Delft University of Technology
Parallel and Distributed Systems Report Series

Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies: A Technical Report

Yong Guo, Sungpack Hong, Hassan Chafi,
Alexandru Iosup, and Dick Epema
{Yong.Guo, A.Iosup, D.H.J.Epema}@tudelft.nl, {Sungpack.Hong, Hassan.Chafi}@oracle.com

Report number PDS-2015-002

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.ewi.tudelft.nl`

Information about Parallel and Distributed Systems Group:
`http://www.pds.ewi.tudelft.nl/`

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 of achieve good processing performance. To keep low the performance 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 the coverage of our model and method, and the design of future partitioning policies.

Contents

List of Figures

List of Tables

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 [1], GraphLab [2], and GraphX [3], 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 [4, 5, 6] 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 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 [7]. For graphs with billions of edges [8], 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 [5] 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 heuristic and rely on a limited set of theoretical metrics, such as the edge cut ratio [4], the number of vertices per partition [9, 10], etc. The partitions are created online by real-world graph-processing systems, which indicates 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 [11, 12, 13] 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 [5] 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 [14]. 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 [15] 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 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 [16], 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 [1, 2, 17]. 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 [1] and Hama [18], 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 neighbours that are not located in the same machine. Synchronization is needed between two consecutive iterations to ensure that all vertices has 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 are not balanced. To improve performance, graph-processing systems, such as GraphLab [2] and GraphHP [19], 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 [20] when executing graph algorithms.

Graph-processing systems can be categorized into three main *multi-phase* systems, based on their vertex computation abstractions [21]: *one-phase* [1, 17], *two-phase* [14, 22, 23], and *three-phase* [24, 25]. 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 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 Section 3 and Section 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

Table 1: Graph-processing systems using different partitioning approaches.

Partitioning approach	Example heuristics	Example systems using the approach
Traditional heuristics	METIS [15], ParMETIS [30]	-
Streaming	Hash, LDG [4]	Giraph [17], HeAPS [5]
Vertex-cut	Random, Balanced p-way [24]	PowerGraph [24], GraphX [3]
Dynamic	Exchange [9], Migration [10]	GPS [9], Mizan [10]
Chunking	File size	Hadoop [16], Stratosphere [32]

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 [21, 26], from scientific workflow scheduling [27] to recent work on large-scale graph processing [3]. 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 [7]. To achieve an approximate solution, many *traditional heuristics* [15, 28] have been proposed. Many of them adapt the *multi-level partitioning* scheme, which typically includes *three main phases* [28], 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 [15] and its family of partitioning policies [29], 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 [30], 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 [31].

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 [1, 17], 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 [24] 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 [4] 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 neighbours, while using the remaining capacity of partitions as a penalty. Tsourakakis et al. [6] 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. [5] 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 environment. 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 [4, 5].

The second direction is to partition graphs by *vertex-cut* [24, 3]. Vertex-cut partitioning place edges, instead of vertices, to different partitions. According to percolation theory [33], 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 [10] selects the execution time of each machines as the metric for workload imbalance and maintains a distributed hash table to record the position of vertices. GPS [9] 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. [34] 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 constrains 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 [4, 6, 35] are generally used to measure the quality of partitions. For real graph-processing systems, these metrics do not directly represent the performance of partitioning [5]. 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. [36] design their graph partitioning heuristic based on label propagation and size constrains for social networks and web graphs. Guerrieri and Montresor [37] 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 [4], FENNEL [6], and Xu et al. [5] 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 [2] and Giraph [17]. Their evaluations are also limited to the use of a single algorithm, PageRank; our own and related studies [11, 12, 13] 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 supersteps 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. [5], 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. [5], 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 k-th iteration of some application, and if T^k denotes the (total) run time of the k-th iteration across all machines, then we have:

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

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

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

We assume conservatively that in each iteration all vertices are active (that is, considered for processing) and that messages are sent to all their neighbours, for three reasons. First, many popular algorithms match well this assumption, such as community detection [38] and PageRank [39]. 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 6.2. 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 [21], 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

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, $l = 1, 2, ..., L$.
$\overline{T}_i^s$	time spent preparing outgoing messages across all threads.
$\overline{T}_{i,q}^s$	time spent preparing outgoing messages in the q-th thread, $q = 1, 2, ..., Q$.
$\overline{T}_i^x$	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

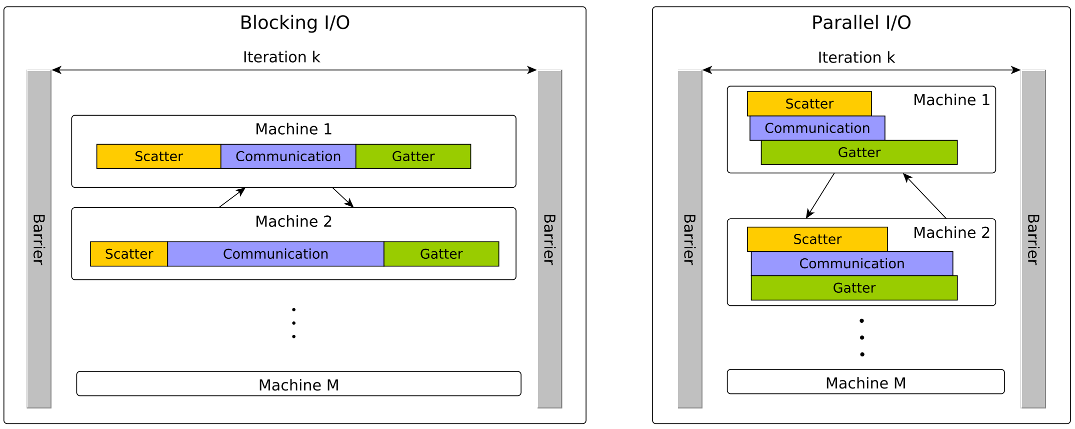

Figure 1: The computation phases and communication in one iteration of the Scatter-Gather abstraction.

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 [1, 17]. 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 [14]. Importantly, most one-phase systems can be converted to two-phase systems [21], 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 6.1.

