	$\overline{T}_i^g$ with D_{lo}
[0.9, 1]	39	28	10
[0.8, 0.9)	8	11	7
[0.7, 0.8)	9	9	1
[0.6, 0.7)	5	4	6
[0, 0.6)	11	20	48

(D_{ri}) as the RTI graph characteristics. Combining the results of both phases we identify as the complete set of RTI graph characteristics for PageRank the number of vertices ($\#V$), the total out-degree (D_{to}), and remote in-degree (D_{ri}).

We have conducted a similar set of experiments for the weakly connected component (WCC) algorithm, which computes the maximal groups of vertices connected by edges. The RTI graph characteristics of WCC are the number of vertices ($\#V$), the total degree (D_t), and the remote degree (D_r).

4.3. Four new graph partitioning policies

In this section, we design four new graph-partitioning policies based on the findings from the experiments in Section 4.2. The first of these, called the *degree-balanced* (DB) policy, is new, while the other three of these are randomized versions of the I, IO, and O policies from Section 4.2.

Our target is to design a good partitioning for graph-processing systems in general, not for a specific algorithm. Combining the RTI graph characteristics identified by running PageRank and WCC, we show that the number of vertices is a common characteristic. For different algorithms, they may propagate messages through in- or out-edges. It is difficult to determine which graph characteristics about degree we should balance. We decide to select total in-degree and total out-degree, because of two main reasons. First, the remote or local degree of a partition can only be calculated after the finish of partitioning. We cannot use them during the execution of partitioning. Second, from the perspective of the system, balancing the total in-degree and total out-degree is a generic way to cover different algorithms. Thus, the primary purpose of our DB policy is to balance the total in- and out-degree per partition, and its secondary purpose is to balance the sum of the in-degree and out-degrees across the partitions by setting a constraint on the number of vertices of the partitions.

With DB, every next vertex is assigned to the degree-smallest of what we call the opposite partitions. For a vertex with in-degree V_i and out-degree V_o, a partition with total current in-degree D_{ti} and total current out-degree D_{to} is called *opposite* if $V_i > V_o$ and $D_{ti} \leq D_{to}$, or the other way around. The *degree-smallest* partition is the partition with the smallest sum of its current total in-degree and total out-degree. We set a *constraint* on the number of vertices per partition to ensure that they do not become too imbalanced. In the DB policy, this constraint is flexible and can be set by the user. The process of assigning a vertex to a partition by the DB policy is shown in Policy 1.

In order to show the balance of the partitions created by the DB policy, we apply it to the three datasets (Twitter, Scale_26, and

Policy 1 The DB policy

```
Input: $V_i$, $V_o$, the constraint on the number of vertices $C$, a sorted
    queue of partitions $P[M]$ with ascending $D_{ti} + D_{to}$, the number
    of partitions $M$
Output: the index of the assigned partition $Index$, a sorted queue
    of partitions after the assignment
 1: $Flag \leftarrow 0$   ▷Flag indicates if there is an opposite partition of the vertex in the queue.
 2: if $V_i > V_o$ then
 3:     for $j = 1 \rightarrow M$ do
 4:         if $D_{ti}^j \leq D_{to}^j$ then   ▷$D_{ti}^j$ and $D_{to}^j$ is the current total in-degree and the current total out-degree of the jth partition $P^j$.
 5:             Assign the vertex to $P^j$, update $D_{ti}^j$ and $D_{to}^j$.
 6:             $Flag \leftarrow 1, Index \leftarrow j$
 7:             break
 8:         end if
 9:     end for
10:     if $Flag = 0$ then   ▷Cannot find an opposite partition for the vertex.
11:         Assign the vertex to $P^1$, update $D_{ti}^1$ and $D_{to}^1$.   ▷Assign the vertex to the smallest/first partition.
12:         $Index \leftarrow 1$
13:     end if
14: else if $V_i < V_o$ then
15:     for $j = 1 \rightarrow M$ do
16:         if $D_{ti}^j \geq D_{to}^j$ of $P^j$ then
17:             Assign the vertex to $P^j$, update $D_{ti}^j$ and $D_{to}^j$.
18:             $Flag \leftarrow 1, Index \leftarrow j$
19:             break
20:         end if
21:     end for
22:     if $Flag = 0$ then
23:         Assign the vertex to $P^1$, update $D_{ti}^1$ and $D_{to}^1$.
24:         $Index \leftarrow 1$
25:     end if
26: else   ▷$V_i = V_o$
27:     Assign the vertex to $P^1$, update $D_{ti}^1$ and $D_{to}^1$.   ▷Assign the vertex to the smallest/first partition.
28:     $Index \leftarrow 1$
29: end if
30: if $\#V$ of $P^{Index} \geq C$ then
31:     Remove $P^{Index}$ from the queue.
32:     $M \leftarrow M - 1$
33: end if
34: Ascending sort the partition queue $P[M]$ by $D_{ti} + D_{to}$ of each
    partition
```

Datagen_p10m) to create 16 partitions each. We set the constraint on the size of the partitions to 1.5 times their average size (we assume the size of the graph to be known ahead of time). In order to show the balance, we normalize the number of vertices, the total in-degree and the total out-degree of each partition relative to their average values across all partitions. Fig. 4 shows that the graph characteristics are very well balanced for the Twitter partitions. For the Scale_26 and Datagen_p10m graphs, we achieve similar

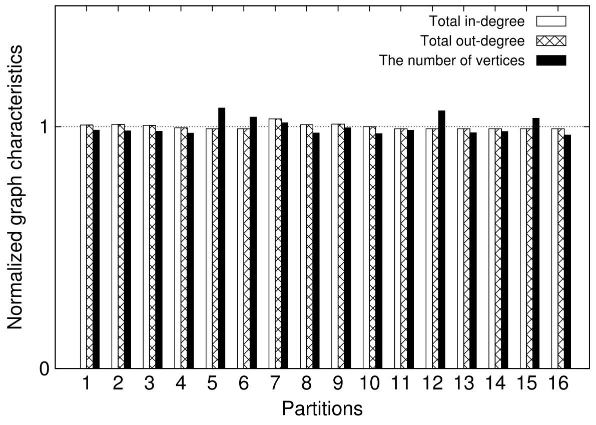

Fig. 4. The normalized values of the graph characteristics achieved by the DB policy for Twitter.

results. We have also partitioned the graphs into different numbers of partitions (2, 4, 8, and 32), and also then we achieve balanced partitions. Our results even indicate that we can achieve balanced numbers of vertices without setting a constraint.

