Delft University of Technology
Parallel and Distributed Systems Report Series

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

Yong Guo, Marcin Biczak, Ana Lucia Varbanescu,
{Yong.Guo}@tudelft.nl

Alexandru Iosup, Claudio Martella, and Theodore L. Willke

To be submitted after editing

Report number PDS-2013-004

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

Giraph, GraphLab, and other graph-processing platforms are increasingly used in a variety of domains, including social networking and gaming, targeted advertisements, and bioinformatics. 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 and even dataset. To alleviate this challenge, in this work we propose and apply an empirical method for evaluating and comparing graph-processing platforms. We define a benchmarking suite for graph-processing platforms, which includes a comprehensive process and a selection of representative metrics, datasets, and algorithmic classes. In our process, we focus on evaluating for each system the basic performance, the resource utilization, the scalability, and various performance overheads. Our selection includes five classes of graph-processing algorithms and seven graphs of up to 1.8 billion edges each. We report on job execution time and various normalized metrics. Finally, we use our benchmarking suite on six different platforms and, besides the valuable insights gained for each platform, we also present the first comprehensive comparison of graph-processing platforms.

Contents

List of Figures

List of Tables

1 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, 2, 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 contrast to these previous studies, and to the Graph500 benchmark, in this work we propose a comprehensive experimental method for comparing graph-processing platforms, implement it as a benchmarking suite, and apply it to six real and popular graph-processing platforms.

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, tuning of the system and of the application, and sharing of best-practices. 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 input dataset, the algorithm, and the graph-processing platform. Thus, gaining a thorough understanding 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 traversing graphs [7, 8], for detecting communities [9, 10, 11], for searching for important vertices [12, 13], for sampling graphs [14], for predicting graph evolution [15, 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 [16], HyperGraphDB [17], and GraphChi [18] 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 be less efficient than even single-node platforms. Generic data processing systems such as Hadoop [19], YARN [20], Dryad [21], Stratosphere [22], and HaLoop [23] can scale out on multiple nodes, but may exhibit low performance due to distribution and new overheads. Graph-specific platforms such as Pregel [5], Giraph [24], PEGASUS [25], GraphLab [26], and Trinity [27] also address various limitations in large-scale graph processing.

New performance evaluation and benchmarking suites are needed to respond to the three sources of diversity, that is, to provide comparative information about the performance and other non-functional characteristics of different platforms, through the use of empirical methods and processes. However, the state-of-the-art in graph-processing platform evaluation has very limited breadth and depth. For example, Graph500 is 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 [28, 29], which is rarely found or reproduced by common graph processing deployments and thus rarely reaches the users. 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 use, vertical and horizontal scalability, and overhead. Using this method, we create the equivalent of a benchmarking suite by selecting and implementing five graph algorithms and seven large-scale 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 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. Our work also aligns with the goals and ongoing activity of the SPEC Research Group and its Cloud Working Group, of which some of the authors are members.

Towards answering the research question, our main contributions are:

1. We propose a method for the comprehensive evaluation of graph processing platforms (Section 2), which defines *both* the algorithm and the dataset, and addresses multiple performance aspects such as raw performance, scalability, and resource utilization. The proposed method, which is equivalent to a benchmarking suite for graph-processing platforms, also includes 5 representative algorithms and 7 representative datasets.

2. We demonstrate how this benchmarking suite can be implemented for six different graph processing platforms (Section 3). We further discuss the requirements to extend this approach for other processing platforms, showing that the feasibility of the extension depends on the usability of the platform (Section 5).

3. We provide a first performance comparison of six graph-processing platforms, emphasizing their strong points and identifying their limitations (Section 4 and 5).

2 Towards Benchmarking 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.

Our method can be seen as a benchmarking suite: it includes similar stages as the definition of a benchmarking suite and has similar goals. Furthermore, we demonstrate in this work how the method introduced in this section can provide a useful skeleton for building proper benchmarks. However, there are still numerous limitations to our method, which we discuss in Section 5.2.

2.1 Performance Aspects, Metrics, Process

To be able to reason about performance behavior, we first need to identify the performance requirements of such platforms, the system parameters to be monitored, the metrics that can be used to characterize platform performance, and an overall process that defines how performance is evaluated. We discuss several important limitations of our design, which prevent it from being a benchmark, in Section 5.2.

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 near-real-time querying, but users are also interested in the the scale and complexity of the graphs a platform can handle, and in platform programmability.

2. Resource use: the ability of a platform to efficiently utilize the resources it has. Ideally, we want platforms to waste as little compute and memory resources as possible, while still preserving their processing power.

3. Scalability: the ability of a platform to maintain its performance behavior when resources are added or removed from its infrastructure. 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 algorithm and dataset.

4. Processing overheads: 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 algorithm and datasets.

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 quantified by extracting useful performance metrics metrics. We summarize in Table 1 the performance metrics used in this work. $\#V$ and $\#E$ are the number of vertices and the number of edges of graphs.

Table 1: Summary of metrics.

Metric	**How measured?**	**Derived**	**Relevant aspect**
Job execution time (T)	Time the full execution	-	Raw processing power Figure 1, 3, and 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 use Figure 5, 6, 7, 8, 9, and 10
Horizontal scalability	T of different cluster size (N)	-	Scalability Figure 11
Vertical scalability	T of different cores per node (C)	-	Scalability Figure 13
Normalized edges per second (NEPS)	-	$\#E/T/N$ or $\#E/T/N/C$	Scalability Figure 12 and 14
Computation time (T_c)	Time actual for calculating	-	Raw processing power Figure 15 and 16
Overhead time (T_o)	-	$T - T_c$	Processing overheads Figure 15 and 16

We define the job execution time as the time from job submission until its completion, so both read and write time are recorded in the job execution time. The job execution time can be further divided into computation time and overhead time. The computation time is 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. We define the performance metric Edges Per Second (EPS) of a platform executing an algorithm as the ratio between the number of edges of the input graph and the job execution time. EPS is the a straightforward extension of the TEPS metric used by

Graph500. To investigate the performance per computing unit, we further define the Normalized Edges Per Second (NEPS) as the ratio between EPS and the total number of computing nodes or cores. By using the number of vertices, we define vertex-centric metrics, Vertices Per Second (VPS) and Normalized Vertices Per Second (NVPS). For the resource use, we monitor the CPU utilization, the memory usage, and the network traffic of platforms.

We select for the evaluation process three types of performance tests: *Load* (*Stress*) tests, which launch an expected (peak) load on the system under test (SUT); *Capacity* tests, which in our method either increase the load by changing the input dataset or keep the load fixed but vary the capacity of the (distributed) system; and *Exploratory* tests, which in our method are similar to the other tests but evaluate the capacity of the system to perform its task without crashing.

2.2 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.

2.2.1 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 [30]: 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 structure (as shown by average node degree and other traditional graph metrics). We also consider in our selection the dataset size (on disk). Since the dataset size can depend on the data format, 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 [31, 32] such as complex plain-text representations, universal data formats (e.g. XML), relational databases, relationship formalisms (e.g., RDF), etc.

Table 2: Summary of datasets.