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 Figure 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-

Table 3: $\overline{T}_i$ 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	$\overline{T}_i^g + \overline{T}_i^x + \overline{T}_i^s$	$max(\overline{T}_{i,l}^g) + \overline{T}_i^x + max(\overline{T}_{i,q}^s)$	$max(\overline{T}_i^g, \overline{T}_i^s, \overline{T}_i^x)$	$max(max(\overline{T}_{i,l}^g), max(\overline{T}_{i,q}^s), \overline{T}_i^x)$

phase abstraction uses a single computation function, all computation for a vertex is 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 main target of partitioning graphs for real graph-processing systems is to achieve the shortest run time. Similarly to Xu et al. [5], we set the objective function for finding a graph partitioning that minimizes the total runtime T_r:

$$\min\{T_r\} = N \times \min\{\max_i\{\overline{T}_i\}\} \tag{4}$$

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 A Class of Graph Partitioning Policies

In this section, we propose a method for identifying the most influential graph characteristics on the algorithm run time in a given graph-processing system (Section 4.1), we empirically validate our method on a real graph-processing system (Section 4.2), and design new streaming graph-partitioning policies (Section 4.3).

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

As many popular graph-processing systems [1, 17] can only process directed graphs, we consider without loss of generality, graph-processing systems which use a directed graph representation. Given a partition of a directed graph, we distinguish several characteristics which can determine the algorithm run time and which we summarize in Table 4. Our target is to identify the graph characteristics that have the strongest impact on the algorithm run time. We propose a three-step method to achieve this.

Step 1: Select the run time model for the graph-processing system. As we have shown in Section 3, the run time models are different for graph-processing systems. For a given graph-processing system, we want to identify the category it belongs to, in order to select the corresponding run time model from Table 3.

Step 2: Determine the PRTI graph characteristics. For the selected model, we determine which are the PRTI graph characteristics that may have an impact on each component of the model, based on the operation of the graph-processing algorithm. The PRTI graph characteristics represent a candidate set for the next step of our method.

Step 3: Identify the RTI graph characteristics. We take an experimental approach to identify the most representative graph characteristics in the PRTI set. To this end, we evaluate the relationship between the run time and different subsets of the PRTI set using the R-squared (R^2) coefficient obtained through linear regression [40]. For each subset we perform multiple experiments using different setups (e.g., policies, datasets, and configurations) and we build a histogram which shows how many times the R^2 value occurred in a given

Table 4: The graph characteristics of a partition

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, equals 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

range (see Table 7). We select the RTI graph characteristics as the subset of PRTI with the most occurrences in the highest range of R^2 values.

4.2 Empirical results validating the method

In this section, we empirically validate our method on the PGX.D graph-processing system. First, we use Table 3 to identify the corresponding run time model for PGX.D (*step 1* in our method). As PGX.D is a multi-threaded graph-processing system with two-phase abstraction and parallel I/O, its run time model is:

$$\overline{T}_i = \max(\max(\overline{T}^g_{i,l}), \max(\overline{T}^s_{i,q}), \overline{T}^x_i). \tag{5}$$

Next, we seek to understand the operation of PGX.D in order to select the PRTI characteristics (*step 2* in our method). 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. Thus, $\max(\overline{T}^g_{i,l})$ and $\max(\overline{T}^s_{i,q})$ 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) on each partition. If the sum of a vertex's in-degree and out-degree is greater than a pre-defined threshold, the vertex will be selected as a ghost. SGN is optional for users.

The PRTI graph characteristics may vary for different graph-processing algorithms and also for different components (e.g., computation and communication) within the same graph-processing algorithm. We apply the step 2 in our method on different components of the PageRank algorithm. The scatter phase in PageRank reads all vertices and prepares messages to remote neighbours through 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. In this case, 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.

We explain the experimental setup in terms of system configurations, datasets, and policies which we employ in order to identify the RTI characteristics (*step 3* in our method). In this section, we perform experiments with PageRank (maximum 10 iterations) on PGX.D deployed on a 16-machine cluster in Oracle Labs. The detailed

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

Table 6: Summary of datasets

Dataset	**V**	**E**	**Max D**	**Type & Source**
Twitter	41,652,230	1,468,365,182	3,081,112	Real-world, Public [42]
Scale_26	32,804,978	1,073,741,824	1,710,236	Synthetic, Graph500 [43]
Datagen_p10m	9,749,927	687,174,631	648	Synthetic, LDBC [44]

V and **E** are the vertex count and edge count of the graphs. **Max D** is the largest total degree of a vertex.

configuration of the cluster is summarized in Table 5. We set four configurations of the worker and copier threads: w24c2, w18c8, w12c14, and w6c20 [14]. The notation w24c2 means that for each working machine, we set 24 worker threads and 2 copier threads. We conduct experiments with or without using the selective ghost node technique.

Our experiments 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 the 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 similar structures and properties to the real-world social networks, such as Facebook. The LDBC generator is used by the Graphalytics project [41], 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 of 50,000 for Twitter and Scale_26, and 600 for Datagen_p10m.

We use three streaming graph-partitioning policies which are incorporated in PGX.D: the in-degree balanced policy (I), and the out-degree balanced policy (O), and the total degree balanced policy (IO). All policies assign vertices to different partitions by balancing the in-degree, the out-degree across partitions, or the total degree. To this end, each policy determines the average (in-/out-/total) degree ($\overline{D}_p$), which is the ration between the (in-/out-/total) degree of the entire graph and the number of partitions M. Further, vertices are assigned to partitions sequentially from partition 1 to partition M. When the (in-/out-/total) degree of partition $i < M$ exceeds the average (in-/out-/total) degree, we start to assign vertices to partition $i + 1$.

We obtain 72 executions of PageRank by using different setups in terms of system configurations, datasets, and policies. For each execution, we pick the working machines with the longest run times of the scatter and gather phases and we calculate for each machine its graph partition characteristics. We use different subsets of characteristics from the PRTI set, which we evaluate empirically in order to determine the RTI graph characteristics.