From the experimental results in Section 4.2, we find that the run time of the machines varies even though they have equal numbers of edges to process in the I, IO, and O policies. The reason is that the numbers of vertices of the partitions, which are run-time-influencing, are not balanced. To address this issue, we change the streaming order of the vertices in these policies, from the sequential ordering to a *random ordering*, which accesses vertices randomly. There are also other stream orderings, such as the *BFS ordering* and the *DFS ordering*. We select the random ordering for three main reasons. First, from the evaluation of Stanton and Kliot [39], the random ordering has comparable performance to the BFS and DFS orderings in many cases. Second, the BFS and DFS orderings need to pre-traverse the graphs, which is time consuming, in particular for large graphs. The traverse time may be even longer than the partitioning time. Third, the BFS and DFS orderings can be more complicated when a graph has multiple connected components. By using the random ordering of each original policy in PGX.D, we create three new policies called RI, RIO, and RO, in which "R" stands for the random ordering. Fig. 7 shows a comparison of the O and RO policies. The RO policy achieves more balanced numbers of vertices across partitions, while keeping the balance of the total degrees.

Many graphs are not static, but mutate over time. Although we only cover static graphs in our experiments, our partitioning policies can be used to partition mutating graphs online as well, obviating the need to re-partition a graph after it has changed. For example, the DB policy does not need to know meta information of the graph (such as the number of vertices and edges) or the neighborhoods of vertices to assign vertices. When partitioning a mutating graph, it can simply assign new vertices one-by-one based on its rules, and update the meta information of every partition (such as the total in-degree and total out-degree). However, many graph-processing systems cannot support online graph-partitioning policies and process mutating graphs. We are not able to show the ability of partitioning mutating graphs of our policies in our experiments.

5. Experimental results

In this section we conduct comprehensive experiments with different graph partitioning policies, applications, and system configurations. In Section 5.1 we present our experimental setup, and at the end of the section we explain the experiments reported in later sections.

5.1. Experimental setup

Experimental environment: We keep using the same cluster as shown in Table 5. Besides using InfiniBand, in Section 5.5 we also evaluate the performance on 1 Gbit/s Ethernet. We run all experiments on 16 working machines, except for the scalability test in Section 5.4, in which we use four different numbers of machines (2, 4, 8, and 32).

Datasets: We will only present the results of executing graph-processing algorithms on large-scale graphs. In fact, we have also run experiments on a smaller graph, Livejournal [38] (with 4,847,571 vertices and 68,993,773 edges). However, the performance differences of the graph-partitioning policies are quite small in that case. In Section 5.6, we include four more Graph500 graphs than we have used in Section 4.2, with the scale factor running from 22 to 25. For these graphs, the numbers of vertices and edges are doubled with every step of the scale factor.

Algorithms: We have conducted a comprehensive survey of graph-processing algorithms [17]. Our survey covers over 100 research articles published in 10 representative conferences (including VLDB, SIGKDD, SIGMOD, etc.) in recent years. Graph algorithms in previous publications can be categorized into different classes by functionality. We find that the top 3 occurred classes of algorithms are graph traversal, general statistics, and connected components. The percentages of the occurrence of these 3 classes of algorithms are 46.3%, 16.1%, and 13.4%, respectively. In total, they have about 70% occurrence among all types of algorithms. We select one exemplar algorithm from each of these 3 classes, Breadth-First Search (BFS) from graph traversal, PageRank from general statistics, and Weakly Connected Components from connected components. PageRank and BFS propagate updates through out-edges. WCC propagates updates through both in- and out-edges, and does not need any parameter. For PageRank, the termination condition is set to maximum 10 iterations. For BFS, we select the same source vertex for each graph for all partitioning policies.

Partitioning policies: In total, we evaluate 12 graph-partitioning policies: 2 streaming policies (R and H) commonly used by graph-processing systems, 2 streaming policies (LDG and CB) from the literature, the 3 original streaming policies (I, IO, and O) used in PGX.D, our 4 new streaming policies (RI, RIO, RO and DB) presented in Section 4.3, and the state-of-the-art partitioner (M). Except for RI, RIO, and RO, all policies use the sequential ordering of the graphs. We summarize the partitioning policies in Table 9. According to the experimental results of the CB policy [45], we set its degree threshold percentage to 30 %.

The experiments we have conducted are as follows:

- In Section 5.2, we evaluate the impact of the configurations of worker threads and copier threads.
- In Section 5.3, we measure the workload imbalance of partitions by using the edge cut ratio and the standard deviation of normalized run-time-influencing graph characteristics.
- In Section 5.4, we show the run time of graph-processing algorithms with different datasets. We also present the scalability of each partitioning policy.
- In Section 5.5 we report the performance of using Ethernet and the impact of using the selective ghost node technique.
- In Section 5.6 we investigate the time spent on graph partitioning, considering different numbers of partitions and graph sizes.

A summary of the experiments, and of the remaining sections, is in Table 8.

Table 8
Experimental setup for each experiment in Section 5.

Section	Algorithms	Datasets	Metrics	Threads	Network	SGN technique
Section 5.2	PageRank	Twitter	Run time	All	InfiniBand	No
Section 5.3	PageRank	Twitter, Scale_26, Datagen_p10m	ECR, SD	w12c14	InfiniBand	No
Section 5.4	All	Twitter, Scale_26, Datagen_p10m	Run time, scalability	w12c14	InfiniBand	No
Section 5.5	All	Twitter, Scale_26, Datagen_p10m	Performance ratio	w12c14	InfiniBand, Ethernet	Yes
Section 5.6	–	Twitter, from Scale_22 to Scale_26	Partitioning time	–	–	–

Table 9
Twelve partitioning policies in our experiments.

