Towards Benchmarking IaaS and PaaS Clouds for Graph Analytics

Alexandru Iosup,
Mihai Capotă,
Tim Hegeman,
Yong Guo,
Wing Lung Ngai
Delft University of Technology
Delft, the Netherlands

Ana Lucia Varbanescu,
Merijn Verstraaten
University of Amsterdam
Amsterdam, the Netherlands

Contact info:
A.Iosup@tudelft.nl
A.L.Varbanescu@uva.nl

ABSTRACT

Cloud computing is a new paradigm for using ICT services—only when needed and for as long as needed, and paying only for service actually consumed. Benchmarking the increasingly many cloud services is crucial for market growth and perceived fairness, and for service design and tuning. In this work, we propose a generic architecture for benchmarking cloud services. Motivated by recent demand for data-intensive ICT services, and in particular by processing of large graphs, we adapt the generic architecture to Graphalytics, a benchmark for distributed and GPU-based graph analytics platforms. Graphalytics focuses on the dependence of performance on the input dataset, on the analytics algorithm, and on the provisioned infrastructure. The benchmark provides components for platform configuration, deployment, and monitoring, and has been tested for a variety of platforms. We also propose a new challenge for the process of benchmarking data-intensive services, namely the inclusion of the data-processing algorithm in the system under test; this increases significantly the relevance of benchmarking results, albeit, at the cost of increased benchmarking duration.

1. INTRODUCTION

Cloud services are an important branch of commercial ICT services. Cloud users can provision from Infrastructure-as-a-Service (IaaS) clouds "processing, storage, networks, and other fundamental resources" [43] and from Platform-as-a-Service (PaaS) clouds "programming languages, libraries, [programmable] services, and tools supported by the provider" [43]. These services are provisioned on-demand, that is, when needed, and used for as long as needed and paid only to the extent to which they are actually used. For the past five years, commercial cloud services provided by Amazon, Microsoft, Google, etc., have gained an increasing user base, which includes small and medium businesses [5], scientific HPC users [16, 35], and many others. Convenient and in some cases cheap cloud services have enabled many new ICT applications. As the market is growing and diversifying, benchmarking and comparing cloud services, especially from commercial cloud providers, is becoming increasingly more important. In this work, we study the process of benchmarking IaaS and PaaS clouds and, among the services provided by such clouds for new ICT applications, we focus on graph analytics, that is, the processing of large amounts of linked data.

Benchmarking is a traditional approach to verify that the performance of a system meets the requirements. When benchmarking results are published, for example through mixed consumer-provider organizations such as SPEC and TPC, the consumers can easily compare products and put pressure on the providers to use best-practices and perhaps lower costs. For clouds, the new use for benchmarking results is to convince customers about the performance, the elasticity, the stability, and the resilience of offered services, and thus to convince customers to rely on cloud services for the operation of their businesses.

Because of its many uses, benchmarking has been the focus of decades of scientific and practical studies. There are many successful efforts on benchmarking middleware [8, 7], on benchmarking databases [24], on the performance evaluation of grid and parallel-system schedulers [17, 12, 20, 31], and on benchmarking systems in general [36, 4].

To benchmark IaaS and PaaS clouds, older benchmarking techniques need to be adapted and extended. As an example of adaptation, cloud benchmarks need to adapt traditional techniques to the new cloud-workloads. We conjecture that the probable characteristics of current and near-future workloads can be derived from three major trends emerging from the last decade of grid and large-scale computing. First, individual jobs are now predominantly split into smaller compute or data-intensive tasks (many tasks [51]); there are almost no tightly coupled parallel jobs. Second, the duration of individual tasks is diminishing with every year; few tasks are still running for longer than one hour and a majority require only a few minutes to complete. Third, compute-intensive jobs are split either into bags-of-tasks (BoTs) or DAG-based workflows, but data-intensive jobs may use a variety of programming models, from MapReduce to general dataflow.