Graphs	**# V**	**# E**	**d** ($\times 10^{-5}$)	$\bar{\mathbf{D}}$	**Directivity**
Amazon	262,111	1,234,877	1.8	5	directed
WikiTalk	2,388,953	5,018,445	0.1	2	directed
KGS	293,290	16,558,839	38.5	113	undirected
Citation	3,764,117	16,511,742	0.1	4	directed
DotaLeague	61,171	50,870,316	2,719.0	1,663	undirected
Synth	2,394,536	64,152,015	2.2	54	undirected
Friendster	65,608,366	1,806,067,135	0.1	55	undirected

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

We have selected seven graph datasets for this work. Table 2 shows a summary of the characteristics of the selected graphs[1]. The graphs have diverse sources, and a wide range of different size and graph metrics. The synthetic graph ("Synth" in Table 2) is produced by the generator described in Graph500 [33]. The other graphs have been extracted from real-world use, and have been shared through the Stanford Network Analysis Project (SNAP) [34]) and the the Game Trace Archive (GTA) [2]. For the Amazon graph, vertices represent products and two frequently co-purchased products are connected by edges. The WikiTalk graph presents Wikipedia users and their discussion. The KGS and DotaLeague represent players and their playing relationships for two popular games, Go and Defense of the Ancients, respectively; the inclusion of gaming graphs corresponds to the growth of the gaming industry, which now services hundreds of millions of players world-wide. The Citation graph contains the citations between patents granted from 1975 to 1999 in the U.S. Friendster represents records users and their friendships of in a social network. Among the graph characteristics that vary, the disk usage for our graphs ranges from tens of MB to tens of GB.

2.2.2 Algorithm selection

We have conducted a comprehensive survey of graph-processing articles published in 10 representative conferences (namely, CCGrid, CIKM, HPDC, ICDE, IPDPS, PPoPP, SC, SIGKDD, SIGMOD, and VLDB), in recent years; in total, 124 articles. We found that a large variety of graph processing algorithms exist in practice [35]. Table 3 summarizes the survey. Because one article may use multiple algorithms, the total number of algorithms is more than the number of articles. 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, in turn.

Table 3: Survey of graph algorithms.

Class	**Typical algorithms**	**Number**	**Percentage [%]**
General Statistics	Triangulation [36], Diameter [37], BC [38]	24	16.1
Graph Traversal	BFS, DFS, Shortest Path Search	69	46.3
Connected Components	MIS [39], BiCC [40], Reachability [41]	20	13.4
Community Detection	Clustering, Nearest Neighbor Search	8	5.4
Graph Evolution	Forest Fire Model [1], Preferential Attachment Model [42]	6	4.0
Other	Sampling, Partitioning	22	14.8
Total		149	100

The General statistics (STATS) algorithm computes the number of vertices and edges, and the average of the Local Clustering Coefficient (LCC) of all vertices. The results obtained with STATS can inform the graph analyst about how long a more advanced algorithm may take to execute and what type of computer resource

[1]We extract from each raw graph the largest connected component, so that the vertices are reachable to each other in these graphs.

will be utilized the most by the algorithm.

Algorithm 1: Algorithm of STATS.

```
Input: Graph in our defined format
Output: the number of vertices and edges, average LCC
verticesTotal = 0;
edgesTotal = 0;
avgLCC = 0;
foreach Vertex v do
    SendMyOutEdges(EdgesDstList, NeighbourList);
    verticesTotal += 1;
    edgesTotal += getDegree(v);
    VertexNeighbourhood = CreateVertexNeighbourhood([EdgeList]);
    counter = CountEdgesBetweenNeighbours(VertexNeighbourhood);
    avgLCC += CalculateLCC(counter, vertexDegree);
avgLCC= avgLCC / verticesTotal;
```

Breadth-first search (BFS) is a widely used algorithm in graph processing, which is often a building block for more complex algorithm, 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.

Algorithm 2: Algorithm of BFS.

```
Input: Graph in our defined format and source vertex N
Output: Traversed graph
queue = N.getNeighbours();
while queue not empty do
    vertex = queue.dequeue();
    foreach vertex : vertex.getNeighbours() do
        if vertex not visited then
            enqueue vertex onto queue
```

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.

Algorithm 3: Algorithm of CONN.

```
Input: Graph in our defined format
Output: Connected components
foreach vertex do
    vertex.label = vertex.id;
while labelChanged do
    labelChanged = false;
    foreach vertex do
        smallestLabel = vertex.retrieveNeighboursLabel();
        if smallestLabel ¡ vertex.label then
            vertex.label = smallestLabel;
            labelChanged = true;
```

Community detection (CD): community detection is important for social network applications, as users of these networks tends to form communities, that is, groups whose constituent nodes form more relationships group within the group than with nodes outside the group. Communities are also very 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.

Algorithm 4: Algorithm of CD.

```
Input: Graph in our defined format, initial score for all vertices, hop attenuation, limitation of iteration
Output: Detected communities
foreach vertex do
    vertex.label = vertex.id;
while communityLabelChanged do
    labelChanged = false;
    foreach vertex do
        newLabel = choseLabel(receivedLabels);
        if newLabel != currentLabel then
            currentLabel = newLabel;
            labelChanged = true;
        labelScore = updateLabelScore();
        foreach neighbour do
            send(currentLabel, labelScore);
```

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.

Algorithm 5: Algorithm of EVO.

```
Input: Graph in our defined format, number of new vertices, forward (p) and backward (r) burning
       probabilities, limitation of iteration
Output: Graph after evolution
while counter < v do
    ambasador = choseRandomAmbasador();
    createEdge(ambasador);
    x = geometricallyDistributedMean($(1-p)^{-1}$);
    y = geometricallyDistributedMean($(1-rp)^{-1}$);
    createdOutLinks = 0;
    createdInLinks = 0;
    step = 1;
    while createdOutLinks < x do
        createOutLinks(step, ambasador);
        step++;
    step = 1;
    while createdInLinks < y do
        createdInLinks(step, ambasador);
        step++;
    counter++;
```

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 [8], the real-time community detection algorithm proposed by Leung et al. [10], and the the Forest Fire Model for graph evolution designed by Leskovec et al. [1]. Algorithm 1 to 5 show the pseudo code of the five algorithms we selected.

3 Experimental setup

The method introduced in Section 2 defines a benchmarking skeleton. In this section we create a full benchmarking suite (bar the issues explained in Section 5.2) by implementing the graph-processing algorithms of a selected set of test platforms, and by configuring and tuning these platforms.

3.1 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.

We select for this study graph-specific non-distributed platforms, and both graph-specific and generic distributed platforms. Because the resource limitation and relatively little interest in the community, we do not select for this study any generic and non-distributed platform. Table 4 summarizes our selected platforms: Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. We introduce each platform in the following, in turn.

Table 4: 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

Hadoop 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.

We run Hadoop on jdk1.7.0_ (We keep the java version consistent in all the other platforms). We change some configuration setting as follow: the in-memory file-system can use up to 1.5 GB memory for merging map-outputs at the reduces; the maximum number of streams to merge at once when sorting files is set to 80; the memory limitation used for sorting files is 1.5 GB. In section 4.1, we set the maximum value of simultaneously map/reduce tasks on a computing machine as 1, each task JVM process can run with a maximum heap size of 20 GB. In the experiments of vertical scalability in section 4.3, the maximum value of map/reduce will increase to 7 and the maximum heap size per process will decrease to around 3 GB (total amount of 20 GB per machine). For HDFS, we use only one single replica per block without compression because our focus is no fault-tolerance. When loading input files to HDFS, we store each dataset in a number of blocks, which equals to the total number of available slots for map (reduce) tasks we set. For writing output, we set the block size to the graph size over the number of map tasks for the biggest graph Friendster. For the other graphs, we use the default value 64 MB of block size.

YARN 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.