Policy	Streaming	Mechanism
R	Yes	Randomly assign a vertex to a partition.
H [27]	Yes	Hash partitioning.
LDG [39]	Yes	Assign a vertex to the partition, which has most neighbors of the vertex.
CB [45]	Yes	Assign a vertex to a partition with the smallest workload or with the least incremental workload.
I [18]	Yes	Balance the in-degree of partitions, original policy in PGX.D.
IO [18]	Yes	Balance the total-degree of partitions original policy in PGX.D.
O [18]	Yes	Balance the in-degree of partitions, original policy in PGX.D.
RI	Yes	The I policy using random ordering, proposed in this work.
RIO	Yes	The IO policy using random ordering, proposed in this work.
RO	Yes	The O policy using random ordering, proposed in this work.
DB	Yes	The greedy degree-balanced policy, proposed in this work.
M [19]	No	METIS, multi-level graph partitioning.

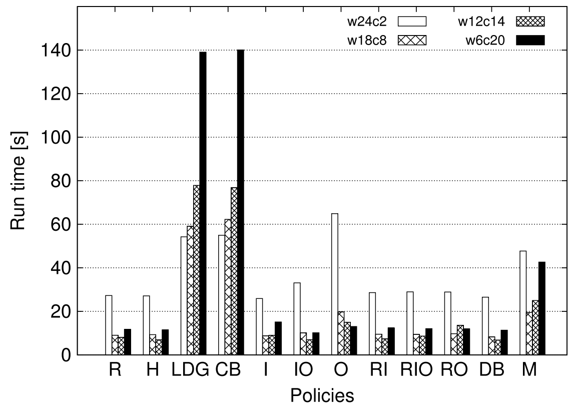

Fig. 5. The run time of PageRank for Twitter with four thread configurations.

5.2. *The impact of the configuration of worker and copier threads*

There are many possible configurations with different numbers of worker threads and copier threads. The configuration of worker threads and copiers threads can significantly influence the performance of PGX.D [18]. In this section, we explore the impact of the thread configuration on 12 partitioning policies.

Key findings:

- The configuration of worker and copier threads has a significant impact on the run time of PGX.D for all partitioning policies.
- In most experimental runs, the thread configuration w12c14 shows the best performance.

We use four configurations, w24c2, w18c8, w12c14, and w6c20, which give a reasonable coverage of the possible configurations. Fig. 5 shows the run time of PageRank for the Twitter dataset. In general, the best performance is obtained from either w12c14 or w18c8 for different partitioning policies. We also conduct other groups of experiments, with different algorithms, datasets and machines. In most cases, the configuration of w12c14 achieves the best performance, and so we empirically use this as our default thread configuration for the following experiments.

5.3. *Workload distribution*

In this section we discuss the workload distribution among working machines. The workload includes two parts, the communication workload between working machines and the computation workload on each machine.

Key findings:

- The edge cut ratio is not a good indicator for the quality of partitioning for real graph-processing systems, at least when communication is not the performance bottleneck of the system.
- The standard deviation of the normalized run-time-influencing graph characteristics can be used to measure the imbalance of the computation workload.
- The design of partitioning policies should not only focus on minimizing the communication, but also on balancing the communication between pairs of machines.

The edge cut ratio (ECR) is defined as the ratio of the number of edges that connect vertices that are placed in two partitions over the total number of edges in the graph. ECR is used by many previous studies to measure the total communication workload. We show the ECR of the 12 partitioning policies on Twitter, Scale_26, and Datagen_p10m in Fig. 6. Because CB, LDG, and M consider the neighborhoods of the vertex to be assigned and of the already assigned vertices in each partition, they are the top 3 policies that achieve the lowest ECR for all three datasets (except that LDG ranks sixth for Datagen_p10m). In contrast, the ECR of other policies is very high, because they assign vertices without considering their neighborhoods.

We use the standard deviation (SD) of the normalized (see Section 4.3 for the normalization) run-time-influencing (RTI) graph characteristics (i.e., the number of vertices, total out-degree, and total in-degree) to understand the computation workload across working machines. Fig. 7 shows the results for Twitter, which is partitioned into 16 splits. As shown in Fig. 4, the Twitter partitions under the DB policy have balanced RTI graph characteristics, so the SD of all normalized RTI graph characteristics is small. We also find that the SDs for the CB and LDG policies are significantly higher than for the other policies. The reason is that vertices are accumulated to very large partitions to reduce edge cuts in CB and LDG. For the M policy, although the SD of the normalized number of

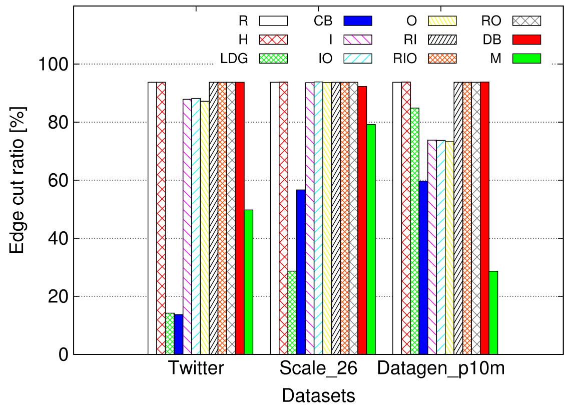

Fig. 6. The Edge Cut Ratio of all partitioning policies for 3 datasets.

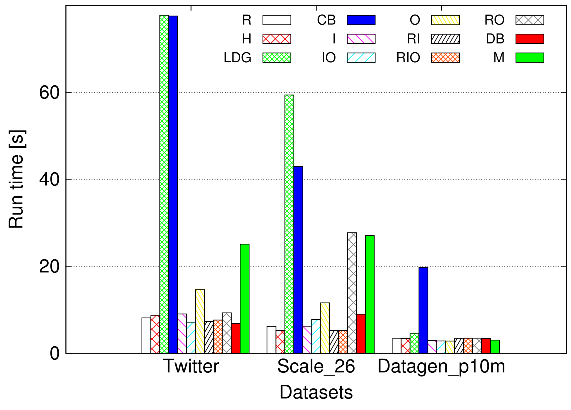

Fig. 8. The run time of PageRank for 3 datasets with all partitioning policies.

