How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis

Yong Guo*, Marcin Biczak*, Ana Lucia Varbanescu†, Alexandru Iosup*, Claudio Martella‡ and Theodore L. Willke§
*TU Delft, The Netherlands, Email: {Yong.Guo, M.Biczak, A.Iosup}@tudelft.nl
†University of Amsterdam, The Netherlands, Email: A.L.Varbanescu@uva.nl
‡VU University Amsterdam, The Netherlands, Email: claudio.martella@vu.nl
§Systems Architecture Lab, Intel Corporation, USA, Email: theodore.l.willke@intel.com

Abstract—**Graph-processing platforms are increasingly used in a variety of domains. Although both industry and academia are developing and tuning graph-processing algorithms and platforms, the performance of graph-processing platforms has never been explored or compared in-depth. Thus, users face the daunting challenge of selecting an appropriate platform for their specific application. To alleviate this challenge, we propose an empirical method for benchmarking graph-processing platforms. We define a comprehensive process, and a selection of representative metrics, datasets, and algorithmic classes. We implement a benchmarking suite of five classes of algorithms and seven diverse graphs. Our suite reports on basic (user-lever) performance, resource utilization, scalability, and various overhead. We use our benchmarking suite to analyze and compare six platforms. We gain valuable insights for each platform and present the first comprehensive *comparison* of graph-processing platforms.**

I. Introduction

Large-scale graphs are increasingly used in a variety of revenue-generating applications, such as social applications, online retail, business intelligence and logistics, and bioinformatics [1]–[3]. By analyzing the graph structure and characteristics, analysts are able to predict the behavior of the customer, and tune and develop new applications and services. However, the diversity of the available graphs, of the processing algorithms, and of the graph-processing platforms currently available to analysts makes the selection of a platform an important challenge. Although performance studies of individual platforms exist [4], [5], they have been so far restricted in scope and size. In this work, we propose a step forward: a comprehensive experimental method for analyzing and comparing graph-processing platforms. We further implement this method as a benchmarking suite and we apply it on six popular platforms. Our initial target is to provide benchmarking functionality for Small and Medium Enterprises (SMEs), who have access to clusters of a few tens of machines.

For both system developers and graph analysts (system users), a thorough understanding of the performance of these *platforms* (which we define as the combined hardware, software, and programming system that is being used to complete a graph processing task), under different input graphs and for different algorithms, is important—it enables informed choices, system and application tuning, and best-practices sharing. However, the execution time, the resource consumption, and other performance and non-functional characteristics of graph-processing systems depend to a large extent on the dataset, the algorithm, and the graph-processing platform. Thus, gaining a thorough understanding of graph-processing performance is impeded by three dimensions of diversity.

Dataset diversity: We are witnessing a significant increase in the availability and collectability of datasets represented as graphs, from road to social networks, and from bioinformatics material to citation databases.

Algorithm diversity: A large number of graph algorithms have been implemented to mine graphs for calculating basic graph metrics [6], for partitioning graphs [7]–[9], for traversing graphs [10]–[12], for detecting communities [13], [14], for searching for important vertices [15], [16], for sampling graphs [17], [18], for predicting graph evolution [1], etc.

Platform diversity: Many types of platforms are being used for different communities of developers and analysts. Addressing a variety of functional and non-functional requirements, a large number of processing platforms are becoming available. Neo4j [19], HyperGraphDB [20], and GraphChi [21] are examples of efficient single-node platforms with limited scalability. To scale-up, distributed systems with more computing and memory resources are used to process large-scale graphs, but they can even be less efficient than single-node platforms. For example, generic data processing systems such as Hadoop [22], YARN [23], Dryad [24], Stratosphere [25], and HaLoop [26] can scale out on multiple nodes, but may exhibit low performance due to distribution and overheads. Graph-specific platforms such as Pregel [5], Giraph [27], PEGASUS [28], GraphLab [29], and Trinity [30] are designed to provide feasible alternatives, but are not yet thoroughly evaluated for non-trivial algorithms and large datasets.

New performance evaluation and benchmarking suites are needed to respond to the three sources of diversity, that is, to provide comparative information about different performance metrics of different platforms, through the use of empirical methods and processes. However, the state-of-the-art in comparative graph-processing platform evaluation relies nowadays on Graph500 [31], the de-facto standard for comparing the performance of the hardware infrastructure related to graph processing. By choosing BFS as the single representative application and a single class of synthetic datasets, Graph500 has triggered a race in which winners use heavily optimized, low-level, hardware-specific code [32], which is rarely found or reproduced by common graph processing deployments and thus rarely reaches SMEs. Moreover, even the few existing platform-centric comparative studies are usually performed to prove the superiority of a given system over its direct competitors, so they only address a limited set of metrics and do not provide sufficient detail regarding the causes that lead to performance gaps.

Addressing the lack of a comprehensive evaluation method and set of results for graph processing platforms, this work addresses a key research question: *How well do graph processing platforms perform?* To answer this question, we propose an empirical performance-evaluation method for (large-scale) graph-processing platforms. Our method relies on defining a comprehensive evaluation process, and on selecting representative datasets, algorithms, and metrics for evaluating important aspects of graph-processing platforms—execution time, resource utilization, vertical and horizontal scalability, and overhead. Using this method, we create the equivalent of a benchmarking suite by selecting and implementing five algorithms and seven datasets from different application domains.

We implement this benchmarking suite on six popular platforms currently used for graph processing—Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j—and conduct a comprehensive performance study. This demonstrates that our benchmarking suite can be applied for many existing platforms, and also provides a first and detailed performance comparison of the six selected platforms. Our approach exceeds previous performance evaluation and benchmarking studies in both breadth and depth: we implement and measure multiple algorithms, use different types of datasets, and provide a detailed analysis of the results. Scale-wise, our study aligns well with SMEs cluster sizes and matches state-of-the-art studies (Section VI). Our work also aligns with the goals and ongoing activity of the SPEC Research Group and its Cloud Working Group[1], of which some of the authors are members.

Our main contributions are:

1) We propose a method for the comprehensive evaluation of graph processing platforms (Section II), which defines an evaluation process, and addresses multiple performance aspects such as raw performance, scalability, and resource utilization. When selecting the algorithms and datasets, the proposed method is equivalent to a benchmarking suite for graph-processing platforms; we select in this work five algorithms and seven datasets. We discuss the limitations of the coverage and representativeness of our comprehensive evaluation (Section V).
2) We demonstrate how our benchmarking suite can be implemented for six different graph processing platforms (Section III).
3) We provide a first performance comparison of six graph-processing platforms, quantifying their strong points and identifying their limitations (Sections IV).