Since the Hadoop configuration can be ported to YARN seamlessly, we keep the settings of YARN same to that of Hadoop. In addition, the maximum allocation for every container request at the resource manager is 20 GB, respectively. The maximum value will be modified during the vertical scalability tests.

Stratosphere 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 [21]. 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.

We install Stratosphere using the HDFS of hadoop-0.20.203.0 as the distributed file system. The configuration of HDFS is identical to that of Hadoop. We run the NameNode service of HDFS on the master node of Stratosphere, and the DateNode services on the Stratosphere worker nodes. For each worker node, the maximum amount of main memory Stratosphere system can use is 20 GB. We select the default network channel for transporting data between nodes. For each worker node, Stratosphere use up to 15,360 network buffers with the default size of 64 KB. We limited the number of concurrent tasks on one work node to 1 in section 4.1. This limitation will increase to 7 in the vertical scalability experiment.

Giraph 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 system crashes due to lack of memory.

We implement the Giraph on top of hadoop-0.20.203.0. For Giraph, we keep all the configurations same to Hadoop. When deploy Giraph on the cluster, we apply one more machine than Hadoop and YARN as the ZooKeeper.

GraphLab 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.

Similar to Stratosphere, we deploy distributed GraphLab on top of the HDFS of hadoop-0.20.203.0 with the same configuration. For distributed running of GraphLab program, we install MPI with the version of mpich2-1.5rc3. Each DataNode service of HDFS run on one MPI node. The number of cores per MPI node is limited to 1 in subsection 4.1. Up to 7 cores per node will be tested in the vertical scalability.

Neo4j 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.

We set the Neo4j on a single machine of DAS4. The java heap size is set to 20 GB. We perform data ingestion into Neo4j database with the use of multiple batch transactions. Our transaction threshold is set as 10,000 vertices or 250,000 edges. The data ingestion process can be shorten by increasing the batch transaction threshold. However, there is a risk of the occurrence of GC overhead error when setting high threshold. The in-depth analysis of the data injection process is out of scope of this paper.

3.2 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. However, the unsophisticated user cannot select appropriate values and tune the task of expects. We use common best-practices for tuning each of the platforms as we discussed in Section 3.1.

Hardware: We deploy the distributed platforms on DAS4 [43], 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 a 10 Gbit/s Infiniband network and 1 Gbit/s Ethernet network. NFS constructed over the Infiniband network 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. For CD, we set the initially a score of 1.0 to vertices. The hop attenuation is set to 0.1. The limitation of iteration is set to 5, because after 5 iterations, even the algorithms is not converge, 95% of vertices are clustered [9, 10]. For EVO, the graph evolute with a growth of 0.1% number of vertices and with 6 iterations. The forward and backward burning probability are set to 0.5. 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 4.4).

4 Experimental results

In this section we present a selection of the experimental results. We evaluate the six graph processing platforms selected in Section 3, using the process and metrics, and the datasets and algorithms introduced in Section 2.

The experiments we have performed are:

- Basic performance (Section 4.1): 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 (Section 4.2): we have investigated the CPU utilization, memory usage, and network traffic. We report them for both the master and computing nodes on the distributed platforms.
- Scalability (Section 4.3): 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 4.4): we have analyzed the execution time in detail, and report important findings related to the platform overhead.

4.1 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 Section 3, 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, 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 [44]. 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.

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 for 10%.
- Several of the platforms are unable to process all datasets for all algorithms, and crash.

4.1.1 Results for one selected algorithm

We present here the results obtained for one selected algorithm, BFS (see Section 2.2.2).

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 5 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 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.

Table 5: Statistics of BFS.

	Amazon	**WikiTalk**	**KGS**	**Citation**	**DotaLeague**	**Synth**	**Friendster**
Coverage [%]	99.9	98.5	100	0.1	100	100	100
Iterations	68	8	9	11	6	8	23

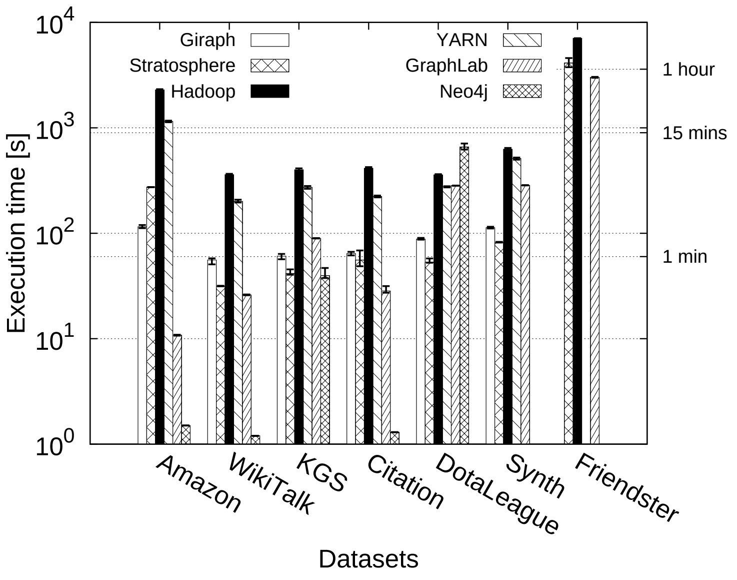

Figure 1: The execution time of algorithm BFS of all datasets of all platforms.

Hadoop always performs worse than the other platforms, mainly because Hadoop has a significant I/O between two continuous iterations (see Section 3). 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.

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 4.4). 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 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.

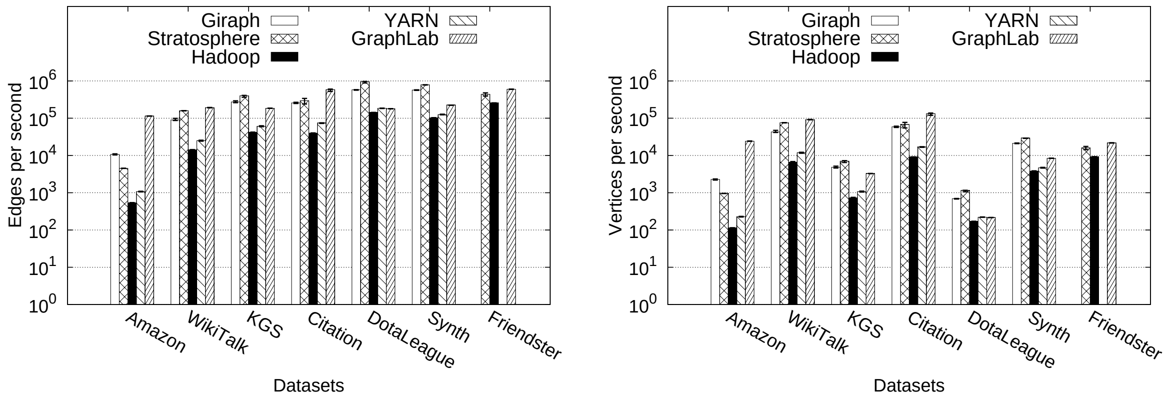

Figure 2: The EPS and VPS of executing BFS.

4.1.2 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.