As an example of extension, cloud benchmarks need to extend traditional techniques to accommodate the new application domains supported by clouds. Currently, the use of clouds is fragmented across many different application areas, including hosting applications, media, games, and web sites, E-commerce, On-Demand Workforce and CRM, high-performance computing, search, and raw resources for various usage. Each application area has its own (de facto) performance standards that have to be met by commercial clouds, and some have even developed benchmarks (e.g., BioBench [3] for Bioinformatics and RUBiS [57] for online business). More importantly, many of these applications have rely upon unique, deep, and distributed software stacks, which pose many unresolved challenges to traditional benchmarking approaches—even the definition of the system under test becomes complicated.

We discuss in this article a generic approach to IaaS and PaaS cloud benchmarking. we propose a generic architecture for IaaS and PaaS cloud benchmarking. We have designed the architecture so that it is already familiar to existing practitioners, yet provide new, cloud-specific functionality. For example, current IaaS cloud operators lease resources to their customers, but leave the selection of resource types and the selection of the lease/release moments as a customer task; because such selection can impact significantly the performance of the system built to use the leased resources, the generic benchmarking architecture must include policies for provisioning and allocation of resources. Similarly, the current platforms may require the configuration of deep stacks of software (middleware), so the generic benchmarking architecture much include policies for advanced platform configuration and management.

Although we have designed the architecture to be generic, we have not yet proven that it can be used to benchmark the vast diversity of existing cloud usage scenarios. Authors of this article have already used it, in practice, to benchmark a variety of IaaS cloud usage scenarios [32, 61, 59]. Motivated by the increasingly important fraction of application data, in some cases over three-quarters (IHS and Cisco studies in April 2014) or even higher (IDC Health Insights report in December 2014), that is already or will soon reside in the cloud, we propose in this work an application of the generic architecture for *data-intensive* applications, in the graph-processing application domain. We focus on *graph analytics*, which is a data-intensive process that is increasingly used to provide insight into social networks, personalized health-care, natural language processing, and many other fields of science, engineering, retail, and e-government. As a consequence, several well established graph analytics platforms, such as GraphLab, Giraph, and GraphX, are competing with many graph analytics platforms that are currently in operation or under development. Due to the high diversity of workloads encountered in practice, both in terms of algorithm and of dataset, and to the influence of these workloads on performance, it is increasingly difficult for users to identify the graph analytics platform best suited for their needs. We propose in this work Graphalytics, a benchmarking approach that focuses on graph analytics and is derived from the generic architecture. Graphalytics focuses on understanding how the performance of graph analytics platforms depends on the input dataset, on the analytics algorithm, and on the provisioned infrastructure. It provides components for platform configuration, deployment, and monitoring. It also manages the benchmarking process, from configuration to reporting.

Last, starting from the experience we have accumulated designing and using Graphalytics in practice, we identify an important new challenge for benchmarking in clouds: the need to understand the platform-specific and data-dependent performance of algorithms used for graph analytics. For clouds, it is currently not possible for customers to have access to the low-level technical specifications of the service design, deloyment, and tuning, and thus also not possible to easily select a particular algorithm to implement and deploy on the provisioned platform. Instead, either the cloud customers or the cloud operators need to determine an appropriate algorithm, for example through benchmarking. Thus, for clouds we see the need to include the algorithm in the system under test. We show preliminary results that give strong evidence for this need, and discuss steps towards algorithm-aware benchmarking processes.

This article has evolved from several regular articles [19, 27, 25], a book chapter [33], and a series of invited talks given by the authors between 2012 and 2014, including talks at MTAGS 2012 [34], HotTopiCS 2013 [29], etc[1]. This work has also benefited from valuable discussion in the SPEC Research Group's Cloud Working Group and Big Data Working Group. The new material in this article focuses on the application of the general architecture for IaaS cloud benchmarking to graph analytics, and on a new challenge for benchmarking big data processes, related to the inclusion of processing-algorithm alternatives in the evaluation process.

The remainder of this article is structured as follows. In Section 2, we present a primer on benchmarking computer systems. Then, we introduce a generic approach for IaaS cloud benchmarking, in Section 3. We apply the generic architecture to graph analytics, and propose the resulting benchmarking framework Graphalytics, in Section 4. We introduce a new challenge for cloud benchmarking, related to the inclusion of the algorithm in the system under test, in Section 5. Last, we conclude in Section 6.