[1]http://research.spec.org/working-groups/rg-cloud-working-group.html

II. Method and Benchmarking Suite for Graph-Processing Platforms

In this section we present an empirical method for evaluating the performance of graph-processing platforms. Our method includes four stages: identifying the performance aspects and metrics of interest; defining and selecting representative datasets and algorithms; implementing, configuring, and executing the tests; and analyzing the results.

A. Performance Aspects, Metrics, Process

To be able to reason about performance behavior, we first need to identify the performance requirements of graph-processing platforms, the system parameters to be monitored, the metrics that can be used to characterize platform performance, and an overall process how performance is evaluated.

In this study, we focus on four performance aspects:

1) *Raw processing power*: the ability of a platform to (quickly) process large-scale graphs. Ideally, platforms should combine deep analysis and short job runtimes. For SMEs, the latter could mean several minutes.
2) *Resource utilization*: the ability of a platform to efficiently utilize the resources it has. SMEs, who cannot afford inefficient use of their scarce resources, want platforms to waste as little compute and memory resources as possible.
3) *Scalability*: the ability of a platform to maintain its performance behavior when resources are added to its infrastructure. In our method, we test the strong scalability of platforms in both horizontal scale (by adding computing nodes distributedly) and vertical scale (by adding computing cores per node, thus ignoring network effects). Ideally, we want platforms to be able to automatically improve their performance linearly with the amount of added resources, but in practice this gain (or loss) depends both on the number and type of these resources, and on the algorithm and dataset.
4) *Overhead*: the part of wall-clock time the platform does not spend on true data processing. The overhead includes reading and partitioning the data, setting up the processing nodes, and eventually cleaning up after the results have been obtained. Ideally, the overhead should be constant and small relative to the overall processing time, but in practice the overhead may be related to algorithms and datasets.

Although we already use a more comprehensive set of workloads and metrics than the state-of-the-art (Section VI), there are still numerous limitations to our method, which we discuss in Section V.

The performance aspects can be observed by monitoring traditional system parameters (e.g., the important moments in the lifetime of each processing job, the CPU and network load, the OS memory consumption, and the disk I/O) and be quantified through various performance metrics. We summarize in Table I the performance metrics used in this work; our technical report [33] defines them more thoroughly.

TABLE I
SUMMARY OF PERFORMANCE METRICS.

Metric	How measured?	Derived	Relevant aspect (use)
job execution time (T)	Time the full execution	-	Raw processing power (Figure 1, 3, 4)
Edges per second (EPS)	-	$\#E/T$	Raw processing power (Figure 2)
Vertices per second (VPS)	-	$\#V/T$	Raw processing power (Figure 2)
CPU, memory, network	Monitoring sampled each second	-	Resource utilization (Technical report [33])
Horizontal scalability	T of different cluster size (N)	-	Scalability (Figure 5)
Vertical scalability	T of different cores per node	-	Scalability (Figure 6)
Normalized edges per second (NEPS)	-	$\#E/T/N$	Scalability (Figure 5, 6)
Computation time (T_c)	Time actual for calculating	-	Raw processing power (Figure 7)
Overhead time (T_o)	-	$T - T_c$	Overhead (Figure 7)

$\#V$ and $\#E$ are the number of vertices and the number of edges of graphs, respectively.

TABLE II
SUMMARY OF DATASETS.

	Graphs	#V	#E	d	$\bar{\mathbf{D}}$	Directivity
G1	Amazon	262,111	1,234,877	1.8	5	directed
G2	WikiTalk	2,388,953	5,018,445	0.1	2	directed
G3	KGS	293,290	16,558,839	38.5	113	undirected
G4	Citation	3,764,117	16,511,742	0.1	4	directed
G5	DotaLeague	61,171	50,870,316	2,719.0	1,663	undirected
G6	Synth	2,394,536	64,152,015	2.2	54	undirected
G7	Friendster	65,608,366	1,806,067,135	0.1	55	undirected

d is the link density of the graphs ($\times 10^{-5}$). $\bar{\mathbf{D}}$ is the average vertex degree of undirected graphs and the average vertex in-degree (or average vertex out-degree) of directed graphs.

TABLE III
SUMMARY OF ALGORITHMS.

	Algorithm	Main features	Use
A1	STATS	single step, low processing	decision-making
A2	BFS	iterative, low processing	building block
A3	CONN	iterative, medium processing	building block
A4	CD	iterative, medium or high processing	social network
A5	EVO	iterative (multi-level), high processing	prediction

B. Selection of graphs and algorithms

This section presents a selection of graphs and algorithms, which is akin to identifying some of the main functional requirements of graph-processing systems. We further discuss the representativeness of our selection in Section V.

1) Graph selection: The main goal of the graph selection step is to select graphs with different characteristics but with comparable representation. We use the classic graph formalism [34]: a graph is a collection of vertices V (also called nodes) and edges E (also called arcs or links) which connect the vertices. A single edge is described by the two vertices it connects: $e = (u, v)$. A graph is represented by $G = (V, E)$. We consider both directed and undirected graphs. We do not use other graph models (e.g., hypergraphs).

Regarding the graph characteristics, we select graphs with a variety of values for the number of nodes and edges, and with different structures (see Table II). We store the graphs in plain text with a processing-friendly format but without indexes. In our format, vertices have integers as identifiers. Each vertex is stored in an individual line, which for undirected graphs, includes the identifier of the vertex and a comma-separated list of neighbors; for directed graphs, each vertex line includes the vertex identifier and two comma-separated lists of neighbors, corresponding to the incoming and to the outgoing edges. Thus, we do not consider other data models proposed for exchanging and using graphs such as complex plain-text representations, universal data formats (e.g. XML), relational databases, relationship formalisms (e.g., RDF), etc.

Why these datasets? We select seven graphs which could match, in scale and diversity, the datasets used by SMEs. Table II shows the characteristics of the selected graph datasets. The graphs have diverse sources (e-business, social network, online gaming, citation links, and synthetic graph), and a wide range of different sizes and graph metrics (e.g., high vs. low degree, 1,663 vs. 2, respectively, directed and undirected graphs, etc.). The synthetic graph ("Synth" in Table II) is produced by the generator described in Graph500 [31]. The other graphs have been extracted from real-world use, and have been shared through the Stanford Network Analysis Project (SNAP) [35]) and the Game Trace Archive (GTA) [2].