4.1.3 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 one, DotaLeague, and the small Citation graph. Even for the second-largest

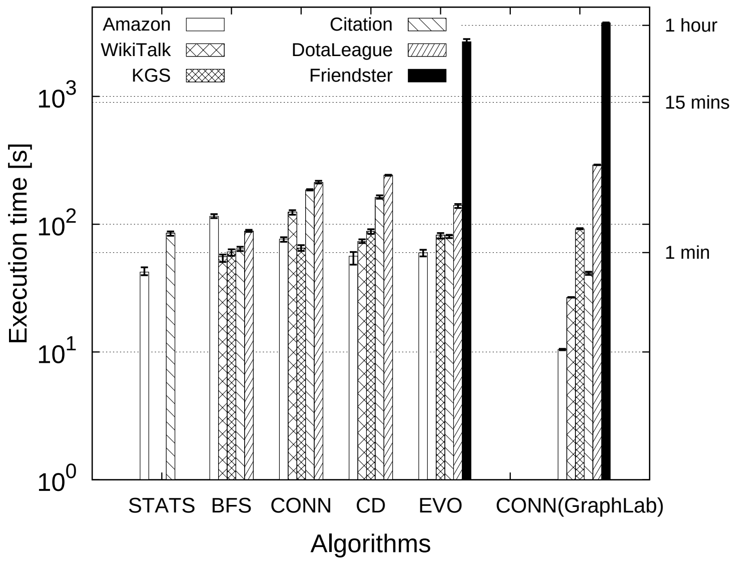

Figure 3: The execution time of all algorithms for all datasets running on Giraph, and for CONN running on GraphLab (right-most bars).

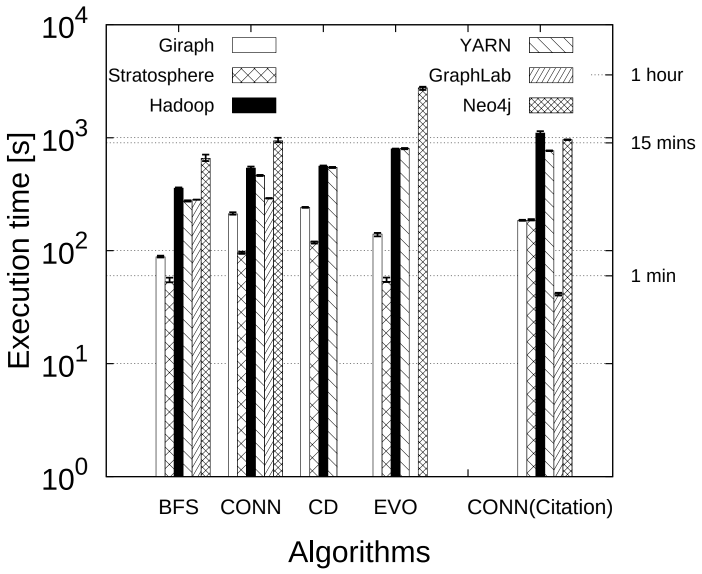

Figure 4: The execution time for all platforms, running all algorithms for the DotaLeague dataset, and CONN for the Citation dataset (right-most bars).

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 4.1.1), more iterations result in higher I/O and other overheads.

4.2 Evaluation of resource usage

To understand the resource usage of each platforms, we investigate in this section the CPU load, memory, and network usage of both the master node and the computing nodes.

For each platform, we execute BFS on DotaLeague. The configuration is consistent to Section 4.1. We monitor the platforms by using the Ganglia Monitoring System [45] with a sampling interval of 1 second. The monitoring results includes the usage of local system such as operating system.

To make the resource usage results comparable, We normalize the execution time of different platforms and, for each platform, of different experiment runs (we use 10 repetitions of each experiment). For each experiment run, we linearly interpolate the real monitoring samples to obtain 100 normalized usage points for each resource. We depict in each figure corresponding to the resource consumption of a computing node the results obtained in practice for a real computing node, such that the depiction is the closest to the average resource consumption observed in practice.

Key findings:

- Few resources are needed for the master node of all platforms.
- The resource usage of the computing nodes varies widely across different platforms.

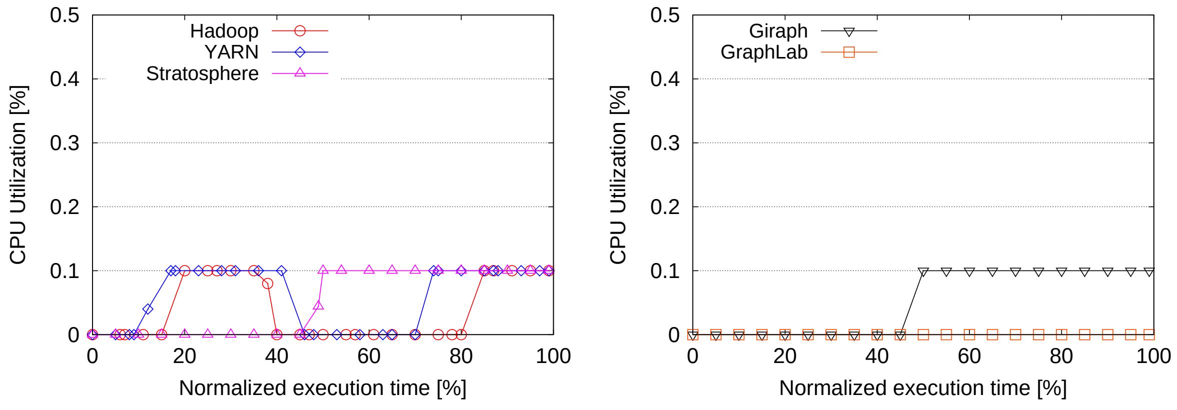

Figure 5: The CPU utilization of the master node.

Possibly because there is only one job submitted to all platforms, the master does not heavily use resources. As shown in Figure 5 and 7, the CPU utilization and the network traffic have low usage for job management and platform operation (heartbeats, etc.). For all platforms, the CPU utilization is below 0.5% and the network traffic is less than 400 Kbit/s (the only exception is Stratosphere, which sometimes can reach up to 1 Mbit/s).

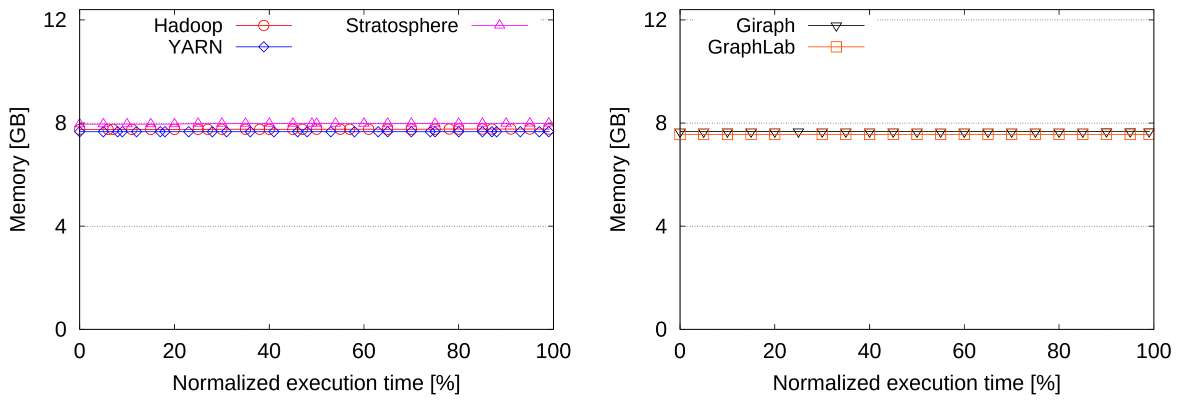

Figure 6: The memory usage of the master node.