2. A PRIMER ON BENCHMARKING COMPUTER SYSTEMS

We review in this section the main elements of the typical benchmarking process, which are basically unchanged since

[1]In inverse chronological order: Lecture at the Fifth Workshop on Big Data Benchmarking (WBDB), Potsdam, Germany, August 2014. Lecture at the Linked Data Benchmark Council's Fourth TUC Meeting 2014, Amsterdam, May 2014. Lecture at Intel, Haifa, Israel, June 2013. Lecture at IBM Research Labs, Haifa, Israel, May 2013. Lecture at IBM T.J. Watson, Yorktown Heights, NY, USA, May 2013. Lecture at Technion, Haifa, Israel, May 2013. Online lecture for the SPEC Research Group, 2012.

the early 1990s. For more detail, we refer to canonical texts on benchmarking [24] and performance evaluation [36] of computer systems. We also discuss the main reasons for benchmarking.

2.1 Elements of Benchmarking

Inspired by canonical texts [24, 36], we review here the main elements of a benchmarking process. The main requirements of a benchmark—relevance, portability, scalability, and simplicity—have been discussed extensively in related literature, for example in [24, Ch.1].

The *System Under Test (SUT)* is the system that is being evaluated. A *white box* system exposes its full operation, whereas a *black box* system does not expose operational details and is evaluated only through its outputs.

The *workload* is the operational load to which the SUT is subjected. Starting from the empirical observation that "20% of the code consumes 80% of the resources", simple *microbenchmarks* (*kernel benchmarks* [24, Ch.9]) are simplified or reduced-size codes designed to stress potential system bottlenecks. Using the methodology of Saavedra et al. [53] and later refinements such as Sharkawi et al. [55], the results of microbenchmarks can be combined with application profiles to provide credible performance predictions for any platform. *Synthetic* and even *real-world (complex) applications* are also used for benchmarking purposes, as a response to system improvements that make microbenchmarks run fast but do not affect the performance of much larger codes. For distributed and large-scale systems such as IaaS clouds, *simple workloads* comprised of a single application and a (realistic) job arrival process represent better the typical system load and have been used for benchmarking [30]. *Complex workloads*, that is, the combined simple workloads of multiple users, possibly with different applications and job characteristics, have started to be used in the evaluation of distributed systems [30, 59]; we see an important role for them in benchmarking.

The *Benchmarking Process* consists of the set of rules, prior knowledge (invariants), and procedures used to subject the SUT to the benchmark workload, and to collect and report the results.

2.2 Why Benchmarking?

Benchmarking computer systems is the process of evaluating their performance and other non-functional characteristics with the purpose of comparing them with other systems or with industry-agreed standards. Traditionally, the main use of benchmarking has been to facilitate the informed procurement of computer systems through the publication of verifiable results by system vendors and third-parties. However, benchmarking has grown as a support process for several other situations, which we review in the following.

Use in system design, tuning, and operation: Benchmarking has been shown to increase pressure on vendors to design better systems, as has been for example the experience of the TPC-D benchmark [24, Ch.3, Sec.IV]. For this benchmark, insisting on the use of SQL has driven the wide acceptance of the ANSI SQL-92; furthermore, the complexity of a majority of the queries has lead to the stress of various system bottlenecks, and ultimately to numerous improvements in the design of aggregate functions and support for them. This benchmark also led to a wide adoption of the geometric mean for aggregating normalized results [4]. The tuning of the DAS multi-cluster system has benefited from the benchmarking activity of some of the authors of this chapter, developed in the mid-2000s [30]; then, our distributed computing benchmarks exposed various (fixable) problems of the in-operation system.

Use in system procurement: Benchmarks such as HPCC, the SPEC suites, the TPC benchmarks, etc. have long been used in system procurement—the systems tested with these benchmarks can be readily compared, so a procurement decision can be informed. Benchmarks can be also very useful tools for system procurement, even when the customer's workloads are not ideally matched by the workloads represented in the benchmark, or when the representativeness of the benchmark for an application domain can be questioned. In such situation, the customer gains trust in the operation of the system, rather than focus on the actual results.

Use in training: One of the important impediments in the adoption of a new technology is the lack of expertise of potential users. Market shortages of qualified personnel in computer science are a major cause of concern for the European Union and the US. Benchmarks, through their open-source nature and representation of industry-accepted standards, can represent best-practices and thus be valuable training material.