2) Algorithm selection: Why these algorithms? We have conducted a comprehensive survey of graph-processing articles published in 10 representative conferences, in recent years; in total, over 100 articles (see technical report [33]). We found that a large variety of graph processing algorithms exist in practice [36] and are likely used by SMEs. The algorithms can be categorized into several groups by functionality, consumption of resources, etc. We focus on algorithm functionality and select one exemplar of each of the following five algorithmic classes, which are common in our survey: general statistics, graph traversal (used in Graph500), connected components, community detection, and graph evolution. We describe in the following the five selected algorithms and summarize their characteristics in Table III.

The General statistics (STATS) algorithm computes the number of vertices and edges, and the average of the local clustering coefficient of all vertices. The results obtained with STATS can provide the graph analyst with an overview of the structure of the graph.

Breadth-first search (BFS) is a widely used algorithm in graph processing, which is often a building block for more complex algorithms, such as item search, distance calculation, diameter calculation, shortest path, longest path, etc. BFS allows us to understand how the tested platforms cope with lightweight iterative jobs.

Connected component (CONN) is an algorithm for extracting groups of vertices that can reach each other via graph edges. This algorithm produces a large amount of output, as in many graphs the largest connected component includes a majority of the vertices.

Community detection (CD) is important for social network applications, as users of these networks tends to form communities, that is, groups whose constituent nodes form more

TABLE IV
SELECTED PLATFORMS.

Platform	Version	Type	Release date
Hadoop	hadoop-0.20.203.0	Generic, Distributed	2011-05
YARN	hadoop-2.0.3-alpha	Generic, Distributed	2013-02
Stratosphere	Stratosphere-0.2	Generic, Distributed	2012-08
Giraph	Giraph 0.2 (revision 1336743)	Graph, Distributed	2012-05
GraphLab	GraphLab version 2.1.4434	Graph, Distributed	2012-10
Neo4j	Neo4j version 1.5	Graph, Non-distributed	2011-10

relationships within the group than with nodes outside the group. Communities are also important in the gaming industry, as the market has an increasingly larger share of social games or of games for which the social component is important.

Graph evolution (EVO): an accurate EVO algorithm not only can predict how a graph structure will evolve over time, but can also help to prepare for these changes (for example data size increase). Thus, graph evolution is an important topic in the field of large-scale graph processing.

STATS and BFS are textbook algorithms. For CONN, CD and EVO, there are a number of variations. Considering the reported performance and accuracy of these algorithms, we select a cloud-based connected component algorithm created by Wu and Du [11], the real-time community detection algorithm proposed by Leung et al. [14], and the the Forest Fire Model for graph evolution designed by Leskovec et al. [1].

III. EXPERIMENTAL SETUP

The method introduced in Section II defines a benchmarking skeleton. In this section we create a full benchmarking suite (bar the issues explained in Section V) by implementing the graph-processing algorithms of a selected set of test platforms, and by configuring and tuning these platforms.

A. Platform selection

We use a simple taxonomy of platforms for graph processing. By their use of computing machines, we identify two main classes of platforms: non-distributed platforms and distributed platforms; distributed platforms use multiple computers when processing graphs. Orthogonally to the issue of distributed machine use, we divide platforms into graph-specific platforms and generic platforms; graph specific platforms are designed and tuned only for processing graph data. Importantly, we omit in our taxonomy parallel platforms; for the scale in our real-world experiments, we see the performance of distributed systems as being a conservative estimate of what a similarly sized but parallel system can achieve.

Why these platforms? We select for this study a graph-specific non-distributed platform, and both graph-specific and generic distributed platforms. Because of what we see that relatively little interest in the community, we do not select for this study any generic and non-distributed platform. Table IV summarizes our selected platforms: Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. These six selected platforms are popular and may be used for graph processing. We introduce each platform in the following, in turn.

Hadoop [22] is an open-source, generic platform for big data analytics. It is based on the MapReduce programming model. Hadoop has been widely used in many areas and applications, such as log analysis, search engine optimization, user interests prediction, advertisement, etc. Hadoop is becoming the de-facto platform for batch data processing. Hadoop's programming model may have low performance and high resource consumption for iterative graph algorithms, as a consequence of the structure of its map-reduce cycle. For example, for iterative graph traversal algorithms Hadoop would often need to store and load the entire graph structure during each iteration, to transfer data between the map and reduce processes through the disk-intensive HDFS, and to run an convergence-checking iteration as an additional job. However, comprehensive results regarding graph-processing using Hadoop have not yet been reported.

YARN [23] is the next generation of Hadoop. YARN can seamlessly support old MapReduce jobs, but was designed to facilitate multiple programming models, rather than just MapReduce. A major contribution of YARN is to separate functionally resource management and job management; the latter is done in YARN by a per-application manager. For example, the original Apache Hadoop MapReduce framework has been modified to run MapReduce jobs as an YARN application manager. YARN is still under development. We select YARN because our hypothesis is that even for the same programming model (YARN and Hadoop), the architecture of the execution engine matters.

Stratosphere [25] is an open-source platform for large-scale data processing. Stratosphere consists of two key components: Nephele and PACT. Nephele is the scalable parallel engine for the execution of data flows. In Nephele, jobs are represented as directed acyclic graphs (DAG), a job model similar for example to that of the generic distributed platform Dryad [24]. For each edge (from task to task) of the DAG, Nephele offers three different types of channels for transporting data, through the network, in-memory, and through files. PACT is a data-intensive programming model that extends the MapReduce model with three more second-order functions (Match, Cross, and CogGroup, in addition to Map and Reduce). PACT supports several user code annotations, which can inform the PACT compiler of the expected behavior of the second-order functions. By analyzing this information, the PACT compiler can produce execution plans that avoid high cost operations such as data shipping and sorting, and data spilling to the disk. Compiled PACT programs are converted into Nephele DAGs and executed by the Nephele data flow engine. HDFS is used for Stratosphere as the storage engine.