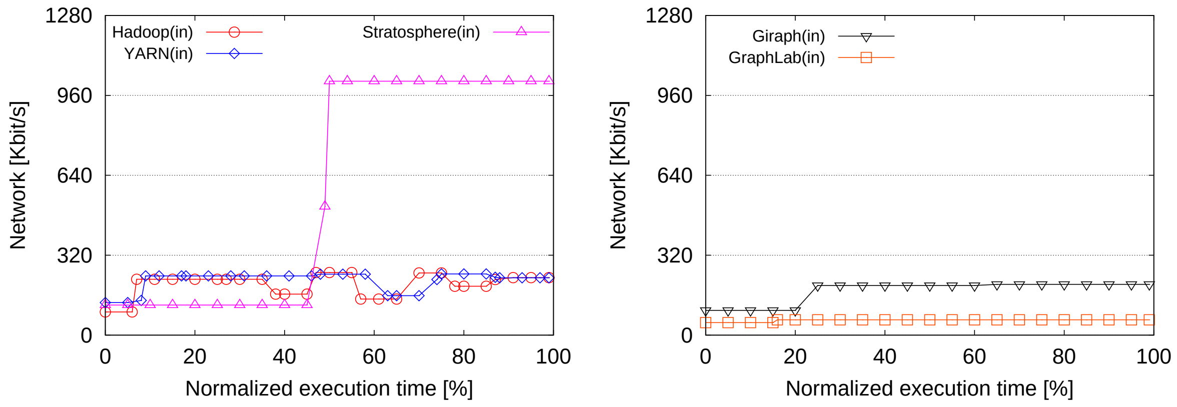

Figure 7: The network traffic of the master node.

From Figure 6, the monitored memory usage of all platforms is around 8 GB, including the memory consumption of operating systems and services such as HDFS. By deploying the Stratosphere and GraphLab on the same cluster, and by observing that GraphLab does not need a manager, we obtain the memory usage for Stratosphere: around 400 MB from the 8 GB used in that system. Similarly, we obtain the memory usage of other platforms, and find it to be around 200 MB.

For computing nodes, Figures 8, 9, and 10 depict the CPU utilization, the memory usage, and the network traffic, respectively. The resource usage of computing nodes in YARN and Hadoop exhibit obvious volatility, due to the BFS job consisting of 6 independent iterations. However, the curves do not actually exhibit 6 usage spikes—the computing node with the resource consumption closest to the average is not used intensively in each of the 6 iterations. The memory usage of Stratosphere keeps around 20 GB, as configured in Section 3. This is because Stratosphere compute nodes allocate the memory assigned by the configuration immediately after startup. This design may decrease the possibility of resource sharing between Stratosphere and other applications. Moreover, by using the network channel for transporting data, Stratosphere exhibits the heaviest network throughput. Compared to the generic platforms, the resource usage of Giraph and GraphLab are much smaller. As we discussed in Section 4.1.1, the reason is the graph-friendly programming model of Giraph and GraphLab: these platforms only process activated vertices in each iteration, which reduces the resource

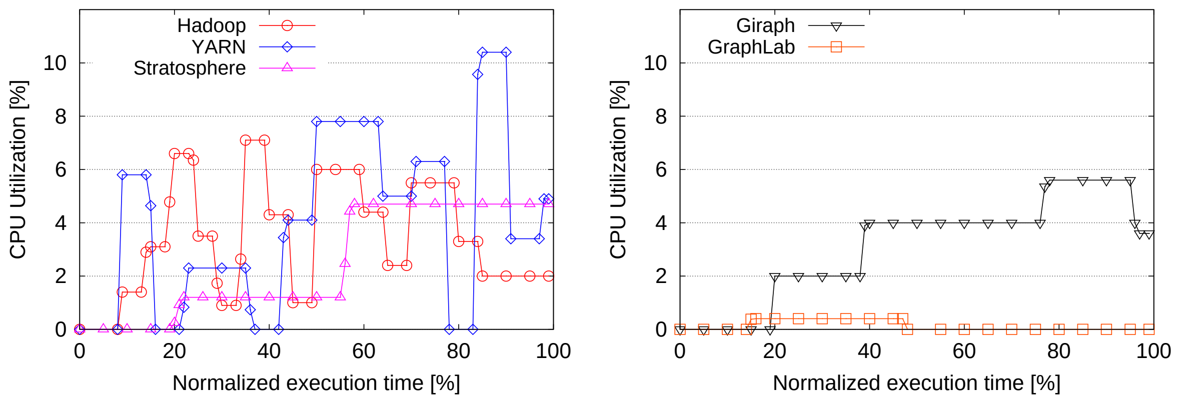

Figure 8: The CPU utilization of a computing node.

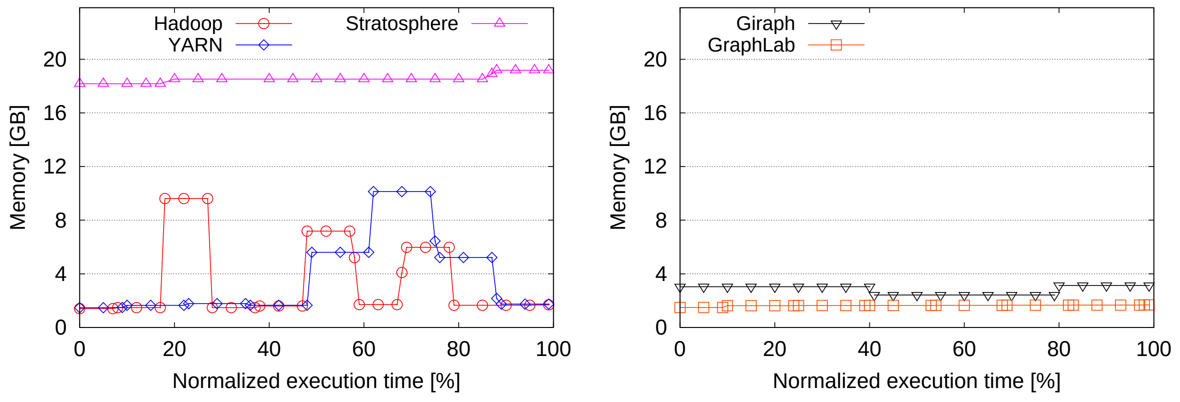

Figure 9: The memory usage of a computing node.

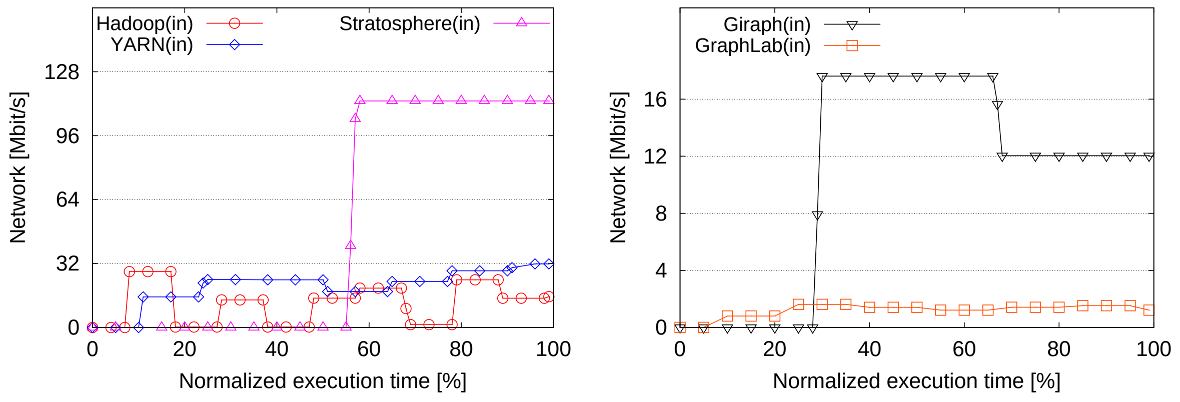

Figure 10: The network traffic of a computing node. Note that the scales of vertical axes are different.

requirement.