We find that the run time of PageRank is dominated by either the scatter phase or the gather phase. In

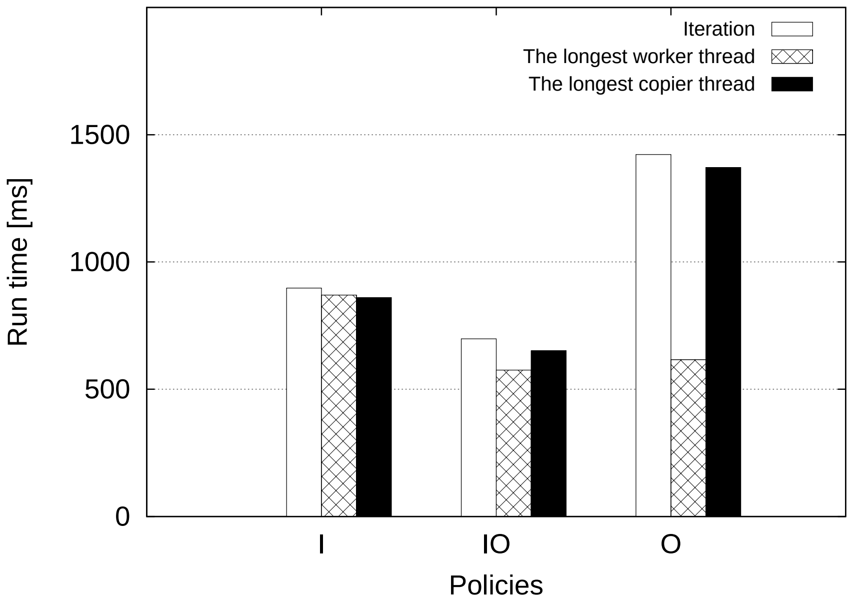

Figure 2: The run time of a thread, the longest worker thread, and the longest copier thread, using w12c14.

Figure 2, 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 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. The 50 ms difference represents the overhead of the system. We have similar findings for other system configurations. As PGX.D optimizes the network traffic and it is deployed on the high-speed InfiniBand network, the communication time in PageRank is overlapped by both the scatter and gather phases. In consequence, we focus on the run times of the scatter and gather phases.

We derive three subsets of characteristics from the PRTI set of each phase. Using the 72 experimental setups, we build the histogram of R^2 values for three subsets of characteristics in each phase. In Table 7 we show the histogram for the scatter phase and we identify the subset of characteristics with the strongest relationship to the algorithm run time as the number of vertices ($\#V$) and the total out-degree (D_{to}). Unlike previous policies which focus on the communication component by minimizing edge-cuts (e.g., D_{to}), our results show that the number of vertices is also an important factor. Thus, we consider the number of vertices ($\#V$) and total out-degree (D_{to}) as the RTI graph characteristics of the scatter phase. Similarly, for the gather phase, we identify the number of vertices ($\#V$) and remote in-degree (D_{ri}) as the RTI graph characteristics. We combine the results of both phases and we identify 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 also conduct the same set of experiments for the weakly connected component (WCC) algorithm, which computes the 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 algo-

Table 7: The number of runs in R^2 ranges for the scatter phase.

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)	6	4	6
[0, 0.6)	11	19	47

rithm. 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 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. Figure 3 shows that the graph characteristics are very well balanced for the Twitter partitions. For the Scale_26 and Datagen_p10m graphs, we achieve similar 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 [4], 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. Figure 6 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.