Giraph [27] is an open-source, graph-specific distributed platform. Giraph uses the Pregel programming model, which is a vertex-centric programming abstraction that adapts the Bulk Synchronous Parallel (BSP) model. An BSP computation proceeds in a series of global supersteps. Within each superstep, active vertices execute the same user-defined computation, and create and deliver inter-vertex messages. Barriers ensure synchronization between vertex computation: for the current superstep, all vertices complete their computation and all messages are sent before the next superstep can start.

Giraph utilizes the design of Hadoop, from which it leverages only the Map phase. For fault-tolerance, Giraph uses periodic checkpoints; to coordinate superstep execution, it uses ZooKeeper. Giraph is executed in-memory, which can speed-up job execution, but, for large amounts of messages or big datasets, can also lead to crashes due to lack of memory.

GraphLab [29] is an open-source, graph-specific distributed computation platform implemented in C++. Besides graph processing, it also supports various machine learning algorithms. GraphLab stores the entire graph and all program state in memory. To further improve performance, GraphLab implements several mechanisms such as: supporting asynchronous graph computation to alleviate the waiting time for barrier synchronization, using prioritized scheduling for quick convergence of iterative algorithms, and efficient graph data structures and data placement. To match the execution mode of the other platforms, we run all our GraphLab experiments in a synchronized mode.

Neo4j [19] is one of the popular open-source graph databases. Neo4j stores data in graphs rather than in tables. Every stored graph in Neo4j consists of relationships and vertices annotated with properties. Neo4j can execute graph-processing algorithms efficiently on just a single machine, because of its optimization techniques that favor response time. Neo4j uses a two-level, main-memory caching mechanism to improve its performance. The file buffer caches the storage file data in the same format as it is stored on the durable storage media. The object buffer caches vertices and relationships (and their properties) in a format that is optimized for high traversal speeds and transactional writes.

B. Platform and experiment configuration

Platform tuning: The performance of these systems depends on tuning. Several of the platforms tested in this work have tens to hundreds of configuration parameters, whose actual value can potentially change the performance of the platform. We use common best-practices for tuning each of the platforms, as explained in our technical report [33].

Hardware: We deploy the distributed platforms on DAS4 [37], which is to provide a common computational infrastructure for researchers within Advanced School for Computing and Imaging in the Netherlands. Each machine we used in the experiments from DAS4 consists of a Intel Xeon E5620 2.4 GHz CPU (dual quad-core, 12 MB cache) and a total memory of 24 GB. All the machines are connected by 1 Gbit/s Ethernet network. NFS is used as the file system in DAS4. The operation system installed on each machine is CentOS release 6.3 with the kernel version 2.6.32. We use a single machine with one single enterprise SATA disk (SATA 3 Gbit/s, 7200 rpm, 32 MB cache) for the Neo4j experiments.

Platform configuration, number of nodes: We deploy the distributed platforms on 20 up to 50 computing machines of DAS4. We set the Neo4j on a single DAS4 machine with regular configuration. For all the experiments of Hadoop, YARN, Stratosphere, and GraphLab, besides the computing machines, we allocate an additional node to take charge of all master services. For Giraph, we use one more node for running ZooKeeper.

Parameters of Algorithms: We try to configure each algorithm with default parameter values. STATS and CONN do not need any parameters. For BFS, we randomly pick a vertex to be the source for each graph. We use only out-edges to propagate for directed graph, thus the directed graphs are not entirely traversed. We set the parameters of algorithms identically on all platforms.

Further experiment configuration: Unless otherwise stated, we repeat each experiment 10 times, and we report the average results from these runs. (An example where 10 repetitions would take too long is presented in Section IV-C).

IV. Experimental results

In this section we present a selection of the experimental results. We evaluate the six graph processing platforms selected in Section III, using the process and metrics, and the datasets and algorithms introduced in Section II. The complete results are available through our technical report [33]. Compared with the previous work (Section VI), our experiments show more comprehensive and quantitative results in diverse metrics.

The experiments we have performed are:

- Basic performance (Section IV-A): we have measured the job execution time on a fixed infrastructure. Based on these execution times, we further report throughput numbers, using the edges per second (EPS) and vertices per second (VPS) metrics.
- Resource utilization (only in technical report [33]): we have investigated the CPU utilization, memory usage, and network traffic.
- Scalability (Section IV-B): we have measured the horizontal and vertical scalability of the platforms; we report the execution time and the normalized edges per second (NEPS) for interesting datasets.
- Overhead (Section IV-C): we have analyzed the execution time in detail, and report important findings related to the platform overhead.

A. Basic performance: job execution time

The fixed infrastructure we use for our basic performance measurements is a cluster of 20 homogeneous computing nodes provisioned from DAS4. With the configuration in [33], each node is restricted at using a single core for computing. We configure the cluster as follows. For the experiments on Hadoop and YARN, we run 20 map tasks and 20 reduce tasks on the 20 computing nodes. Due to the settings used for Hadoop [33], the map phase will be completed in one wave; all the reduce tasks can also be finished in one wave, without any overlap with the map phase [38]. In Giraph, Stratosphere, and GraphLab, we set the parallelization degree to 20 tasks, also equal to the total number of computing nodes.

With these settings, we run the complete set of experiments (6 platforms, 5 different applications, and 7 datasets) and measure the execution time for each combination. In the remainder of this section, we present a selection of our results.

TABLE V
STATISTICS OF BFS.

	G1	G2	G3	G4	G5	G6	G7
Coverage [%]	99.9	98.5	100	0.1	100	100	100
Iterations	68	8	9	11	6	8	23

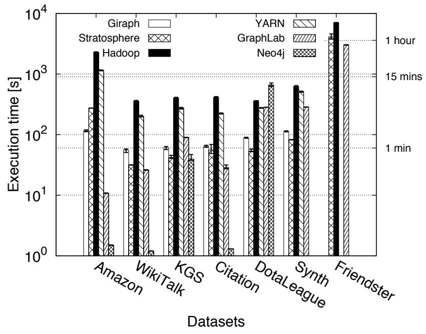

Fig. 1. The execution time of algorithm BFS of all datasets of all platforms.

Key findings:

- There is no overall winner, but Hadoop is the worst performer in all cases.
- Multi-iteration algorithms suffer for additional performance penalties in Hadoop and YARN.
- EPS and VPS are suitable metrics for comparing the platforms throughput.
- The performance of all the platforms is stable, with the largest variance around 10%.
- Several of the platforms are unable to process all datasets for all algorithms, and crash.

1) Results for one selected algorithm: We present here the results obtained for one selected algorithm, BFS (see Section II-B2).