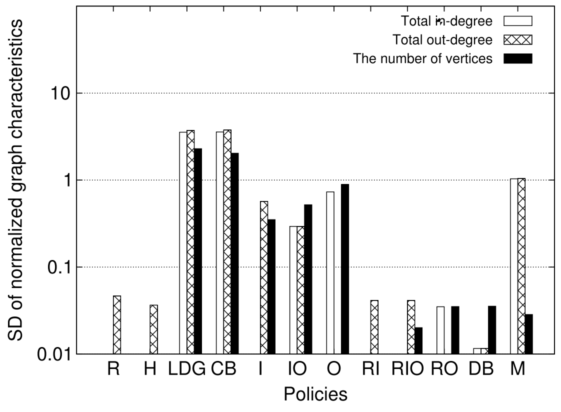

Fig. 7. The standard deviation of the normalized RTI graph characteristics for Twitter for all partitioning policies (the values of missing bars are too small to display).

vertices is small, the SDs of the normalized total in-degree and out-degree are relatively large, which indicates that communication is not balanced between pairs of working machines. Surprisingly, the random-based policies (R and H) also obtain small SD (we have repeated the R partitioning 5 times with different random seeds and obtained consistent results).

In Fig. 8 we show the run time of PageRank on all 3 datasets for all graph partitioning policies. The LDG, CB, and M policies result in the longest run times, even though they achieve a low ECR. The reason is that communication is not the dominant workload in PGX.D when using the high-speed InfiniBand, as we have discussed in Section 4.2. This means that ECR is not a good metric when the communication is not the dominant part of the workload. We find that in general, the partitioning policy with smaller SD of the RTI graph characteristics leads to shorter run time, and so SD can be used as a metric to evaluate the quality of partitioning for computation-dominated processing. Except for the CB, LDG, M, and O policies, the SD of the other policies is less than 0.5 and their run times are very close to each other. In practice, it is useful to find a threshold for SD beyond which the run time of graph processing may significantly increase. This threshold may be determined by analyzing the statistics obtained from many more experiments with various algorithms and datasets.

5.4. *The impact of the partitioning policies on application performance*

In this section we present the performance impact of the partitioning policies on the performance of graph algorithms for different algorithms, datasets, and number of working machines.

Key findings:

- The Degree-Balanced policy achieves good performance, while previous streaming policies from the literature (LDG and CB) perform the worst.
- The graph structure has an impact on the performance of graph partitioning.
- Most partitioning policies show reasonable scalability with the increase of the number of working machines (partitions).

The run time of PageRank for 3 datasets with all partitioning policies is depicted in Fig. 8. There is no overall winner among the partitioning policies, but LDG and CB have the worst performance as the computation workload of for these policies is highly skewed between working machines (see Fig. 7). DB achieves good performance for all graphs. For the Twitter graph, the run time of PageRank is the shortest. Random ordering cannot always help to achieve good performances evidenced by the O and RO policies for partitioning Scale_26. The impact of graph partitioning is more significant in highly skewed graphs, such as Twitter and Scale_26. For Datagen_p10m, we see that only CB has obvious performance impact. Both LDG and M yield results comparative to those other partitioning policies. Simple partitioning policies, such as the commonly used H policy, perform well for most algorithms and graphs. The reason is that computation is the dominant workload in our experiments and the H policy balances normalized RTI graph characteristics as shown in Fig. 7.

In Figs. 9–11 we show that most partitioning policies exhibit good scalability when increasing the number of worker machines up to 16—the benefit of increasing the number of machines from 16 to 32 is not significant. An important reason is that the workload is not heavy enough when processing the graphs with more than 16 machines (i.e., the hardware resource is redundant). For LDG and CB, the scalability is not obvious. To reduce edge-cuts, no matter how many number of partitions, LDG and CB may place vertices to a small subset of partitions, which dominates the run time of the algorithms. We also find that the random ordering results in poor scalability, such as the RO policy shown in Fig. 10.

5.5. *The impact of network and the selective ghost node technique*

In this section, we compare the performance impact of using 56 Gbit/s InfiniBand versus 1 Gbit/s Ethernet, and of using selective ghost node (SGN), which is a commonly used technique in graph-processing systems for reducing network traffic.

Key findings:

- The run time of graph-processing algorithms on high-speed InfiniBand is orders of magnitude smaller than on low-speed Ethernet.

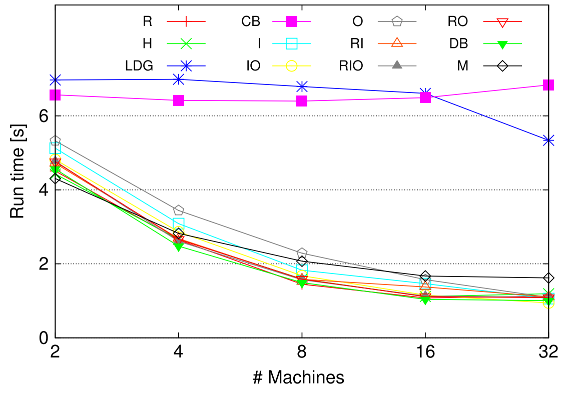

Fig. 9. The scalability of the BFS algorithm for Twitter.

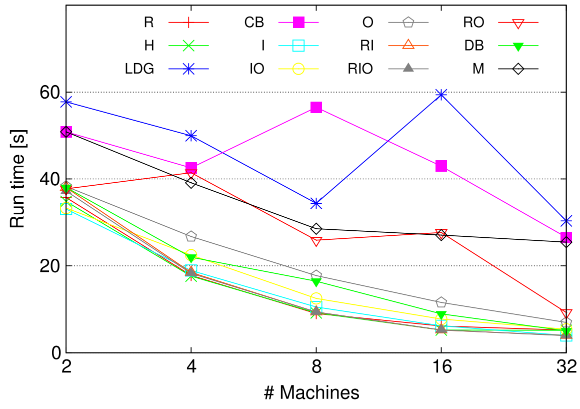

Fig. 10. The scalability of the PageRank algorithm for Scale_26.

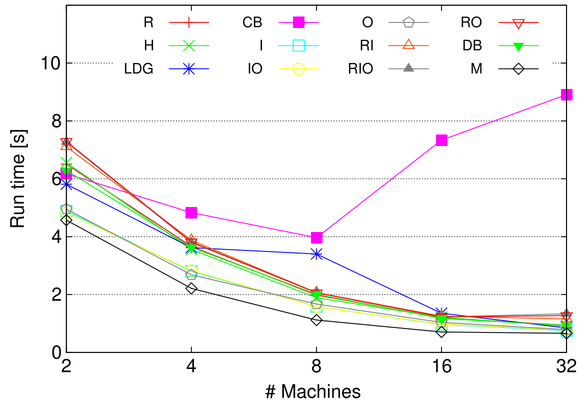

Fig. 11. The scalability of the WCC algorithm for Datagen_p10m.