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 $D^j_{ti} \leq D^j_{to}$ then    ▷ $D^j_{ti}$ and $D^j_{to}$ is the current total in-degree and the current total out-degree of the $j$-th partition $P^j$.
5:              Assign the vertex to $P^j$, update $D^j_{ti}$ and $D^j_{to}$.
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 $D^1_{ti}$ and $D^1_{to}$.    ▷ 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 $D^j_{ti} \geq D^j_{to}$ of $P^j$ then
17:             Assign the vertex to $P^j$, update $D^j_{ti}$ and $D^j_{to}$.
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 $D^1_{ti}$ and $D^1_{to}$.
24:         $Index \leftarrow 1$
25:     end if
26: else    ▷ $V_i = V_o$
27:     Assign the vertex to $P^1$, update $D^1_{ti}$ and $D^1_{to}$.    ▷ Assign the vertex to the smallest/first partition.
28:     $Index \leftarrow 1$
29: end if
30: if $D^{Index}_{to} + D^{Index}_{ti} \geq C$ 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
```

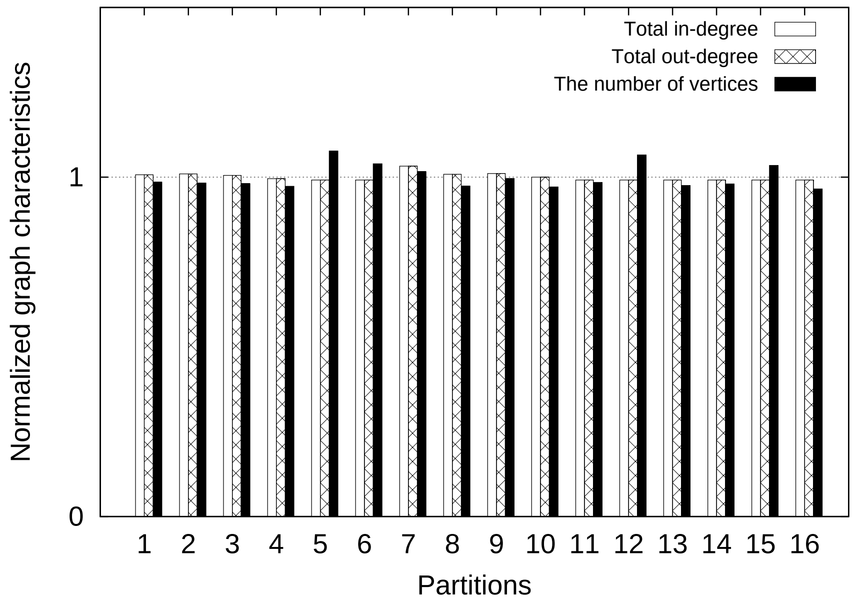

Figure 3: The normalized values of the graph characteristics achieved by the DB policy for Twitter.

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	-	-	-

5 Experimental Results

In this section we conduct comprehensive experiments with different graph partitioning policies, applictions, and system configurations. We first introduce the experimental setup as far as it has not been mentioned in Section 4.2. A summary of the experiments, and of the remaining sections, is in Table 8.

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 [45] (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, 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 select three algorithms based on our survey of graph-processing algorithms [46]: PageRank, Weakly Connected Components (WCC), and Breadth-First Search (BFS). PageRank and BFS propagate updates through out-edges. WCC propagates updates through both in- and out-edges, and does not need any

Table 9: Twelve partitioning policies in our experiments.

Policy	Streaming	Mechanism
R	Yes	Randomly assign a vertex to a partition.
H [1]	Yes	Hash partitioning.
LDG [4]	Yes	Assign a vertex to the partition, which has most neighbours of the vertex.
CB [5]	Yes	Assign a vertex to a partition with the smallest workload or with the least incremental workload.
I [14]	Yes	Balance the in-degree of partitions, original policy in PGX.D.
IO [14]	Yes	Balance the total-degree of partitions original policy in PGX.D.
O [14]	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 [15]	No	METIS, multi-level graph partitioning

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 [5], 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.

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 [14]. 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. Figure 4 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

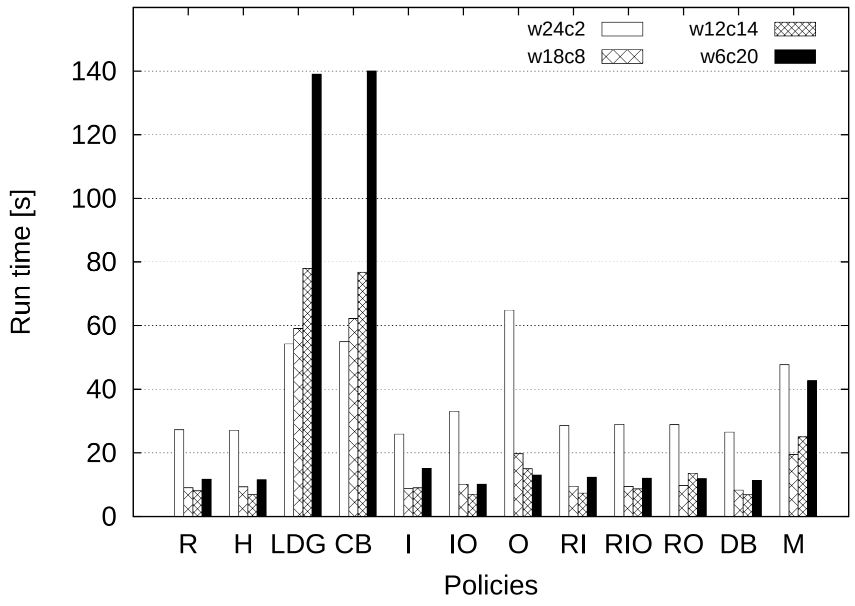

Figure 4: The run time of PageRank for Twitter with four thread configurations.

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 Figure 5. 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. Figure 6 shows the results for Twitter, which is partitioned into 16 splits. As shown in Figure 3, the Twitter partitions under the DB policy have balanced

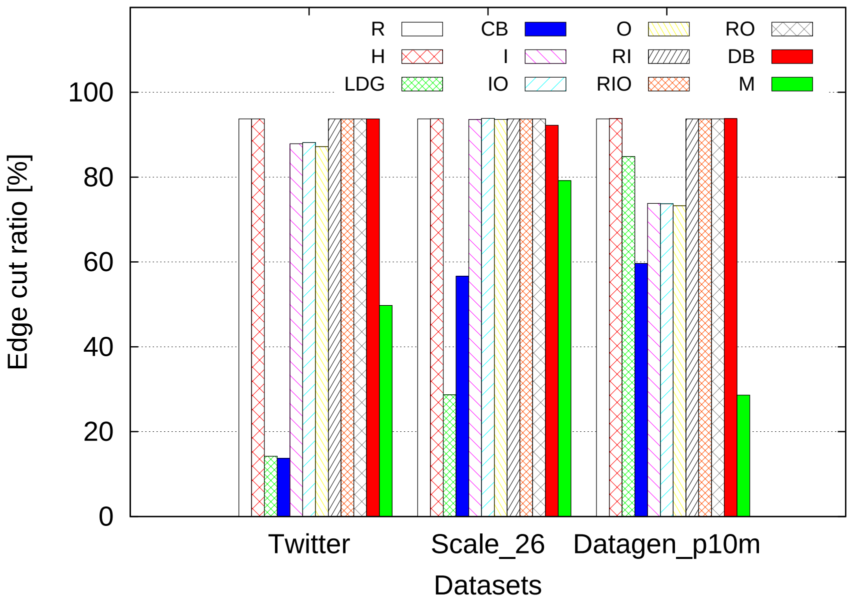