Because the starting node for the BFS traversal will impact performance by limiting the coverage and number of iterations of the algorithm, we summarize in Table V the vertex coverage and iteration count observed for the BFS experiments presented in this section. Overall, BFS covers over 98% of the vertices, with the exception of the Citation (G4) dataset. The iteration count depends on the structure of each graph and varies between 6 and 68; we expect higher values to impact negatively the performance of Hadoop.

We depict the performance of the BFS graph traversal in Figure 1 and discuss in the following the main findings. Similarly to most figures in this section, Figure 1 has a logarithmic vertical scale.

Hadoop always performs worse than the other platforms, mainly because Hadoop has a significant I/O between two continuous iterations (see Section III). In these experiments, Hadoop does not use spills, so it has no significant I/O within the iteration. As expected, the I/O overhead of Hadoop is worse when the number of BFS iterations increases. For example, although Amazon is the smallest graph in our study, it has the largest iteration count, which leads to a very long execution time. YARN performs only slightly better than Hadoop—it has not been altered to support iterative applications. Although Stratosphere is also a generic data-processing platform, it performs much better than Hadoop and YARN (up to an order of magnitude lower execution time). We attribute this to Stratosphere's ability to optimize the execution plan based on code annotations regarding data sizes and flows, and to the much more efficient use of the network channel.

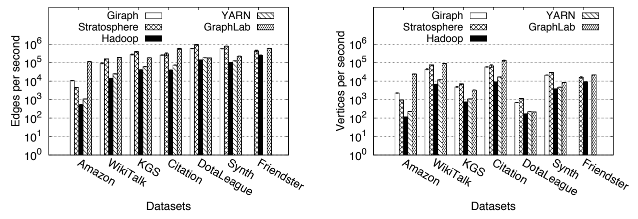

Fig. 2. The EPS and VPS of executing BFS.

In contrast to the generic platforms, for Giraph and GraphLab the input graphs are read only once, and then stored and processed in-memory. Both Giraph and GraphLab realize a dynamic computation mechanism, by which only selected vertices will be processed in each iteration. This mechanism reduces the actual computing time for BFS, in comparison with the other platforms (more details are discussed in Section IV-C). In addition, GraphLab also addresses the problem of smart dataset partitioning, by limiting the cut-edges between machines when splitting the graph. These systemic improvements make the performance of both Giraph and GraphLab less affected by large BFS iteration counts than the performance of other distributed platforms.

Because of the two-level main-memory cache of Neo4j, we differentiate two types of executions: cold-cache (first execution) and hot-cache (follow-up executions). Figure 1 depicts the average results obtained for hot-cache executions. The two-level cache allows Neo4j to achieve excellent hot-cache execution times, especially when the graph data accessed by the algorithms fits in the cache. However, the cold-cache execution can be very long: for example, the ratios between the cold-cache and hot-cache BFS executions for Citation and DotaLeague are 45 and 5, respectively. Even for cold-cache execution, Neo4j reads from the database only the graph data needed by the algorithm. This "lazy read" mechanism minimizes the I/O overhead and accelerates traversal on the graphs where the BFS coverage of the graph is limited, e.g., for Citation. However, limited by the resources of a single machine, the performance of Neo4j becomes significantly worse when the graph exceeds the memory capacity. For example, the hot-cache value of Synth is about 17 hours, exceeding the scale of Figure 1.

We now report on the *achieved throughput for the BFS algorithm*, in both EPS and VPS, for all platforms and datasets (Figure 2). We note that throughput is a metric that takes into account the dataset structure and provides an indication of the platforms performance per data item—be it an edge or a vertex. For example, KGS and Citation, which have similar numbers of edges, file sizes, and BFS iteration counts, achieve similar EPS values on most platforms. The exception is GraphLab, in which the EPS of Citation is about two times larger than that of KGS. This is due to the restriction of

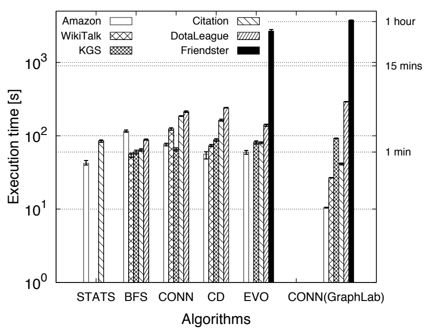

Fig. 3. The execution time of all algorithms for all datasets running on Giraph, and for CONN running on GraphLab (right-most bars).

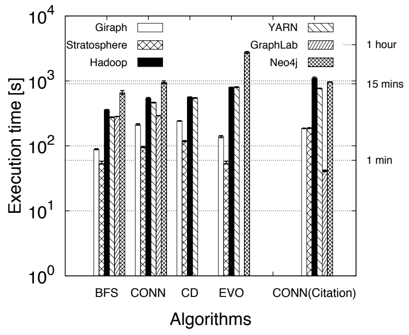

Fig. 4. The execution time for all platforms, running all algorithms for the DotaLeague dataset, and CONN for the Citation dataset (right-most bars).

GraphLab to process only directed graphs, which has required the conversion of the undirected KGS to a directed version. This operation lead to a doubling in the number of edges, resulting in a proportional increase of the execution time.

2) Results for two selected platforms: We focus in this section on the graph-specific platforms (Giraph and GraphLab) and discuss their performance for all the algorithms and datasets, as depicted in Figure 3.

As Giraph is an in-memory-only platform, its performance is not affected by the large penalties of I/O operations. Figure 3 shows that the execution time for all experiments is below 100 seconds. However, when the amount of messages between computing nodes becomes extremely large (tens of gigabytes), Giraph crashes. For example, Giraph crashes for the STATS algorithm running on the WikiTalk dataset; for Friendster, the largest of our datasets, Giraph completes only the EVO algorithm, for which our graph evolution algorithm generates relatively few messages. From the selected results, GraphLab performs better than Giraph for the CONN algorithm for most graphs. Moreover, GraphLab is able to process even the largest graph in this study.

3) Results for two selected datasets: Finally, to understand the impact of algorithm complexity on each platform, we focus now on two interesting datasets—DotaLeague and Citation. We depict their performance, for all algorithms running on all platforms, in Figure 4.