4.3 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 test using the two largest real graphs in our study, Friendster and DotaLeague. 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.

4.3.1 Horizontal scalability

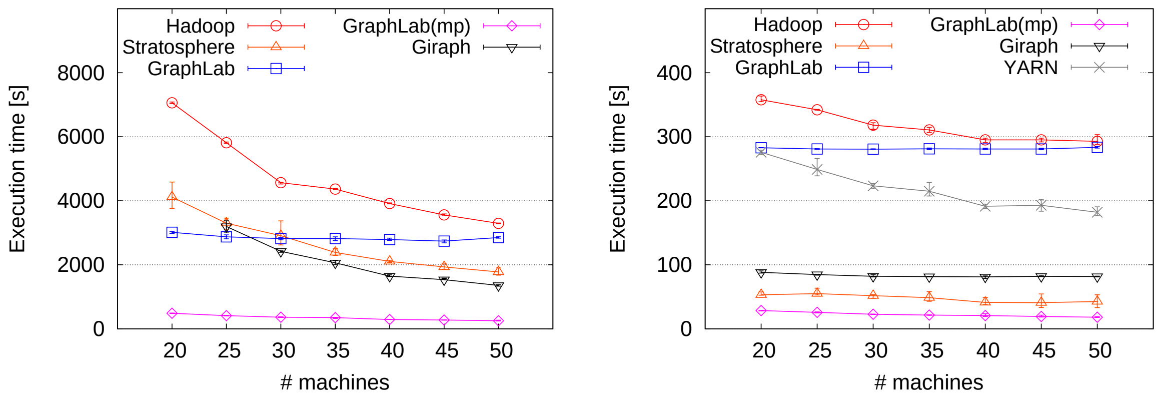

Figure 11: The horizontal scalability of processing Friendster (left) and DotaLeague (right).

Figure 11 shows the horizontal scalability of BFS for Friendster and DotaLeague. Most of the platforms presents significant horizontal scalability only for Friendster, except for GraphLab, which exhibits little scalability for both datasets. The horizontal scalability of GraphLab is constrained by the graph loading phase using one single file. We thus explore tuning GraophLab: 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, for both datasets. Moreover, 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. Figure 12 depicts the NEPS of all platforms on processing Friendster and DotaLeague. The maximum value of NEPS can be reached at different sizes of the cluster, for different platforms. For example,

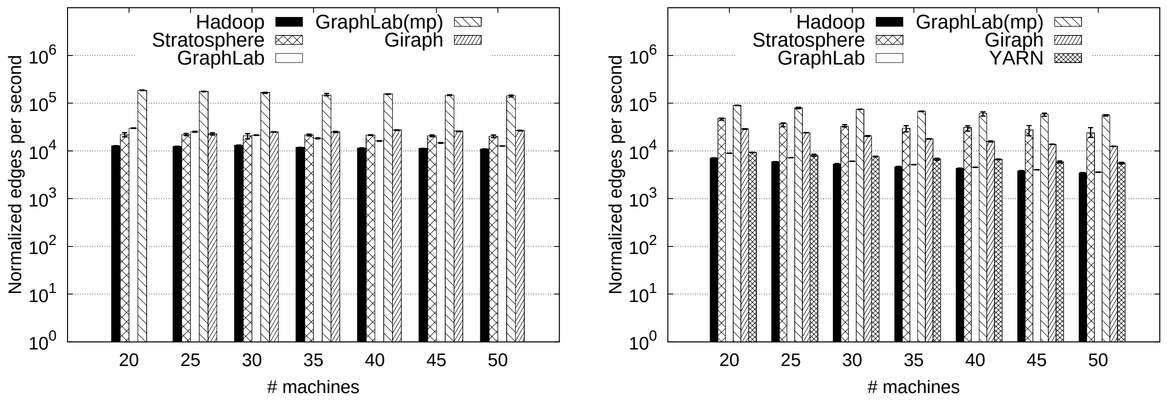

Figure 12: The NEPS of Friendster (left) and DotaLeague (right) in horizontal scalability.

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.

4.3.2 Vertical scalability

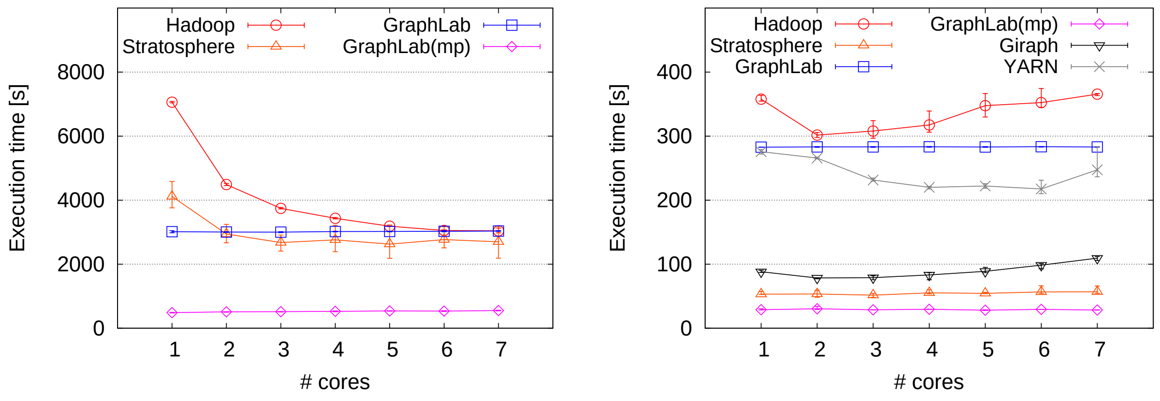

Figure 13: The vertical scalability of processing Friendster (left) and DotaLeague (right). Note that the scales of vertical axes are different.

Figure 13 shows the vertical scalability of running BFS for the Friendster and DotaLeague datasets. 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 tht 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 have not observed significant vertical scalability for the smaller DotaLeague dataset.

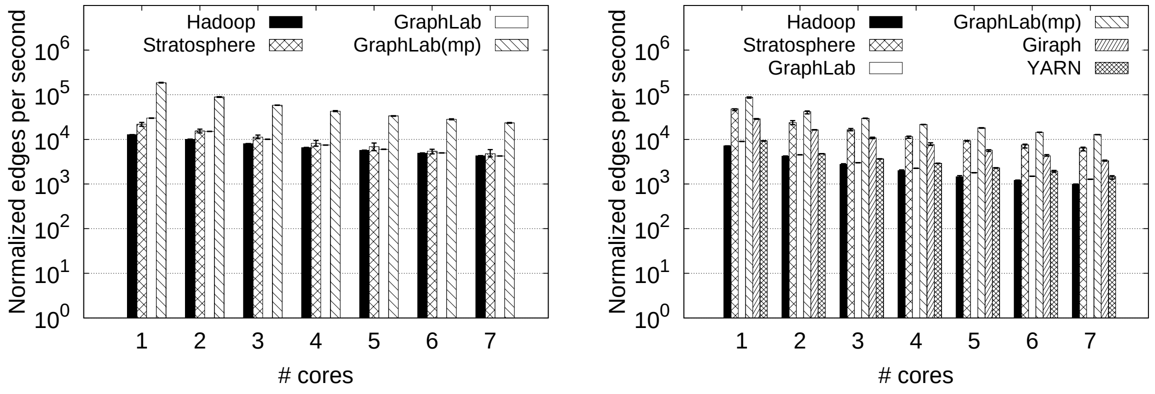

Figure 14: The NEPS of Friendster(left) and DotaLeague(right) in vertical scalability.