Figure 5: The Edge Cut Ratio of all partitioning policies for 3 datasets.

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 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 Figure 7 we show the runtime of PageRank on all 3 datasets for all graph partitioning policies. The LDG, CB, and M policies result in the longest runtimes, 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.

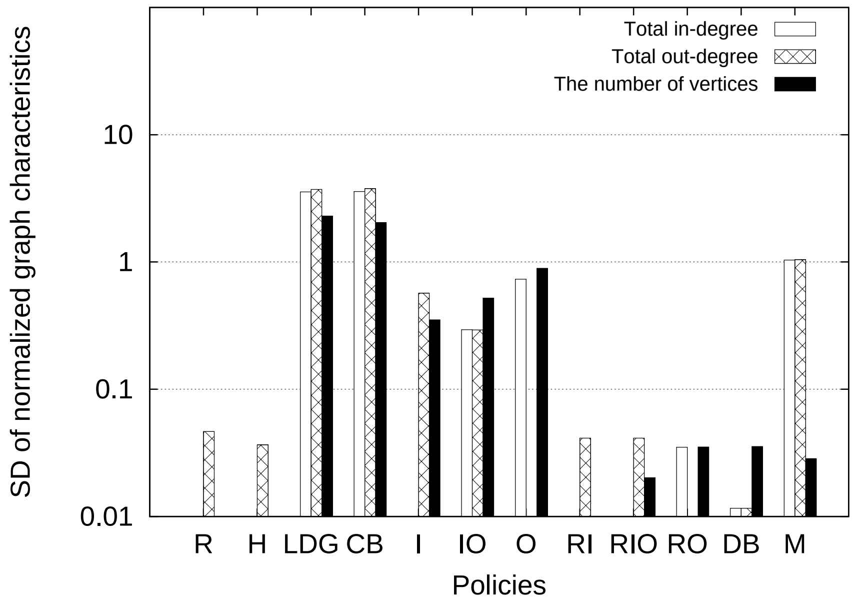

Figure 6: The standard deviation of the normalized RTI graph characteristics for Twitter for all partitioning policies (the values of missing bars are too small to display).

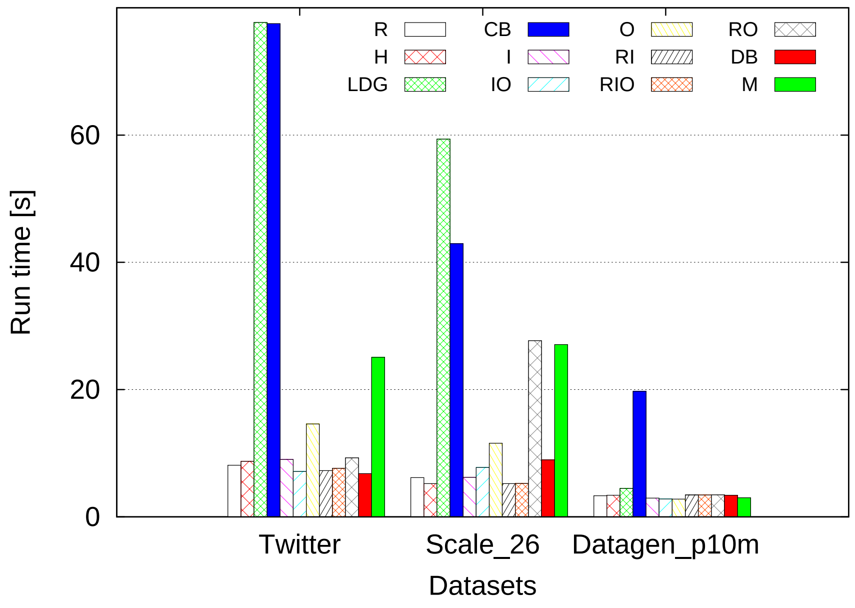

Figure 7: The run time of PageRank for 3 datasets with all partitioning policies.

- 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 Figure 7. There is no overall winner among the partitioning policies, but LDG and CB have the worst performance as the computation

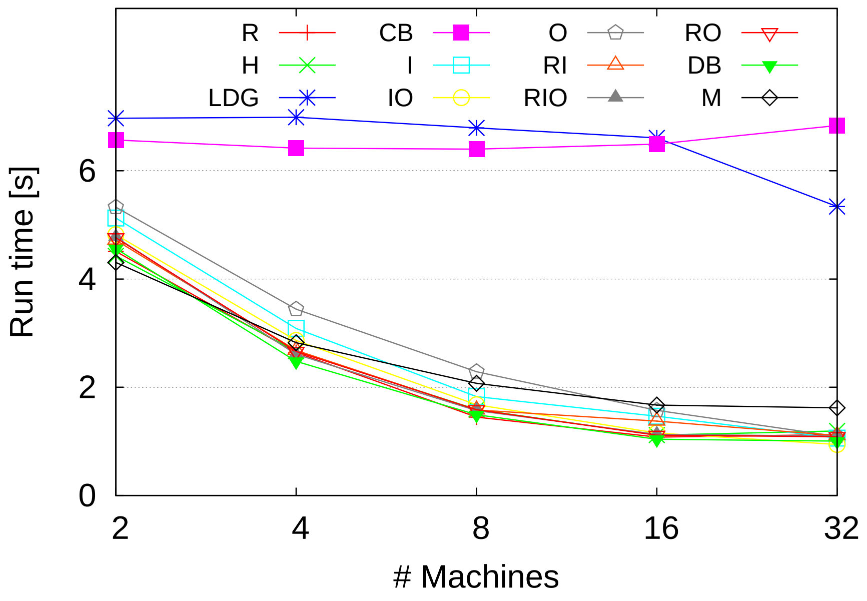

Figure 8: The scalability of the BFS algorithm for Twitter.

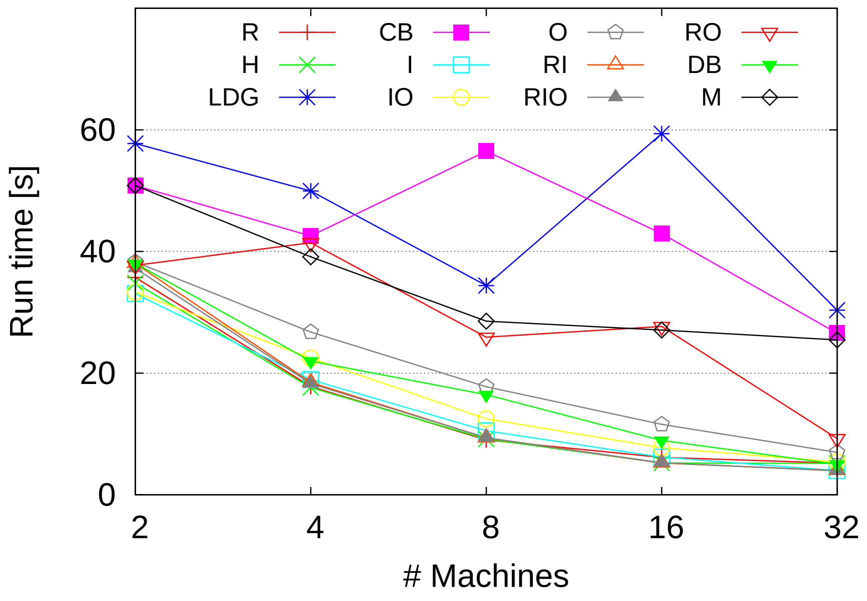

Figure 9: The scalability of the PageRank algorithm for Scale_26.

workload of for these policies is highly skewed between working machines (see Figure 6). 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

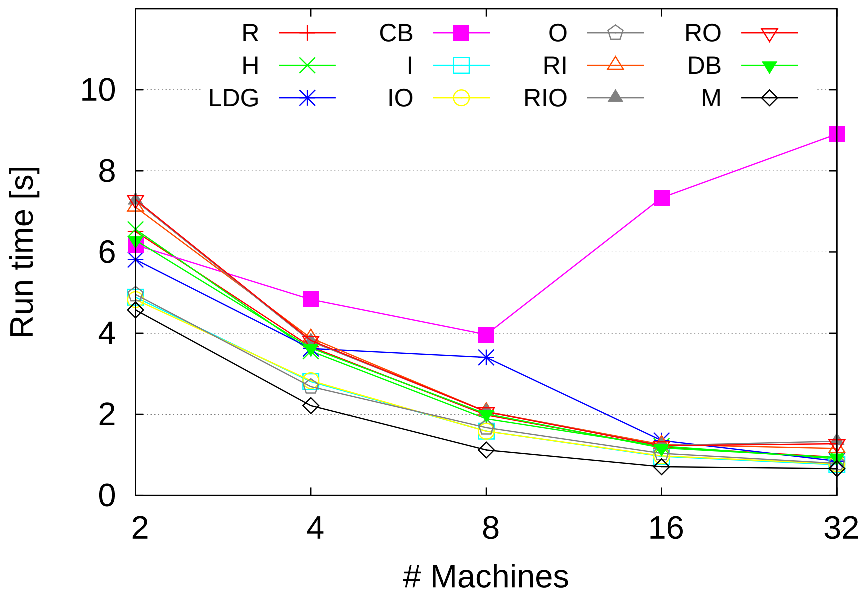

Figure 10: The scalability of the WCC algorithm for Datagen_p10m.

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 Figure 6.

In Figures 8, 9, and 10 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 Figure 9.

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.
- 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 Figure 11. 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 because that the communication is not balanced between pairs of machines. For example, one machine may

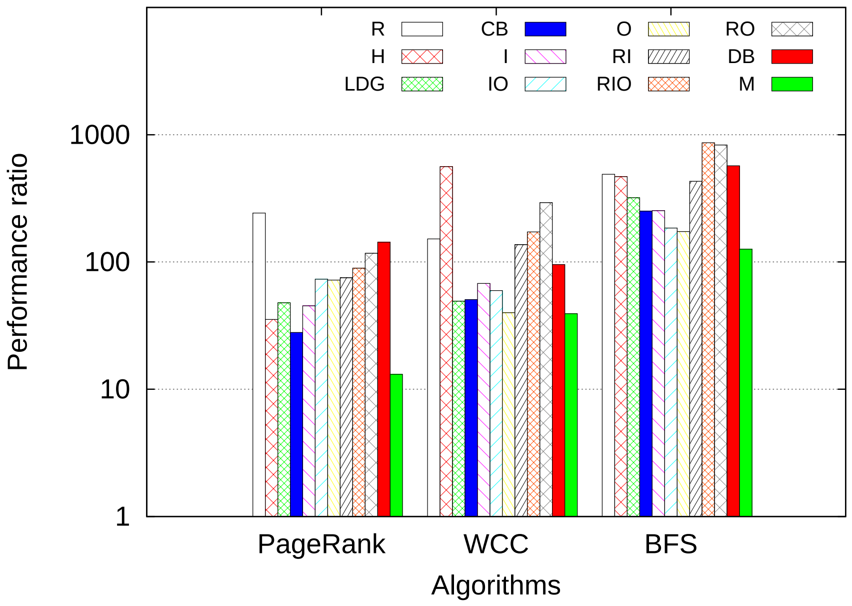

Figure 11: The performance ratio of 3 algorithms for Twitter on InfiniBand relative to Ethernet (vertical axis has logarithmic scale).

have heavy communication with multiple other machines. Other machines may have to wait 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 Figures 12 and 13, 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 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 Figure 14, 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,

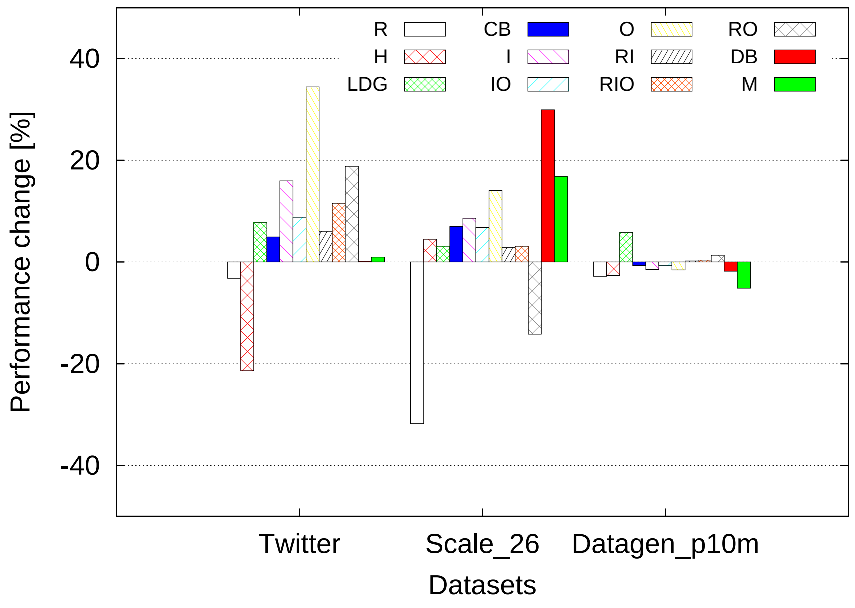

Figure 12: The performance change of PageRank for 3 datasets when using SGN on InfiniBand.

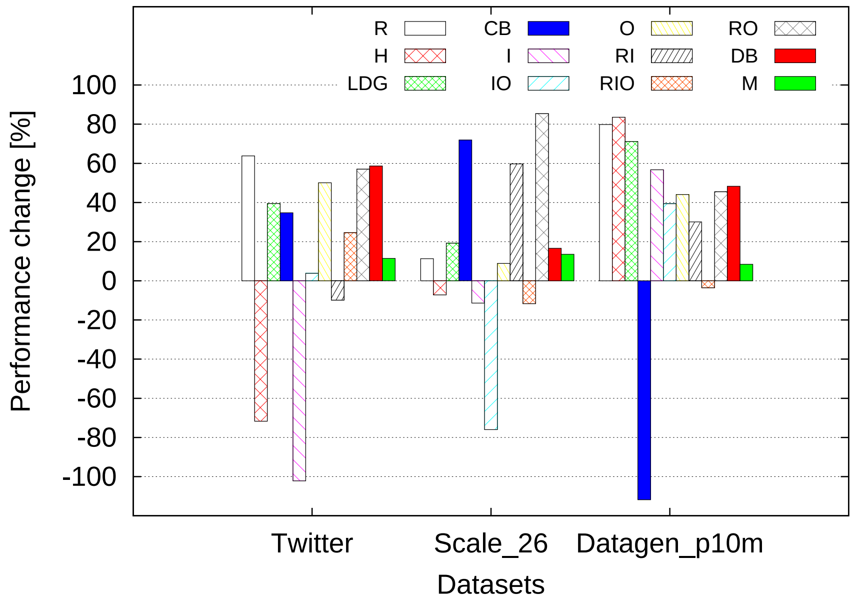

Figure 13: The performance change of PageRank for 3 datasets when using SGN on Ethernet.

in which the coarsening phase is complex and time consuming. This long partitioning time of M matches a previous experiment [6], where more than 8.5 hours 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

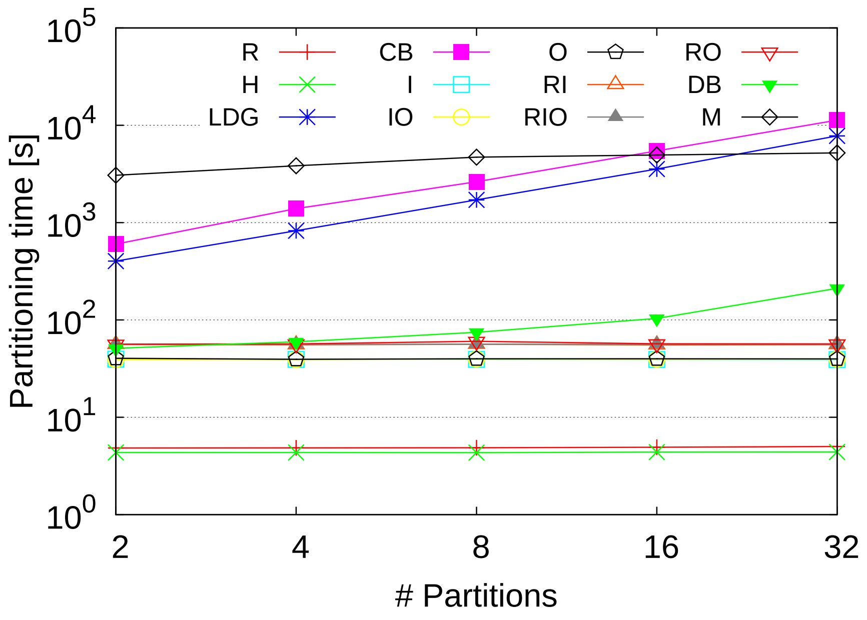

Figure 14: The time spent on partitioning the Twitter graph into different numbers of partitions for all policies (vertical axis has logarithmic scale).

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 are 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. Figure 15 shows the time spent on partitioning Graph500 graphs with 5 different scales (from Scale_22 to Scale_26). We partition each graph to 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 extend the use of our model and method to more graph-processing systems and what are the potential directions for the design of future graph-partitioning policies.