On alternatives to benchmarking: Several alternative methods have been used for the purposes described earlier in this section, among them empirical performance evaluation, simulation, and even mathematical analysis. We view benchmarking as an empirical evaluation of performance that follows a set of accepted procedures and best-practices. Thus, the use of empirical performance evaluation is valuable, but perhaps without the representativeness of a (de facto) standard benchmark. We see a role for (statistical) simulation [47, 18, 21] and mathematical analysis when the behavior of the system is well-understood and for long-running evaluations that would be impractical otherwise. However, simulating new technology, such as cloud computing, requires careful (and time-consuming) validation of assumptions and models.

3. A GENERIC ARCHITECTURE FOR IAAS AND PAAS CLOUD BENCHMARKING

We propose in this section a generic architecture for IaaS and PaaS cloud benchmarking. Our architecture focuses on conducting benchmarks as sets of (real-world) experiments that lead to results with high statistical confidence, on considering and evaluating IaaS clouds as evolving black-box systems, on employing complex workloads that represent multi-tenancy scenarios, on domain-specific scenarios, and on a combination of traditional and cloud-specific metrics.

3.1 Overview

Our main design principle is to adapt the proven designs for benchmarking to IaaS clouds. Thus, we design an architecture starting from our generic architecture for IaaS

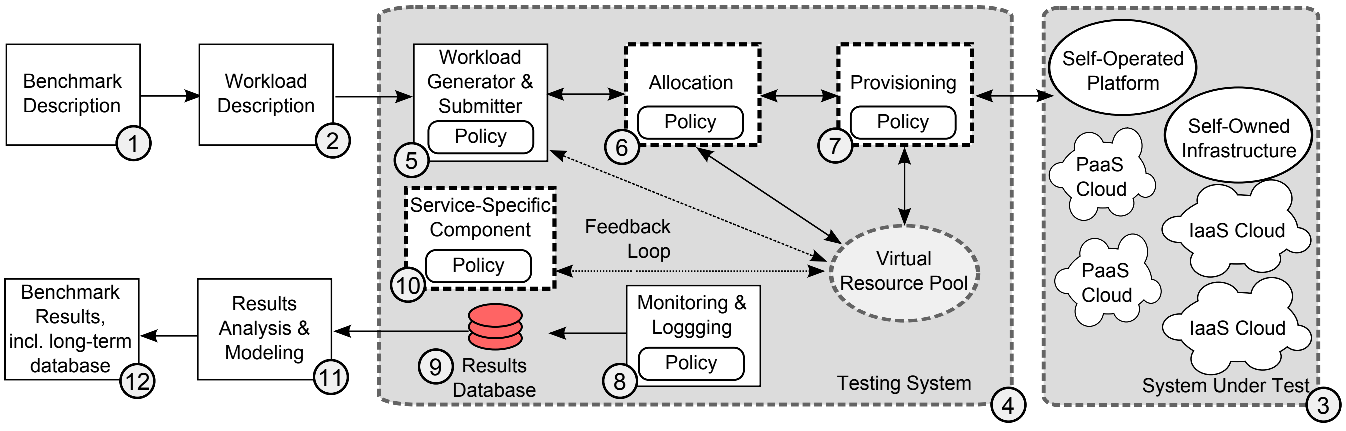


Figure 1: Overview of our generic architecture for benchmarking IaaS and Paas clouds.

cloud benchmarking [34, 29, 33], which in turn builds on our GrenchMark framework for grid benchmarking [30, 61]. The result, a generic architecture for benchmarking IaaS and PaaS clouds, is depicted in Figure 1. The main differences between the architecture proposed in this work, and the generic architecture for IaaS benchmarking we have proposed in our previous work, is the focus of the current architecture on *both* IaaS and PaaS cloud elements. The most important conceptual change occurs for component 10, which the current architecture is making aware of the service to be provided, and in particular of the platform configuration and management policies. In Section 4, where we adapt this architecture for graph analytics PaaS, we show more practical distinctions.