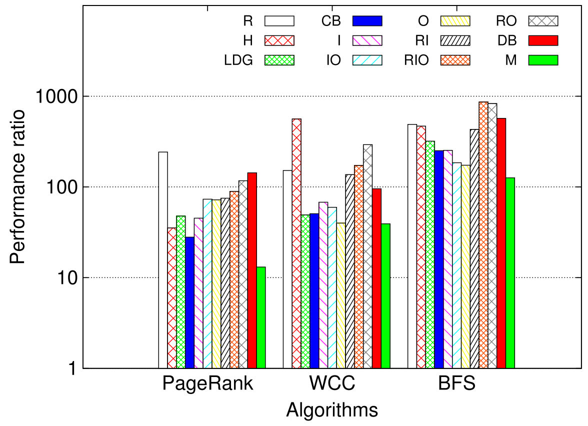

Fig. 12. The performance ratio of 3 algorithms for Twitter on InfiniBand relative to Ethernet (vertical axis has logarithmic scale).

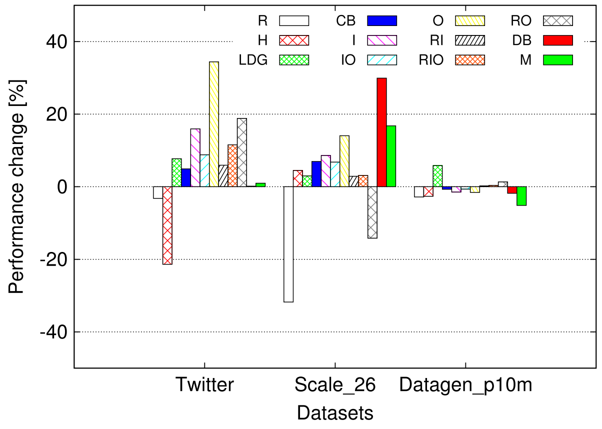

Fig. 13. The performance change of PageRank for 3 datasets when using SGN on InfiniBand.

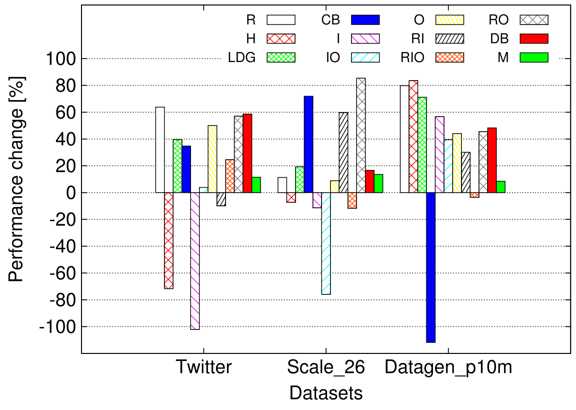

Fig. 14. The performance change of PageRank for 3 datasets when using SGN on Ethernet.

- Using the selective ghost node technique may not always have a positive impact on the performance.

We report the performance of InfiniBand relative to Network when running 3 algorithms with Twitter in Fig. 12. In all experiments, using InfiniBand leads to much better performance, from 10 times to nearly 900 times faster than the Ethernet. It is very interesting that the performance ratio can be as much as hundreds times, while the bandwidth of the InfiniBand is only about 50 times larger than that of the Ethernet. It may be because that the communication is not balanced between pairs of machines. For example, one machine may have heavy communication with multiple other machines. Other machines may have to wait for that machine to finish their communication, which makes the data transfer and message processing extremely slow.

We show the performance improvement for PageRank of 3 datasets by using SGN on InfiniBand and on Ethernet in Figs. 13 and 14, respectively. Not all values are positive, indicating that using SGN cannot always help to achieve good performance, because the time synchronizing ghost nodes can be longer than the run time reduced by using SGN. Overall, the performance change on Ethernet is larger than that on InfiniBand, because Ethernet is more sensitive to the change of network traffic.

5.6. The time spent on partitioning graphs

The complexity of the partitioning policies and the time spent on partitioning graphs are also important for us to determine the

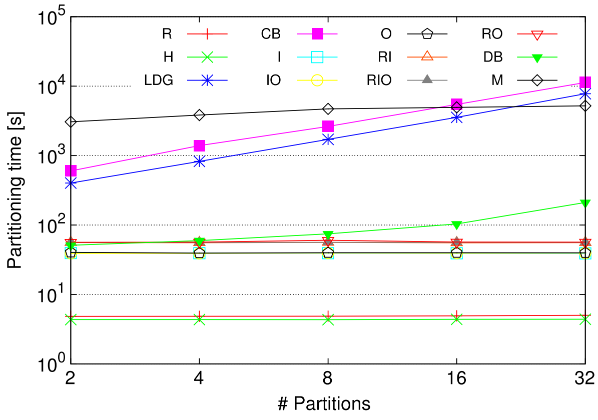

Fig. 15. The time spent on partitioning the Twitter graph into different numbers of partitions for all policies (vertical axis has logarithmic scale).

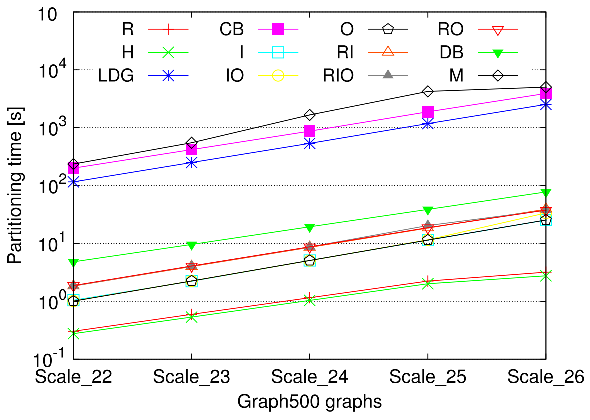

Fig. 16. The time spent on partitioning Graph500 graphs into 16 partitions for all policies (vertical axis has logarithmic scale).

choice of policies. Because the M policy is implemented in an offline single-machine partitioner, and the LDG and CB policies need to acquire the global information to assign vertices, it is non-trivial to implement these policies in a distributed manner. In this section, we compare the time spent on partitioning graphs on a single machine.