We check the performance per computing unit (computing core) by NEPS in vertical scalability. As shown in Figure 14, we get similar results to that of horizontal scalability, all NEPS drops for all platforms.

4.4 Evaluation of performance overhead

In this section, we evaluate two elements of performance 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 in this study.

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 6 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 can find that the fraction of time spent with overheads varies across the platforms and datasets from Figure 15 and 16. The job execution time for CONN on Friendster of GraphLab is more than one hour, exceeding the scale of Figure 16. 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.

Table 6: Data ingestion time.

	Amazon	WikiTalk	KGS	Citation	DotaLeague	Synth	Friendster
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

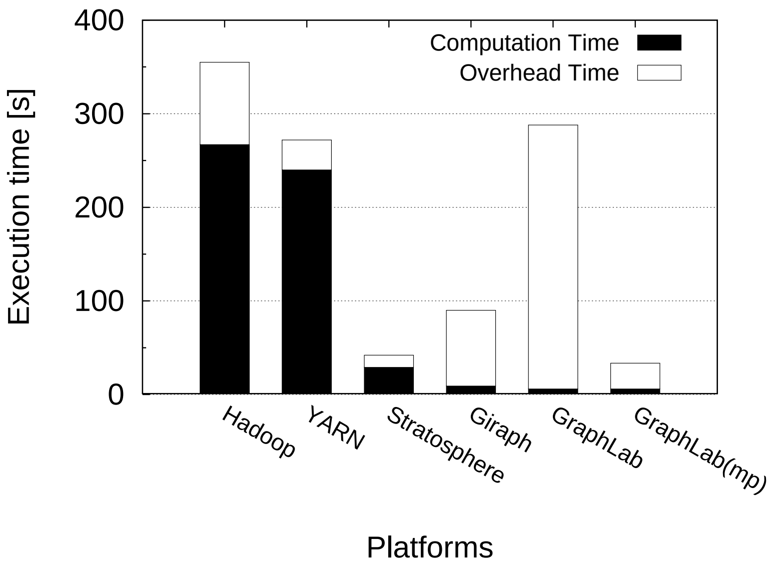

Figure 15: The execution time breakdown of platforms for BFS on DotaLeague.

5 Discussion

5.1 Our experience as platform users

Although performance is a key selection criterion for a graph-processing platforms, usability may also be important. In this section, we discuss our experience as platform users. We have documented during development the time it took to develop each algorithm on each platform; Table 7 summarizes the development time and the lines of code used to implement core part of algorithms (that is, without the code developed to read and write the graph, to parse parameters, etc. In our experience, YARN and Hadoop require similar development effort, which contrasts to the platforms that already implement our selected algorithms, such as BFS in Neo4j and CONN in GraphLab.

Although the number lines of core code in our implementation is relatively small, in our experience the platforms needed sufficiently long time to understand the programming models of different platforms and to make the algorithm execute correctly. The non-core code can also be non-trivial. As most of the graph-processing algorithms are iterative, for YARN and Hadoop a driver program is needed to run same Map and Reduce tasks every iteration; in contrast, Stratosphere supports more complex second-order functions and is thus more applicable for graph processing. Last, different platforms offer different programming trade-offs. We found the vertex-centric programming models of Neo4j, GraphLab, and Giraph as facile to learn and reducing the development effort, when compared to the other platforms. GraphLab programs need to be written in C++, in contrast to the Java code required by the others. Regarding the data format, only GraphLab used a directed graph data-structure, which means users have to create two edges (in and out) for undirected edges.

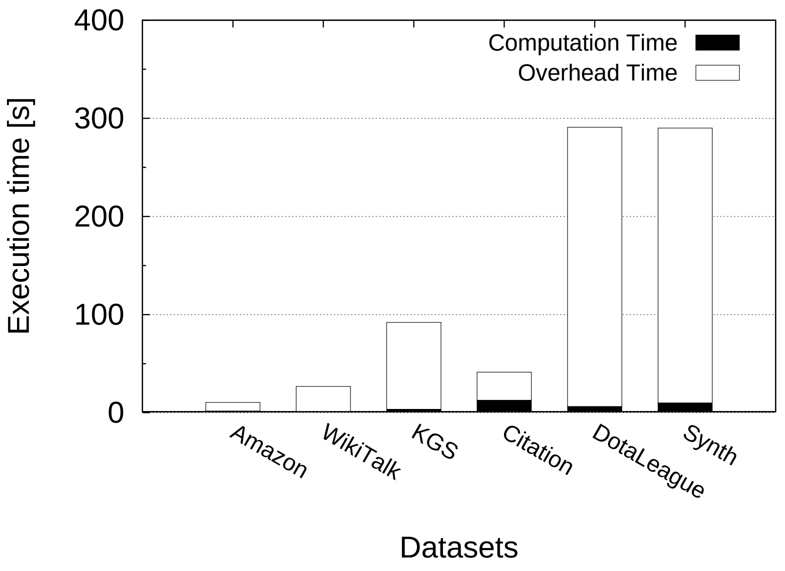

Figure 16: The execution time breakdown of GraphLab for CONN on datasets.

Table 7: Example of development time and lines of core code. Legend: d–day, h–hour, loc–lines of code.

	Hadoop(Java)	**Stratosphere(Java)**	**Giraph(Java)**	**GraphLab(C++)**	**Neo4j(Java)**
BFS	1 d, 110 loc	1 d, 150 loc	1 d, 45 loc	1 d, 120 loc	1 h, 38 loc
CONN	1.5 d, 110 loc	1 d, 160 loc	1 d, 80 loc	0.5 d, 130 loc	1 d, 100 loc

5.2 Towards a Benchmark

The method proposed in Ssection 2 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[2].

Methodologically, our method has limitations in its process, workload design, and metrics design. 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 (would 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. 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). These limitations also affect all other benchmarks and performance evaluation studies included in the related work of this study.

From a practical perspective, our method has limitations in portability, time, and cost. The portability

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

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 Section 5.1. 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 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.

6 Related Work

Table 8: Overview of performance evaluation and comparison of graph-processing on different platforms. Legend: V–vertices, E–edges, C–computers.

Platforms	**Algorithms**	**Dataset type**	**Largest dataset**	**System**
Neo4j, MySQL [46]	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 [47]	CONN, 3 others	real	39 MV, 1.5 BE	60 C
Trinity, etc. [27]	BFS, 2 others	synthetic	1 BV	16 C
PEGASUS [25]	CONN,2 others	synthetic, real	282 MV	90 C
CGMgraph [48]	CONN, 4 others	synthetic	10 MV	30 C
PBGL, CGMgraph [49]	CONN, 3 others	synthetic	70 MV, 1 BE	128 C
Hadoop, PEGASUS [50]	1 other	synthetic, real	1 BV, 20 BE	32 C
HaLoop, Hadoop [23]	2 others	synthetic, real	1.4 BV, 1.6 BE	90 C
Our method	5 classes	synthetic, real	66 MV, 1.8 BE	50 C

Many previous studies focus on performance evaluation of graph-processing, for different platforms. Table 8 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 many types of metrics. Finally, previous research compares few platforms; in contrast, we investigate 6 popular graph-processing platforms with different architectures.