The *Benchmarking Process* consists of the set of rules, prior knowledge (invariants), and procedures used to subject the SUT to the benchmark workload, and to collect and report the results. In our architecture, the process begins with the user (e.g., a prospective cloud user) defining the benchmark configuration, that is, the complex workloads that define the user's preferred scenario (component 1 in Figure 1). The scenario may focus on processing as much of the workload as possible during a fixed test period or on processing a fixed-size workload as quickly or cheaply as possible. The benchmarking system converts (component 2) the scenario into a set of workload descriptions, one per (repeated) execution. The workload may be defined before the benchmarking process, or change (in particular, increase) during the benchmarking process. To increase the statistical confidence in obtained results, subjecting the SUT to a workload may be *repeated* or the workload may be *long-running*. The definition of the workload should avoid common pitfalls that could make the workload unrepresentative [36, 20].

After the preparation of the workload, the SUT (component 3 in Figure 1) is subjected to the workload through the job and resource management services provided by the testing system (component 4, which includes components 5–10). In our benchmarking architecture, the SUT can be comprised of one or several self-owned infrastructures, and public and private IaaS clouds; the SUT can also be comprised of self-managed or cloud-provisioned PaaS, which provide services used by the application developed by the customer. For both IaaS and PaaS usage scenarios, the SUT provides resources for the execution of the workload; these resources are grouped into a *Virtual Resource Pool*. The results produced during the operation of the system may be used to provide a *feedback loop* from the Virtual Resource Pool back into the Workload Generator and Submitter (component 5); thus, our architecture can implement open and closed feedback loops [54].

As a last important sequence of process steps, per-experiment results are combined into higher-level aggregates, first aggregates per workload execution (component 11 in Figure 1), then aggregates per benchmark (component 12). The reporting of metrics should try to avoid the common pitfalls of performance evaluation; see for example [22, 4]. For large-scale distributed systems, it is particularly important to report not only the basic statistics, but also some of the outliers, and full distributions or at least the higher percentiles of the distribution (95-th, 99-th, etc.). We also envision the creation of a general database of results collected by the entire community and shared freely. The organization and operation of such a database is within the scope of future work.

3.2 Distinguishing Design Features

We present in the remainder of this section several of the distinguishing features of this architecture.

Commercial clouds may not not provide (comprehensive) services for managing the incoming stream of requests (components 5, 6, and 8 in Figure 1) or the resources leased from the cloud (components 7 and 8). Our architecture supports various policies for provisioning and allocation of resources (components 6 and 7, respectively). Our generic cloud-benchmarking architecture also includes support for evolving black-box systems (components 9, 11, and 12), complex workloads and multi-tenancy scenarios (components 1, 2, and 5), domain-specific components (component 10), etc.

Experiments conducted on large-scale infrastructure should be designed to minimize the time spent effectively using resources. The interplay between components 1, 2, and 5 in Figure 1 can play a non-trivial role in resolving this challenge, through automatic selection and refinement of complex test workloads that balance the trade-off between accuracy of results and benchmark cost; the main element in a dynamic tuning of this trade-off is the policy present in component 5. The same interplay enables multi-tenancy benchmarks.

Several of the possible SUTs expose complete or partial operational information, acting as white or partially white boxes. Our architecture allows exploiting this information, and combining results from black-box and white-box testing. Moreover, the presence of the increasingly higher-level aggregations (components 11 and 12 in Figure 1) permits both the long-term evaluation of the system, and the combination of short-term and long-term results. The policy for monitoring and logging in component 8 allows the user to customize which information is collected, processed, and stored in the results database, and may result in significantly lower overhead and, for cloud settings, cost. In this way, our architecture goes far beyond simple black-box testing.

Supports domain-specific benchmarks is twofold in our architecture. First, components 5–7 support complex workloads and feedback loops, and policy-based resource and job management. Second, we include in our architecture a domain-specific component (component 10) that can be useful in supporting cloud programming models such as the compute-intensive workflows and bags-of-tasks, and the data-intensive MapReduce and Pregel. The policy element in component 10 allows this component to play a dynamic, intelligent role in the benchmarking process.