Because Friendster is too large for some platforms, we present here the results for graphs that all platforms can process: the second-largest real graph, DotaLeague, and the small Citation graph. Even for the second-largest graph, Giraph, Hadoop and YARN crashed when running STATS; we also had to terminate Stratosphere after running STATS for nearly 4 hours without success; similarly, STATS and CD run for more than 20 hours in Neo4j and are not shown in Figure 4. For the other algorithms, BFS, CONN, CD, and EVO, the number of iterations is between 4 and 6. From Figure 4, the execution time of BFS is lower than the execution time of CONN and CD, on all platforms. In each iteration of CONN and CD, many more vertices will be active, in comparison to BFS. Furthermore, in CONN, the number of active vertices stays relatively constant in each iteration, while CD is more compute-intensive and variable. For EVO, Stratosphere takes advantage of its programming model, as it can represent one EVO iteration by a single map-reduce-reduce procedure; in contrast, Hadoop and YARN need to run two MapReduce jobs per iteration and thus their execution time increases.

Citation is much smaller and sparser than DotaLeague. The CONN of Citation takes 20 iterations. The execution time of CONN of Citation on Hadoop, YARN, and Stratosphere increases compared with 6-iteration CONN of DotaLeague. As we explained for the analysis of BFS (Section IV-A1), more iterations result in higher I/O and other overheads.

B. Evaluation of scalability

In this section, we evaluate the horizontal and vertical scalability of the distributed platforms. Besides the job execution time, we also report the NEPS for comparing the performance per computing unit.

To allow a comparison with the previous experiments, we use BFS results. To test scalability, we use the two largest real graphs in our study, Friendster and DotaLeague (the results of DotaLeague are in our technical report [33]). For testing horizontal scalability, we increase the number of machines from 20 to 50 by a step of 5, and keep using a single computing core per machine. For testing vertical scalability, we keep the cluster size at 20 computing machines and increase the number of computing cores per machines from 1 to 7. We step up the number of map (reduce) tasks and parallelization degree equally to the available computing cores.

Key findings:

- Some platforms can scale up reasonably with cluster size (horizontally) or number of cores (vertically).
- Increasing the number of computing cores may lead to worse performance, especially for small graphs.
- The normalized performance per computing unit mostly decreases with the increase of cluster size and with the number of computing cores per node.

1) Horizontal scalability: Figure 5 shows the horizontal scalability of BFS for Friendster (G7). Most of the platforms presents significant horizontal scalability, except for GraphLab, which exhibits little scalability. The horizontal scalability of GraphLab is constrained by the graph loading phase using one single file. We thus explore tuning GraphLab: for GraphLab(mp) we split the input file into *m*ultiple separate *p*ieces, as many as the MPI processes. GraphLab(mp) has much lower execution time than GraphLab. Moreover,

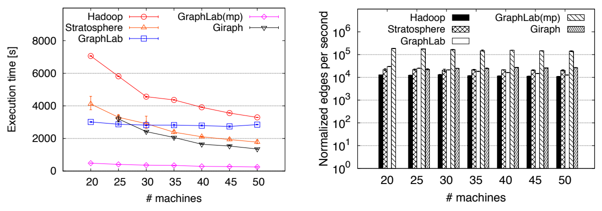

Fig. 5. The horizontal scalability (left) and NEPS (right) of processing G7.

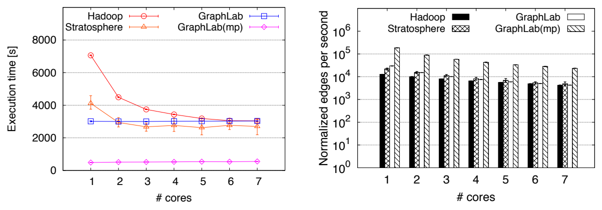

Fig. 6. The vertical scalability (left) and NEPS (right) of processing G7.

TABLE VI
DATA INGESTION TIME.

	G1	**G2**	**G3**	**G4**	**G5**	**G6**	**G7**
HDFS [s]	1.2	1.8	3.0	3.9	7.0	10.9	312.0
Neo4j [**h**]	2.0	17.2	2.6	28.8	3.7	24.7	N/A

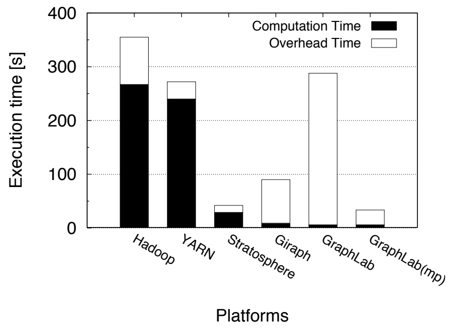

Fig. 7. The execution time breakdown of platforms for BFS.

GraphLab(mp) is scalable, as its execution time decrease from about 480 seconds to 250 seconds when resources are added.

We further investigate the performance per computing unit (computing node) to check if they also be improved. We calculate the EPS from the execution time and normalize it by the number of computing nodes to get the NEPS. The maximum value of NEPS can be reached at different sizes of the cluster, for different platforms. For example, the NEPS of Hadoop and Giraph peaks at 30 and 40 computing nodes of Friendster, respectively. However, the general trend of NEPS is to decrease with the increase of cluster size. We have obtained similar results for the vertex-centric equivalent of NEPS, NVPS.

2) Vertical scalability: Figure 6 shows the vertical scalability of running BFS for Friendster (G7). There is no result of Giraph and YARN of Friendster, because both YARN and Giraph crashed on 20 computing machines. For Friendster, both Hadoop and Startosphere can benefit from using more computing cores. However, after 3 cores, the improvement become negligible. By using more cores, graphs can be processed with higher parallelism, but may also incur latency, for example, due to concurrent accesses to the disk. For GraphLab(mp), for which we split the Friendster file into more pieces with the increase of the number MPI processes, the job execution time does not decrease correspondingly. The reason is that each MPI instance (or machine) has a just single loader for input files, thus in one machine, the MPI processes cannot load graph pieces in parallel.

We check the performance per computing unit (computing core) by NEPS in vertical scalability. We can find similar results to that of horizontal scalability, all NEPS drops for all platforms. The competition between computing cores makes the reduce of execution time not significant enough to improve the average performance of computing cores.

C. Evaluation of overhead

In this section, we evaluate two elements of overhead: data ingestion time and execution time overhead.