6.1 The coverage of our model and method

We discuss in this section the steps to be taken in the future for applying our model and method to other types of systems. We specifically consider the other two types of multi-phase graph-processing systems, and the emerging class of accelerator-based graph-processing systems.

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

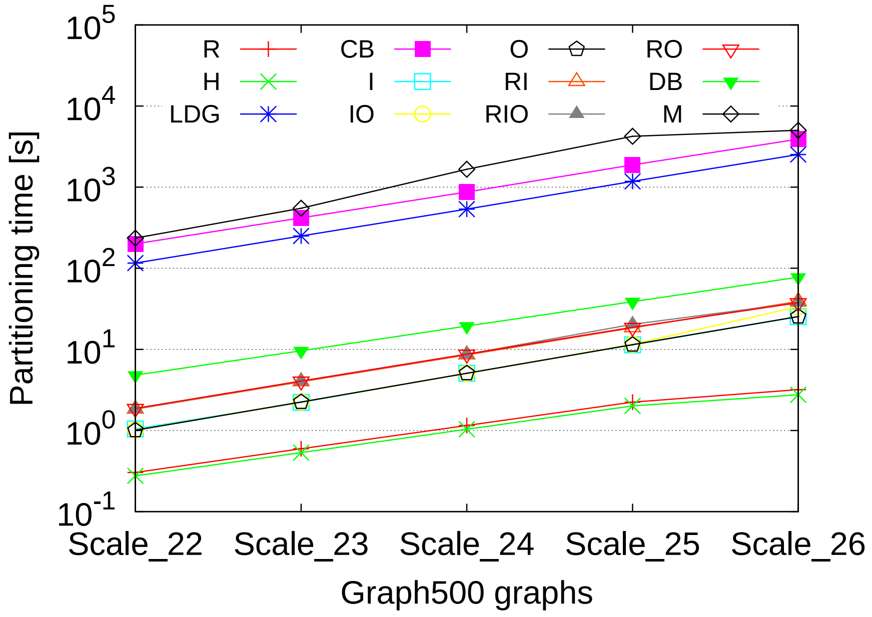

Figure 15: The time spent on partitioning Graph500 graphs into 16 partitions for all policies (vertical axis has logarithmic scale).

abstraction [21], 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 [24]. 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 distribute 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 have not proceeded with the design of new policies and have not conducted meaningful experiments with them.)

An emerging branch of graph-processing systems focus on the use of accelerators (such as GPUs). Currently, the most well-known GPU-enabled graph-processing systems are single-machine [25, 47, 48]. In the future, distributed graph-processing systems using hybrid CPU and GPU(s) as computing resources may be designed and made publicly available. Our method combining modeling and design should be applicable for accelerator-based systems. To this end, we would first model the run time of the systems, including components, such as the computation time of the CPU and the GPU(s), the communication time of inter- and intra- working machines, etc. Then, similarly to what we did in this work, we would need to identify how graph characteristics relate to performance, and design corresponding partitioning policies for GPU-enabled systems.

6.2 The design of future partitioning policies

Policies when considering the heterogeneity of clusters. A well-designed and fast graph-processing system may be selected and used by many graph analysts. The graph-processing system will 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 [11, 12], 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. For example, by using high-speed networks, such as InfiniBand, the stress from communication can be reduced (see Section 4). Thus, the requirement of minimizing communication could become less significant, and new policies and system designs on balancing the computation would lead to performance improvements. In contrast, we may need to give high priority to optimizing the communication for systems with relatively fast processors and large memory, or with relatively low-speed networks. The practical techniques used by graph-processing systems (for example, message aggregation, and other traditional techniques derived from telecommunications and parallel computing) can also affect the relative priority of balancing computation and of minimizing communication. For heterogeneous clusters with multiple types of machines and uneven network, it may be necessary to consider modeling the capability of the entire hardware infrastructure [5].