4. Graphalytics: A GRAPH ANALYTICS BENCHMARK

In this section, we introduce Graphalytics, a benchmark for graph analytics platforms. Graphalytics is derived from the generic architecture, by adapting and extending several of its key components. Graphalytics includes in the benchmarking process the input dataset, the analytics algorithm, and the provisioned infrastructure. It provides components for platform configuration, deployment, and monitoring, and already provides reference implementations for several popular platforms, such as Giraph and Hadoop/YARN. The Graphalytics API offers a unified execution environment for all platforms, and consistent reporting that facilitates comparisons between all possible combinations of platforms, datasets, and algorithms. The Graphalytics API also permits developers to integrate new graph analytics platforms into the benchmark.

4.1 Design Overview

The design of Graphalytics is based on the generic benchmarking architecture, and on our previous work on benchmarking graph processing systems [25, 27].

The components for benchmark and workload description (components 1 and 2 in Figure 2, respectively) in the Graphalytics process are derived from our previous work [27] in benchmarking graph-processing platforms. For the benchmark description (component 1), Graphalytics focuses, process-wise, on processing as many edges or vertices per second, for a variety of input datasets and processing algorithms. The cost of processing, when known, is also of interest for this benchmarking process.

For the workload description (component 2), Graphalytics focuses on algorithms and input datasets. Graphalytics includes already five classes of algorithms, one for each major class of algorithms covered by research published in major conferences by the distributed systems and the databases communities, as indicated by Table 1. More classes of algorithms are currently added to Graphalytics; to keep the number of tested algorithms manageable, we will define more diverse benchmarking scenarios, such that each scenario can focus on a sub-set of existing algorithms.

Component 5 focuses on generating and submitting workloads representative for graph analytics. Graphalytics defines here an archive of commonly used datasets, such as the group considered in our previous work [25]. The archive can grow in time during use, as datasets are either imported, or generated and saved in the database. Currently, Graphalytics includes datasets from the SNAP and GTA [26] collections, and recommends the use of synthetic datasets generated with the Graph500 [23] or the LDBC Social Networking Benchmark [39]. We envision here alternatives to the implementation of this component, especially exploiting the trade-off between fast submission (reads from the database or full-scale generation) and cost (of storage, of computation, etc.)

The service-specific component (component 10) ensures the graph analytics service executes the correct, possibly platform-specific or tuned version of the algorithm, on the input workload provided by the workload generator and submitter (component 5). Additionally, each platform can have specific configuration parameters for the execution of an arbitrary algorithm. Last, the service-specific component checks that the outcome of the execution fulfills the validity criteria of the benchmark, such as output correctness, response time limits, etc.

The monitoring and logging component (component 8) records relevant information about the SUT during the benchmark. Graphalytics supports monitoring and logging for various graph analytics platforms, from the distributed Giraph to the single-node, GPU-based Medusa.

Component 11 enables automated analysis and modeling of benchmarking results. Upon completion of each benchmarking session, the results are aggregated both in human-readable format for direct presentation to the user, and in a standard machine-readable format for storage in the results database. The machine-readable format enables automated analysis of individual platforms, and comparison of results (via component 12). Graphalytics also exposes an API to the platform to enable the platform to signal to the system monitor important events during execution. These events can be superimposed on resource utilization graphs in the benchmark report to enable users to identify bottlenecks and better understand system performance.

Graphalytics includes a centralized repository of benchmark results (component 12), which holds data for various platforms and deployments, and enables comparison of operational data and performance results across many platforms.