Key findings:

- The data ingestion time of Neo4j matches closely the characteristics of the graph. Overall, data ingestion takes much longer for Neo4j than for HDFS.
- The data ingestion time of HDFS increases nearly linearly with the graph size.
- The percentage of overhead time in execution time is diverse across the platforms, algorithms, and graphs.

For Neo4j, data ingestion process converts input graphs to the format used by the Neo4j graph database. In contrast, the distributed platforms evaluated in this work use HDFS, which means for them data ingestion consists of data transfers from the local file system to HDFS. GraphLab even does not need data ingestion if using the local file system (i.e., NFS). Only for Neo4j, because data ingestion takes long (up to days), we only evaluate the data ingestion for Neo4j through one experiment repetition.

Table VI summarizes the data ingestion results. The data ingestion time of Neo4j is up to several orders of magnitude longer than that of HDFS. In our experimental environment, which uses enterprise-grade magnetic disks, the data ingestion time of HDFS increases by about 1 second for every 100 MB of graph data. In contrast, the data ingestion time of Neo4j depends on the structure and scale of graphs, so it changes irregularly across the datasets in this study. Dominguez-Sal et al. [4] report similar results about data ingestion time in their survey of graph database performance.

We define computation time as the time used for making progress with the graph algorithms. The overhead time is the remainder from subtracting the computation time from the job execution time. Thus, the overheads include the time for read and write, and for communication. The fraction of time spent with overheads varies across the platforms (Figure 7). Although BFS is not a compute-intensive algorithm, Hadoop and Stratosphere need to traverse all vertices, which increases their computation time. In GraphLab, most of the time is spent on loading the graph into memory and on finalizing the results. The percentage of overhead time on each platform is closely related to the complexity of the algorithm and the characteristics of graph. For example, we also found that for Citation, the percentage of overhead time is 98% and 70% for BFS and CONN, respectively (see technical report [33]).

V. Discussion: Towards a Benchmark

The method proposed in Section II raises several methodological and practical issues that prevent it from being a benchmark. We argue that our method can result in meaningful, comprehensive performance evaluation of graph-processing platforms, but the path towards an industry-accepted benchmark still raises sufficient challenges. Outside the scope of this work, we continue to pursue resolving these issues via the SPEC Cloud Working Group.

Methodologically, our method has limitations in its process, workload design, and metrics design. Specifically, our *method* does not offer a detailed, infrastructure- and platform-independent process; for example, it does not limit meaningfully the amount of tuning done to a system prior to benchmarking and it does not precisely specify the acceptable components of a platform (should a cloud-based platform include the Internet linking its users to the data center?). The *workload design*, although it covers varied datasets and algorithms, does not feature an industry-accepted process of selection for them, and does not select datasets and algorithms that can stress a specific bottleneck in the system under test. Proving *algorithmic coverage* is currently not feasible, due to field fragmentation and lack of public workload traces; solving this "chicken-and-egg" problem cannot be properly addressed without collecting workload traces for several years and from several major operators.

Metrics-wise, our method does not provide only *a single result*—which helps with the analysis of the causes of performance gaps between platforms—; does not provide metrics for a variety of interesting platform characteristics (e.g., power consumption, cost, efficiency, and elasticity); and could do more in terms of *normalized metrics* (i.e., by normalizing by various types of resources provided by the system, such as number of cores or size of memory). Using other metrics is outside the scope of our work, as different communities are interested in different operational aspects (SPEC identifies tens of relevant metrics).

From a practical perspective, our method has limitations in portability, time, and cost. The *portability* is limited by the need to re-implement algorithms for each platform and to re-configure platforms for each experiment. The *time* spent in implementing our method is analyzed in our technical report [33]. The *cost* of performing a benchmark, in particular in *tuning*, is a non-trivial issue, for which few benchmarks provide a solution. Another non-trivial practical aspect is reporting (an outcome of the analysis stage), which our method does not precisely specify. In contrast, SPEC benchmark users can report results for baseline (not tuned) and peak (tuned) systems, and SPEC results include a full disclosure of the parameters used in configuring the systems; however, SPEC benchmarks are sophisticated products and the result of years of development.

TABLE VII
Overview of related work. Legend: V–vertices, E–edges, C–computers.

Platforms	Algorithms	Dataset type	Largest dataset	System
Neo4j, MySQL [40]	1 other	synthetic	100 KV	1 C
Neo4j, etc. [4]	3 others	synthetic	1 MV	1 C
Pregel [5]	1 other	synthetic	50 BV	300 C
GPS, Giraph [41]	CONN, 3 others	real	39 MV, 1.5 BE	60 C
Trinity, etc. [30]	BFS, 2 others	synthetic	1 BV	16 C
PEGASUS [28]	CONN,2 others	synthetic, real	282 MV	90 C
CGMgraph [42]	CONN, 4 others	synthetic	10 MV	30 C
Hadoop, GraphLab, etc. [43]	1 other	real	3 MV, 117 ME	32 C
PBGL, CGMgraph [44]	CONN, 3 others	synthetic	70 MV, 1 BE	128 C
HaLoop, Hadoop [26]	2 others	synthetic, real	1.4 BV, 1.6 BE	90 C
Hadoop, PEGASUS [45]	1 other	synthetic, real	1 BV, 20 BE	32 C
Our work	**5 classes**	**synthetic, real**	**66 MV, 1.8 BE**	**50 C**

Experiments with larger environments and datasets, and with new algorithms and metrics, can add to the bulk of the results presented in this work. However, the need for such experiments is not supported by existing published evidence: as we show in Section VI, our work already extends and complements previous work. Requiring experiments in larger clusters, while relevant for companies such as Facebook (thousands of nodes), does not match the needs of SMEs that want efficient graph-processing, and the information we have about state-of-the-art (Section VI).

These limitations also affect other benchmarks [39] and performance evaluation studies included in the related work of this study. As we indicated in the introduction of our article, we point out that the de-facto standard in benchmarking graph-processing platforms is Graph500 (one algorithm and one graph type); in contrast, our work provides a significant improvement in both algorithms (processing patterns and scope) and datasets (two public graph archives, several application domains, and various graph structures).