Policies that balance communication. Minimizing communication is an important target of graph partitioning. Metrics, such as the edge cut ratio, have been proposed to depict the amount of communication. However, minimizing communication 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. We envision that for graph partitioning the design of policies that can balance the communication between machines, and can still lead to acceptable communication volume, is a very interesting and important direction for future research. The balance of communication 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-processing algorithms consist of a number of iterations, but a few algorithms can still be completed in a single iteration (for example, local clustering coefficient). The iterative algorithms can be further categorized, by the status and count of active vertices in each iteration, into stationary and non-stationary [10]. 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 [1]. In contrast, only a part of vertices are active in one iteration for non-stationary algorithms. The amount of messages received and generated are various in different iterations. Non-stationary algorithms can be further divided into Traversal-Style and Multi-Phase-Style [34], according to whether a vertex can be re-activated. Traversal-Style algorithms access each vertex only once, for example, BFS, Single Source Shortest Path [1], and Random Walk [49]. Multi-Phase-Style algorithms can re-activate vertex to update information, for example, WCC, Maximal Matching [50], and Minimum Spanning Tree [51].

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 challenging 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 [10] or (in the few cases that have tried this approach so far) the previous iterations [34].

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 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.

References

[1] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, pages 135–146, 2010. 4, 5, 6, 9, 10, 17, 27

[2] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *VLDB*, pages 716–727, 2012. 4, 5, 8

[3] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*, 2014. 4, 6, 7

[4] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012. 4, 6, 7, 8, 14, 17

[5] Ningsheng Xu, Bin Cui, Luo-nan Chen, Zi Huang, and Yu Shao. Heterogeneous environment aware streaming graph partitioning. *TKDE*, 2015. 4, 6, 7, 8, 10, 17, 27

[6] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 2014. 4, 7, 8, 24

[7] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006. 4, 6

[8] Fabio Checconi and Fabrizio Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 425–434. IEEE, 2014. 4

[9] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. Technical report, 2012. 4, 6, 7

[10] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a System for Dynamic Load Balancing in Large-Scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013. 4, 6, 7, 27

[11] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IPDPS*, 2014. 4, 8, 27

[12] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *VLDB*, 2014. 4, 8, 27

[13] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, and Dick Epema. An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems. In *CCGrid*, 2015. 4, 8

[14] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine And Lessons From It. In *SuperComputing*, 2015. 5, 9, 12, 17

[15] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP*, pages 113–122, 1995. 5, 6, 17