Key findings:

- The LDG, CB, and M policies need much more time for partitioning graphs than the other streaming policies.
- The number of partitions has a significant impact on the partitioning time of LDG and CB.
- The partitioning time of all policies increases linearly with the size of the graph.

We first explore the time spent on partitioning the same graph into different numbers of partitions. In Fig. 15, we show the time of each policy for partitioning Twitter into 2, 4, 8, 16, and 32 partitions, respectively. For the M policy, we use another machine (equipped with two Intel Xeon CPU E5-2699 2.30 GHz processors and 384 GB memory), because the M policy runs out of memory when using the working machine in Table 5. LDG, CB, and M are the policies with the longest partitioning time. The M policy applies a multi-level scheme, in which the coarsening phase is complex and time consuming. This long partitioning time of M matches a previous experiment [42], where more than 8.5 h is needed to partition the Twitter graph using a less powerful machine. For the assignment of a vertex, the LDG and CB policies need to traverse all partitions to calculate the number of its neighbors in each partition. To assign some low-degree vertices in CB, counting the edges between each pair of partitions is also required. The traversal of partitions is very expensive. With the increase of the number of partitions, the LDG and CB policies need to spend significantly more time on partitioning, because of the complexity of the traversal process. Except for LDG and CB, we observe time increase of DB, which is incurred by sorting the partition queue, the size of which is equal to the number of working machines. In practice, the size of clusters is limited, many of which have less than thousands of machines. Thus, the impact of increasing the number of partitions is limited for the DB policy.

We also investigate the partitioning time on different sizes of graphs. Fig. 16 shows the time spent on partitioning Graph500 graphs with 5 different scales (from Scale_22 to Scale_26). We partition each graph into 16 splits. Similarly to Twitter, we use the same machine with 384 GB memory only for executing the M policy with Scale_26, because out of memory. LDG, CB, and M are the slowest policies. All partitioning policies exhibit good scalability with increasing the size of graphs.

6. Discussion

In this section, we discuss how to use our results, how to extend the use of our model and method to more graph-processing systems, and the potential directions for the design of future graph-partitioning policies.

6.1. How to use our results

We summarize the key findings of our experiments in Table 10. Key findings in Sections 5.4 and 5.6 are about the performance of partitioning policies. It is difficult to obtain clear rules as to which partitioning policy should be used for which graph-processing system, which algorithm, and which graph. We identify four main reasons for this difficulty. First, graph-processing systems are designed and implemented with specific goals and optimization techniques. It is not easy to quantify the impact of these implementations and techniques on the performance of graph-partitioning policies. Second, graph algorithms have various behaviors. We will further discuss the impact of graph algorithms on partitioning policies in Section 6.3. Third, graphs have diverse structures and characteristics. It is very difficult to identify the typical graph structures and the most important graph characteristics that can represent a given graph [26]. In practice, the identified structures and characteristics should be easily calculated, which is crucial for large-scale graphs. Fourth, heterogeneous hardware infrastructure (different CPU, amount of memory, network connection, etc.) also has significant impact. For the same combination of graph-processing system, algorithm, and graph, if the deployed cluster is changed, the best partitioning policies may also change.

Although it is non-trivial to obtain best practice, we discover and summarize some generic suggestions for designing and using policies. Key findings in Sections 5.2 and 5.5 are closely related to the PGX.D system and its hardware infrastructure. They may not be applicable for other systems, but these findings indicate that system configuration and tuning should be carefully conducted (for different partitioning policies). Key findings in Section 5.3 are more generic and can be used by other researchers to design and measure the performance of their graph-partitioning policies. Our DB policy cannot always outperform other partitioning policies in all cases, but in general, it achieves good performance (short run time of graph algorithm and fast partitioning process), if any other graph system that falls in the same run time model of PGX.D, we would suggest to use the DB policy.

Table 10
Key findings of our experiments.

Section	Key findings
5.2	The configuration of worker and copier threads has a significant impact on the run time of PGX.D. In most experimental runs, the thread configuration w12c14 shows the best performance.
5.3	The ECR is not a good indicator for the quality of partitioning for real graph-processing systems. The SD of the RTI graph characteristics can be used to measure the imbalance of the computation workload. The design of partitioning policies should also focus on balancing the communication between machines.
5.4	The DB policy achieves good performance, while LDG and DB perform the poorest. The graph structure has an impact on the performance of graph partitioning. Most partitioning policies show reasonable scalability with the increase of the number of partitions.
5.5	The run time of algorithms on InfiniBand is orders of magnitude smaller than on Ethernet. Using the selective ghost node technique may not always have a positive impact on the performance.
5.6	The LDG, CB, and M policies need much more time for partitioning graphs than the other streaming policies. The number of partitions has a significant impact on the partitioning time of LDG and CB. The partitioning time of all policies increases linearly with the size of the graph.

6.2. *The coverage of our model and method*

In Section 3, we propose a run time model of two-phase graph processing systems, which also encompasses *one-phase* systems. In our experiments, we use PGX.D (a real-world production system based on the two-phase abstraction) as the real graph-processing system. Because we have tested our work on production-quality code, and because of the simplicity of the conversion between the one-phase abstraction and the two-phase abstraction [28], our work also indicates that our method could be applied with trivial adaptations to systems using the one-phase abstraction.

We now discuss the extensions needed to apply our work to systems based on the *three-phase* abstraction. A typical three-phase abstraction is the Gather–Apply–Scatter (GAS) model, which is first implemented in PowerGraph [11]. Vertex-cut partitioning is often implemented in GAS systems: a vertex can have multiple copies, each of which is distributed to a working machine. One copy is selected as the master, and others are mirrors. In GAS, the gather phase collects the local incoming information for vertices, then calculates their partial vertex values. The apply phase collects all partial values and computes final vertex values. Last, the scatter phase distributes the update to corresponding edges. There are two periods of communication in the GAS model, with one period between the gather and apply phases for sending partial vertex values to the master, and another between the apply and scatter phases for distributing final vertex values to all mirrors. We extend our run time model to GAS systems, for example, by observing that the run time becomes the sum of the time spend on each of the three computation phases and the two communication periods in the blocking I/O mode. Next, we can use our method to pick out run-time-influencing graph characteristics for vertex-cut partitioning, and proceed design new policies. (Using these steps, we have already completed a preliminary model for three-phase systems, but we do not report the outcome in this work, as we have not proceeded with the design of new policies and have not conducted meaningful experiments with them.)