Except for graph processing, as the de-facto platforms for large-scale date processing, Hadoop has often been compared with other systems. Pavlo et al. [51] note that Hadoop loads input data faster than two parallel DBMSs, DBMS-X and Vertica, but it is outperformed in running real tasks. In an in-depth study, Jiang et al. [52] report that the performance of Hadoop can be fine-tuned to approach that of Parallel DBMS. Similar to Pavlo's result, from the investigation of Chen and Hsu [53], Hadoop performs worse than Vertica on extracting information from large-scale text. Ouaknine and Kirkpatrick [54] conduct a performance comparison between Hadoop and another large data platform HPCC Systems, when processing ECL queries; their experiments reveal that Hadoop runs nearly 2 times slower than the HPCC Systems, on average.

7 Conclusion and Future Work

An quickly increasing number data-intensive platforms can process large-scale graphs, and have thus become potentially interesting for a variety of users and application domains. To compare in-depth the performance of graph-processing platforms, and thus facilitate platform selection and tuning, we have proposed in this work an empirical method and applied it to a comprehensive performance study of six graph-processing 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, scalability, resource consumption, and performance overhead. We use both performance and throughput metrics, and also use normalized metrics to characterize scalability. We select five typical 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, and size of up to 1.8 billion edges and tens of GB of stored data.

Using our method, 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.

For the future, we plan to extend our work by enhancing our method towards a true benchmark and building an empirically validated performance-boundary model for predicting the worst performance of these platforms.

References

[1] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005. 4, 8, 11

[2] Yong Guo and Alexandru Iosup. The Game Trace Archive. In *NetGames*, 2012. 4, 8

[3] Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F Siegel. Discovering Regulatory and Signalling Circuits in Molecular Interaction Networks. *Bioinformatics*, 2002. 4

[4] D Dominguez-Sal, P Urbón-Bayes, A Giménez-Vanó, S Gómez-Villamor, N Martínez-Bazán, and JL Larriba-Pey. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. *Web-Age Information Management*, pages 37–48, 2010. 4, 23, 26

[5] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010. 4, 26

[6] Duncan J Watts and Steven H Strogatz. Collective Dynamics of 'Small-World' Networks. *nature*, 393(6684):440–442, 1998. 4

[7] David A Bader and Kamesh Madduri. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *International Conference on Parallel Processing*, pages 523–530. IEEE, 2006. 4

[8] Bin Wu and YaHong Du. Cloud-Based Connected Component Algorithm. In *International Conference on Artificial Intelligence and Computational Intelligence*, pages 122–126. IEEE, 2010. 4, 11

[9] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 2007. 4, 13

[10] Ian XY Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. Towards Real-Time Community Detection in Large Networks. *Physical Review E*, 2009. 4, 11, 13

[11] Tanja Falkowski, Anja Barth, and Myra Spiliopoulou. Dengraph: A Density-Based Community Detection Algorithm. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 112–115, 2007. 4

[12] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1999. 4

[13] Mark Redekopp, Yogesh Simmhan, and Viktor K Prasanna. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In *27th IEEE International Parallel Distributed Processing Symposium*. IEEE, 2013. 4

[14] Jure Leskovec and Christos Faloutsos. Sampling from Large Graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636. ACM, 2006. 4

[15] Michael Capobianco and Ove Frank. Graph Evolution by Stochastic Additions of Points and Lines. *Discrete Mathematics*, 1983. 4

[16] Neo4j. `http://www.neo4j.org/`. 4

[17] Borislav Iordanov. HyperGraphDB: a Generalized Graph Database. *Web-Age Information Management*, pages 25–36, 2010. 4

[18] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th conference on Symposium on Opearting Systems Design & Implementation*, 2012. 4

[19] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012. 4

[20] YARN. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`. 4

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 2007. 4, 12

[22] Daniel Warneke and Odej Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*, page 8. ACM, 2009. 4

[23] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, pages 285–296, 2010. 4, 26

[24] Giraph. `http://giraph.apache.org/`. 4

[25] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM*, pages 229–238. IEEE, 2009. 4, 26

[26] 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 *Proceedings of the VLDB Endowment*, pages 716–727, 2012. 4

[27] Bin Shao, Haixun Wang, and Yatao Li. The Trinity Graph Engine. Technical report, Technical Report 161291, Microsoft Research, 2012. 4, 26

[28] Toyotaro Suzumura, Koji Ueno, Hitoshi Sato, Katsuki Fujisawa, and Satoshi Matsuoka. Performance Characteristics of Graph500 on Large-Scale Distributed Environment. In *IEEE International Symposium on Workload Characterization,*, pages 149–158, 2011. 4

[29] Koji Ueno and Toyotaro Suzumura. Highly Scalable Graph Search for the Graph500 Benchmark. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2012. 4

[30] Robin J Wilson. *Introduction to Graph Theory*. Academic Press New York, 1972. 7

[31] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML). *World Wide Web Journal*, 1997. 7

[32] Ora Lassila. With the Resource Description Framework. 1999. 7

[33] Graph500, April 2013. `http://www.graph500.org/`. 8

[34] SNAP. `http://snap.stanford.edu/index.html`. 8

[35] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), 2006. 8

[36] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. On Triangulation-based Dense Neighborhood Graphs Discovery. *VLDB*, 2010. 8

[37] Wei Jiang and Gagan Agrawal. Ex-MATE: Data Intensive Computing with Large Reduction Objects and Its Application to Graph Mining. In *CCGRID*, 2011. 8

[38] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP*, 2013. 8

[39] Aydin Buluç, Erika Duriakova, Armando Fox, John R. Gilbert, Shoaib Kamil, Adam Lugowski, Leonid Oliker, and Samuel Williams. High-Productivity and High-Performance Analysis of Filtered Semantic Graphs. In *IPDPS*, 2013. 8

[40] Guojing Cong and Konstantin Makarychev. Optimizing Large-scale Graph Analysis on Multithreaded, Multicore Platforms. In *IPDPS*, 2012. 8

[41] Jing Cai and Chung Keung Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, 2010. 8

[42] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. 1999. 8

[43] DAS4. `http://www.cs.vu.nl/das4/`. 13

[44] Bogdan Ghit, Nezih Yigitbasi, and Dick Epema. Resource Management for Dynamic MapReduce Clusters in Multicluster Systems. In *Proceedings of the 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) co-located with Supercomputing (SC)*. IEEE, 2012. 14

[45] Ganglia Monitoring System. `http://ganglia.sourceforge.net/`. 18

[46] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010. 26

[47] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. Technical report, 2012. 26

[48] Albert Chan, Frank Dehne, and Ryan Taylor. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *International Journal of High Performance Computing Applications*, 19(1):81–97, 2005. 26

[49] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005. 26

[50] Karthik Kambatla, Georgios Kollias, and Ananth Grama. Efficient Large-Scale Graph Analysis in MapReduce. In *Seventh International Workshop on Parallel Matrix Algorithms and Applications (PMAA)*, 2012. 26

[51] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009. 26

[52] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The Performance of MapReduce: An In-depth Study. In *Proceedings of the VLDB Endowment*, pages 472–483, 2010. 26

[53] Fei Chen and Meichun Hsu. A Performance Comparison of Parallel DBMSs and MapReduce on Large-Scale Text Analytics. In *Proceedings of 16th International Conference on Extending Database Technology*, EDBT, pages 613–624, 2013. 26

[54] Keren Ouaknine and Scott Kirkpatrick. Architecture and Performance Comparison of Hadoop and HPCC Systems. `http://www.kereno.com/HvH.pdf`. 26