VI. Related Work

Many previous studies focus on performance evaluation of graph-processing, for different platforms. Table VII summarizes these studies and compares them with our work. Overall, for the studies in our survey, most of the datasets included in previous evaluation are synthetic graphs. Although some of the synthetic graphs are extremely large, they may not have the characteristics of real graphs. Our evaluation selects 6 *real* graphs and 1 synthetic graph with various characteristics. *Relative to our study, fewer classes of algorithms are used to compare the performance of platforms*. From our observation, a very limited number of metrics have been reported, with many of the previous studies focusing only on the job execution time. *Our work evaluates performance much more in-depth, by considering more types of metrics*. Finally, previous research compares few platforms; in contrast, *we investigate 6 popular platforms with different architectures*. *Our environment is of similar scale with state-of-the-art studies in distributed systems*.

VII. Conclusion

A quickly increasing number of platforms can process large-scale graphs, and have thus become potentially interesting for a variety of users and application domains. We focus in this work on SMEs, which are businesses with little resources to spare in their graph-processing clusters. To compare in-depth the performance of graph-processing platforms, and thus facilitate platform selection and tuning for SMEs, we have proposed in this work an empirical method and applied it to obtain a comprehensive performance study of six platforms.

Our method defines an empirical performance evaluation process and selects metrics, datasets, and algorithms; thus, it acts as a benchmarking suite despite not covering all the methodological and practical aspects of a true benchmark. Our method focuses on four performance aspects: raw performance, resource utilization, scalability, and overhead. We use both performance and throughput metrics, and we also use normalized metrics to characterize scalability. We implement a benchmarking suite that uses five representative graph algorithms—general statistics, breadth-first search, connected component, community detection, and graph evolution—, and seven graphs that represent graph structures for multiple application domains, with sizes up to 1.8 billion edges and tens of GB of stored data.

Using our benchmarking suite, we conduct a first detailed, comprehensive, real-world performance evaluation of six popular platforms for graph-processing, namely, Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. Our results show quantitatively and comparatively the highlights and weaknesses of the tested platforms. The main lessons are listed at the start of each experiment.

References

[1] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *SIGKDD*, 2005.

[2] Y. Guo and A. Iosup, "The Game Trace Archive," in *NetGames*, 2012.

[3] X. Zhang and G. Agrawal, "Supporting high performance bioinformatics flat-file data processing using indices," in *IPDPS*, 2008.

[4] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazán, and J. Larriba-Pey, "Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark," *Web-Age Information Management*, pp. 37–48, 2010.

[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *SIGMOD*, 2010, pp. 135–146.

[6] A. Lugowski, D. M. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis," in *SDM*, 2012.

[7] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, "Multithreaded Clustering for Multi-level Hypergraph Partitioning," in *IPDPS*, 2012.

[8] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *IPDPS*, 2006.

[9] A. Buluç and K. Madduri, "Graph Partitioning for Scalable Distributed Graph Computations," in *Graph Partitioning and Graph Clustering*, 2012.

[10] E. Solomonik, A. Buluç, and J. Demmel, "Minimizing Communication in All-Pairs Shortest Paths," in *IPDPS*, 2013.

[11] B. Wu and Y. Du, "Cloud-Based Connected Component Algorithm," in *ICAICI*, 2010, pp. 122–126.

[12] D. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU graph traversal," in *PPOPP*, 2012.

[13] E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable Multi-threaded Community Detection in Social Networks," in *IPDPS Workshops*, 2012.

[14] I. X. Leung, P. Hui, P. Liò, and J. Crowcroft, "Towards Real-Time Community Detection in Large Networks," *Physical Review E*, 2009.

[15] Ü. V. Çatalyürek, K. Kaya, A. E. Sariyüce, and E. Saule, "Shattering and Compressing Networks for Betweenness Centrality," in *SDM*, 2013.

[16] M. Redekopp, Y. Simmhan, and V. K. Prasanna, "Optimizations and Analysis of BSP Graph Processing Models on Public Clouds," in *IPDPS*, 2013.

[17] T. G. Kolda, A. Pinar, and C. Seshadhri, "Triadic Measures on Graphs: The Power of Wedge Sampling," in *SDM*, 2013.

[18] K. Dempsey, K. Duraisamy, H. Ali, and S. Bhowmick, "A parallel graph sampling algorithm for analyzing gene correlation networks," *Procedia Computer Science*, 2011.

[19] "Neo4j." [Online]. Available: http://www.neo4j.org/

[20] B. Iordanov, "HyperGraphDB: a Generalized Graph Database," *Web-Age Information Management*, pp. 25–36, 2010.

[21] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *OSDI*, 2012.

[22] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[23] "YARN." [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Review*, 2007.

[25] D. Warneke and O. Kao, "Nephele: Efficient Parallel Data Processing in the Cloud," in *MTAGS*, 2009.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *VLDB*, pp. 285–296, 2010.

[27] "Giraph." [Online]. Available: http://giraph.apache.org/

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *ICDM*. IEEE, 2009, pp. 229–238.

[29] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *VLDB*, 2012, pp. 716–727.

[30] B. Shao, H. Wang, and Y. Li, "The Trinity Graph Engine," Technical Report 161291, Microsoft Research, Tech. Rep., 2012.

[31] Graph500. [Online]. Available: http://www.graph500.org/

[32] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance Characteristics of Graph500 on Large-Scale Distributed Environment," in *ISWC*, 2011, pp. 149–158.

[33] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report," Delft University of Technology, Tech. Rep. PDS-2013-004, 2013, http://www.pds.ewi.tudelft.nl/research-publications/technical-reports/2013/.

[34] R. J. Wilson, *Introduction to Graph Theory*. Academic Press, 1972.

[35] "SNAP." [Online]. Available: http://snap.stanford.edu/index.html

[36] R. A. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in *SC*, 2010.

[37] "DAS4." [Online]. Available: http://www.cs.vu.nl/das4/

[38] B. Ghit, N. Yigitbasi, and D. Epema, "Resource Management for Dynamic MapReduce Clusters in Multicluster Systems," in *MTAGS*. IEEE, 2012.

[39] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS*, 2012.

[40] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective," in *Southeast regional conference*, 2010.

[41] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," Tech. Rep., 2012.

[42] A. Chan, F. Dehne, and R. Taylor, "CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines," *IJHPCA*, vol. 19, no. 1, pp. 81–97, 2005.

[43] B. Elser and A. Montresor, "An Evaluation Study of BigData Frameworks for Graph Processing," in *IEEE BigData*, 2013.

[44] D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," *POOSC*, 2005.

[45] K. Kambatla, G. Kollias, and A. Grama, "Efficient Large-Scale Graph Analysis in MapReduce," in *PMAA*, 2012.