6.3. *The design of future partitioning policies*

Policies when considering the heterogeneity of clusters. Graph-processing systems may be deployed on clusters with different hardware, such as machines with different processors and amount of memory, and different type and topology of networks. From our analysis and previous knowledge [15,26], both the computation and the communication processes are important to the run time of graph-processing systems. However, when considering heterogeneous clusters, the computation or the communication may become the dominant bottleneck of the system, and thus requiring more in-depth analysis. We may need to understand the relative priority of balancing computation and of minimizing communication to design graph partitioning policies.

Policies that balance communication. Minimizing communication is an important target of graph partitioning. However, it is only about the total amount of network traffic. We identify two important situations when partitioning can lead to lower network traffic yet incur a longer processing run time. The first situation occurs when most edge-cuts are made between a pair or a small subset of working machines, which means that the processing run time is determined by the communication between these machines. The second situation occurs when the speed of creating messages by working machines varies significantly over time. As we have learned from decades of parallel and distributed computing, message bursts can significantly reduce performance, and can even lead to system crashes. The balance of communication is a very important direction for graph partitioning and should consider both, inter-machine and intra-machine optimizations. The inter-machine optimization requires a balanced amount of messages between pairs of machines. The intra-machine optimization has to find a sequence of processing vertices that can distribute the creation of messages evenly.

Policies addressing algorithmic variety in real-world graph processing. Many graph algorithms are iterative and can be categorized by the status and count of active vertices in each iteration, into stationary and non-stationary [22]. In each iteration of stationary algorithms, all the vertices are active and they receive and generate the same amount of messages. Typical stationary algorithms are PageRank, and Semi-clustering [27]. In contrast, only a part of vertices are active in one iteration for non-stationary algorithms, such as BFS, Single Source Shortest Path [27], and WCC. It is challenging to predict and balance the workload of non-stationary algorithms in each iteration, because we do not know what are the active vertices and developing good predictors has so far proven difficult and algorithm-specific. Dynamic repartitioning may help solve this balancing problem. However, existing repartitioning approaches are unable to do so, because they repartition graphs based on information regarding the current iteration [22] or (in the few cases that have tried this approach so far) the previous iterations [37].

7. Conclusion

Graph partitioning is an important aspect of achieving high performance when designing and using distributed graph-processing systems. Many graph partitioning policies have been proposed so far, aiming to minimize communication, balance the number of vertices on each working machine, and reduce the time spent on partitioning, etc. However, most of the partitioning policies are not designed from the perspective of real-world distributed graph-processing systems. In addition, the performance of existing partitioning policies has not been evaluated in-depth on real systems. In this work, we address this situation by proposing models, partitioning policies, and an experimental evaluation of different partitioning policies in graph processing.

We model the run time of different types of graph-processing systems. We set minimizing the run time as the objective function of partitioning policies. The models we proposed cover the one-phase and two-phase systems, using the blocking I/O and parallel I/O modes, in machine-level and thread-level.

We propose a method to identify run-time-influencing graph characteristics by analyzing the run-time model and by understanding the relationship between different graph characteristics and the run time. Based on the run-time-influencing graph characteristics, we design new graph partitioning policies to obtain balanced partitions.

We use many metrics to evaluate the performance of twelve partitioning policies. We select in our experiments three popular graph-processing algorithms and three large-scale graphs from both real world and synthetic graph generators. We also evaluate the impact of real-world networks and a commonly used technique in graph-processing systems. Our results indicate that the newly-designed DB partitioning policy shows good performance, while existing streaming policies, such as LDG and CB, do not perform well.

We also discuss our preliminary work and ideas regarding how to use our results, the coverage of our model and method, and the design of future partitioning policies. In the future, we plan to implement a distributed graph-processing system that can use both the CPU and the GPU(s), and to design corresponding streaming graph-partitioning policies for this hybrid system.

Acknowledgments

This research is supported by the Veni project @large, by the Dutch Nederlandse Wettenschappelijke Onderzoek (the Dutch Research Agency) Kennis Innovatie Mapping (KIEM; Mapping between Knowledge and Innovation) project KIES-A, by Commit and the Commit project Commissioner, by the NSFC No. 61379146, No. 61272483, No. 61272056, and by the Fund No. JC13-06-03.

References

[1] K. Andreev, H. Racke, Balanced graph partitioning, Theory Comput. Syst. (2006).

[2] S. Arora, S. Rao, U. Vazirani, Expander flows, geometric embeddings and graph partitioning, J. ACM (2009).

[3] D.A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, D. Wagner, Benchmarking for graph clustering and partitioning, in: ESNAM, 2014.

[4] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, P. Boncz, Graphalytics: A big data benchmark for graph-processing platforms, in: GRADES, 2015.

[5] F. Checconi, F. Petrini, Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines, in: IPDPS, 2014.

[6] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, W. Pan, GraphHP: A hybrid platform for iterative graph processing. 2014.

[7] Family of Graph and Hypergraph Partitioning Software. http://glaros.dtc.umn.edu/gkhome/views/metis.

[8] Z. Fu, M. Personick, B. Thompson, MapGraph: A high level api for fast development of high performance graph analytics on GPUs, in: GRADES, 2014.

[9] Giraph. http://giraph.apache.org/.

[10] L. Golab, M. Hadjieleftheriou, H. Karloff, B. Saha, Distributed data placement to minimize communication costs via graph partitioning, in: SSDBM, 2014.

[11] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: Distributed graph-parallel computation on natural graphs, in: OSDI, 2012.

[12] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, GraphX: Graph processing in a distributed dataflow framework, in: OSDI, 2014.

[13] Graph500. http://www.graph500.org/.

[14] A. Guerrieri, A. Montresor, Distributed edge partitioning for graph processing, 2014. arXiv:1403.6270.

[15] Y. Guo, M. Biczak, A.L. Varbanescu, A. Iosup, C. Martella, T.L. Willke, How well do graph-processing platforms perform? An empirical performance evaluation and analysis, in: IPDPS, 2014.