[16] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012. 5, 6

[17] Giraph. http://giraph.apache.org/. 5, 6, 8, 9, 10

[18] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom)*, pages 721–726. IEEE, 2010. 5

[19] Qun Chen, Song Bai, Zhanhuai Li, Zhiying Gou, Bo Suo, and Wei Pan. Graphhp: A hybrid platform for iterative graph processing. 2014. 5

[20] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 13–22. ACM, 2012. 5

[21] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *Computing Survey*, 2015. 5, 6, 8, 9, 26

[22] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013. 5

[23] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, pages 764–780. Springer, 2010. 5

[24] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012. 5, 6, 7, 26

[25] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *GRADES*, 2014. 5, 26

[26] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998. 6

[27] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. Distributed data placement to minimize communication costs via graph partitioning. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, page 20. ACM, 2014. 6

[28] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *Journal of Experimental Algorithmics (JEA)*, 2015. 6

[29] Family of Graph and Hypergraph Partitioning Software. http://glaros.dtc.umn.edu/gkhome/views/metis. 6

[30] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. 1997. 6

[31] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM (JACM)*, 2009. 6

[32] Daniel Warneke and Odej Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *MTAGS*, 2009. 6

[33] Dietrich Stauffer and Ammon Aharony. *Introduction to percolation theory*. CRC press, 1994. 7

[34] Zechao Shang and Jeffrey Xu Yu. Catch the wind: Graph workload balancing on cloud. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 553–564. IEEE, 2013. 7, 27

[35] David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014. 7

[36] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *arXiv preprint arXiv:1404.4797*, 2014. 7

[37] Alessio Guerrieri and Alberto Montresor. Distributed edge partitioning for graph processing. *arXiv preprint arXiv:1403.6270*, 2014. 7

[38] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 2007. 8

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999. 8

[40] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*, volume 821. John Wiley & Sons, 2012. 10

[41] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. 2015. 12

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010. 12

[43] Graph500. http://www.graph500.org/. 12

[44] LDBC. http://ldbcouncil.org/. 12

[45] SNAP. http://snap.stanford.edu/index.html. 16

[46] Yong 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/. 16

[47] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *TPDS*, 2013. 26

[48] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrao Costa, and Matei Ripeanu. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. *TOPC*, 2013. 26

[49] Frank Spitzer and Austria Mathematician. *Principles of Random Walk*. Springer, 1964. 27

[50] Thomas E Anderson, Susan S Owicki, James B Saxe, and Charles P Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Transactions on Computer Systems (TOCS)*, 11(4):319–352, 1993. 27

[51] David Peleg. Distributed Computing: a Locality-Sensitive Approach. In *SIAM*, 2000. 27