[16] Y. Guo, A.L. Varbanescu, A. Iosup, D. Epema, An empirical performance evaluation of gpu-enabled graph-processing systems, in: CCGrid, 2015.

[17] Y. Guo, et al., How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report. Technical Report PDS-2013-004, Delft University of Technology, 2013, http://www.pds.ewi.tudelft.nl/research-publications/technical-reports/2013/.

[18] S. Hong, S. Depner, T. Manhardt, J.V.D. Lugt, M. Verstraaten, H. Chafi, PGX.D: A fast distributed graph processing engine and lessons from it, SuperComputing (2015).

[19] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, in: ICPP, 1995.

[20] G. Karypis, V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, J. Parallel Distrib. Comput. (1998).

[21] G. Karypis, K. Schloegel, V. Kumar, ParMETIS: Parallel graph partitioning and sparse matrix ordering library, 1997.

[22] 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: EuroSys, 2013.

[23] H. Kwak, C. Lee, H. Park, S. Moon, What is twitter, a social network or a news media? in: WWW, 2010.

[24] LDBC. http://ldbcouncil.org/.

[25] 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, in: VLDB, 2012.

[26] Y. Lu, J. Cheng, D. Yan, H. Wu, Large-scale distributed graph computing systems: An experimental evaluation, in: VLDB, 2014.

[27] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: SIGMOD, 2010.

[28] R.R. McCune, T. Weninger, G. Madey, Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing, Comput. Surv. (2015).

[29] H. Meyerhenke, P. Sanders, C. Schulz, Parallel graph partitioning for complex networks, 2014. arXiv:1404.4797.

[30] D.C. Montgomery, E.A. Peck, G.G. Vining, Introduction to Linear Regression Analysis, John Wiley & Sons, 2012.

[31] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank citation ranking: Bringing order to the web. 1999.

[32] U.N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. E (2007).

[33] A. Roy, I. Mihailovic, W. Zwaenepoel, X-Stream: Edge-centric graph processing using streaming partitions, in: SOSP, 2013.

[34] I. Safro, P. Sanders, C. Schulz, Advanced coarsening schemes for graph partitioning, ACM J. Exp. Algorithmics (2015).

[35] S. Salihoglu, J. Widom, GPS: A Graph Processing System. Technical Report, 2012.

[36] S. Seo, E.J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng, Hama: An efficient matrix computation with the MapReduce framework, in: CloudCom, 2010.

[37] Z. Shang, J.X. Yu, Catch the wind: Graph workload balancing on cloud, in: ICDE, 2013.

[38] SNAP. http://snap.stanford.edu/index.html.

[39] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: SIGKDD, 2012.

[40] D. Stauffer, A. Aharony, Introduction to Percolation Theory, CRC Press, 1994.

[41] P. Stutz, A. Bernstein, W. Cohen, Signal/collect: Graph algorithms for the (semantic) web, in: ISWC, 2010.

[42] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic, FENNEL: Streaming graph partitioning for massive scale graphs, in: WSDM, 2014.

[43] D. Warneke, O. Kao, Nephele: Efficient parallel data processing in the cloud, in: MTAGS, 2009.

[44] T. White, Hadoop: The Definitive Guide, O'Reilly Media, Inc., 2012.

[45] N. Xu, B. Cui, L.-n. Chen, Z. Huang, Y. Shao, Heterogeneous environment aware streaming graph partitioning, IEEE Trans. Knowl. Data Eng. (2015).

[46] Y. Zhang, Q. Gao, L. Gao, C. Wang, Accelerate large-scale iterative computation through asynchronous accumulative updates, in: ScienceCloud, 2012.

Yong Guo is a Ph.D. student in the Distributed Systems Group of Delft University of Technology. His research interests are in the area of distributed computing systems, large-scale graph processing, and online gaming. He has built the Game Trace Archive, which provides a virtual meeting space for the game community to exchange and use game traces.

Sungpack Hong is a principal member of technical staff for Oracle Labs. His research interests include parallel and distributed algorithms for large-scale graph analytics, domain-specific languages design and implementation, and system-on-chip architecture design and simulation.

Hassan Chafi is the senior research manager at Oracle Labs where he currently leads various projects. His research investigates high-performance, parallel, in-memory graph analytics and using domain specific languages to simplify parallel programming. Dr. Chafi received his Ph.D. from Stanford University.

Alexandru Iosup is currently an Associate Professor with the Distributed Systems Group at TU Delft. He has received in 2009 his Ph.D. in Computer Science from the Delft University of Technology (TU Delft), the Netherlands. He was a visiting scholar at U. Wisconsin-Madison, U. Innsbruck, and U. California-Berkeley in the summers of 2006, 2008, and 2010, respectively. In 2011, Dr. Iosup has received a Veni grant (the Dutch equivalent of the US NSF CAREER.) He is the author of over 50 refereed scientific publications and has received several awards and distinctions, including best paper awards at IEEE CCGrid 2010, Euro-Par 2009, and IEEE P2P 2006. He has co-founded the Grid Workloads Archive, and the Peer-to-Peer, the Game, and the Failure Trace Archives, all of which provide open access to workload and resource operation traces from large-scale distributed computing environments. His long-term research interests are in the area of distributed computing systems and their applications (keywords: cloud computing, grid computing, peer-to-peer systems, scientific computing, massively multiplayer online games, scheduling, scalability, reliability, performance evaluation, workload characterization).

Dick Epema received the M.Sc. and Ph.D. degrees in mathematics from Leiden University, Leiden, the Netherlands, in 1979 and 1983, respectively. Since 1984, he has been with the Department of Computer Science of Delft University of Technology, where he is currently a professor in the Distributed Systems Group. Since 2011, he is also a part-time full professor of Decentralized Distributed Systems at Eindhoven University of Technology. During 1987–1988, the fall of 1991, and the summer of 1998, he was a visiting scientist at the IBM T.J. Watson Research Center in New York. In the fall of 1992, he was a visiting professor at the Catholic University of Leuven, Belgium, and in the fall of 2009 he spent a sabbatical at UCSB. His research interests are in the areas of performance analysis, distributed systems, peer-to-peer systems, grids, and clouds. He has coauthored more than 100 papers in peer-reviewed conferences and journals, he was a general co-chair of Euro-Par 2009 and IEEE P2P 2010, and he was the general chair of HPDC'12 and CCGrid 2013.