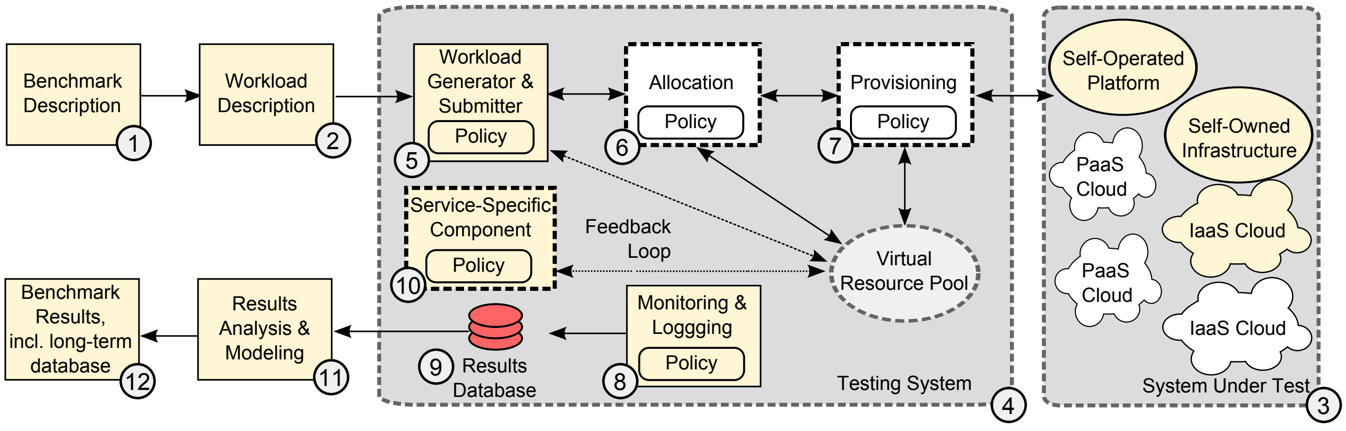


Figure 2: The Graphalytics architecture for benchmarking graph analytics clouds. The highlighted components indicate the parts of the generic architecture for cloud benchmarking that Graphalytics adapts and extends.

Table 1: Survey of algorithms used in graph analytics. (Source: [27])

Class	**Typical algorithms**	**Number**	**Percentage [%]**
General Statistics	Triangulation [60], Diameter [37], BC [56]	24	16.1
Graph Traversal	BFS, DFS, Shortest Path Search	69	46.3
Connected Components	MIS [9], BiCC [15], Reachability [11]	20	13.4
Community Detection	Clustering, Nearest Neighbor Search	8	5.4
Graph Evolution	Forest Fire Model [41], Preferential Attachment Model [6]	6	4.0
Other	Sampling, Partitioning	22	14.8
Total		149	100

The repository offers a comprehensive overview of performance results from the graph analytics community.

4.1.1 *Benchmarking with Graphalytics in Practice*

We present now the practical steps for benchmarking with Graphalytics.

The benchmark reads the configuration files describing the datasets and platforms to be benchmarked. For datasets, the configuration includes the path to data, the format of the data (e.g., edge-based or vertex-based; directed or undirected; etc.), and parameters to each algorithm (e.g., source vertex for BFS, maximum number of iterations for CD, etc.). Graphalytics includes a comprehensive set of datasets for which configuration files are also provided, requiring no further user input. To run a benchmark on a platform supported by Graphalytics, using a dataset included in Graphalytics, users only need to configure the details of their platform setup, e.g., set the path of `HADOOP_HOME` for benchmarking Hadoop MapReduce2.

After reading the configuration, Graphalytics runs the platform-specific implementation of each algorithm included in the configuration. For each combination of dataset and algorithm, the graph analytics platform under test is instructed to execute the platform-specific implementation of the algorithm. The core is also responsible for by uploading the selected dataset to the platform.

The system monitor records relevant information about the graph analytics platform under test during the benchmark. After the test, the output validator checks the results to ensure their validity. The report generator gathers for every dataset and algorithm pair the results from the system monitor and the results validator. Users can export these results, and share them with the community through the centralized repository.

The Graphalytics prototype already includes an implementation of extended monitoring, for the Giraph platform. Figure 3 depicts in its top and middle sub-plots two platform-specific performance metrics collected during a benchmark run, messages sent and memory used, respectively. The bottom sub-plot of the figure depicts the evolution of processing in terms of BSP synchronization supersteps, which characterize the processing model used in Giraph (and Pregel). The supersteps are aligned with the performance metrics and enable the user to associate the metrics with the evolution of the processing algorithm. For example, the memory usage varies in accordance with the succession of supersteps.

4.1.2 *Extensibility of Graphalytics*

The aim of the Graphalytics project is to include as many algorithms and platforms possible. To accommodate this, the design of the Graphalytics is extensible through simple APIs between components.

Developers can add new graph analytics platforms by providing the following: (a) an implementation of each algorithm defined by the Graphalytics project; (b) functions to upload datasets to the platform, and to download results from the platform; (c) a function to execute a specific al-

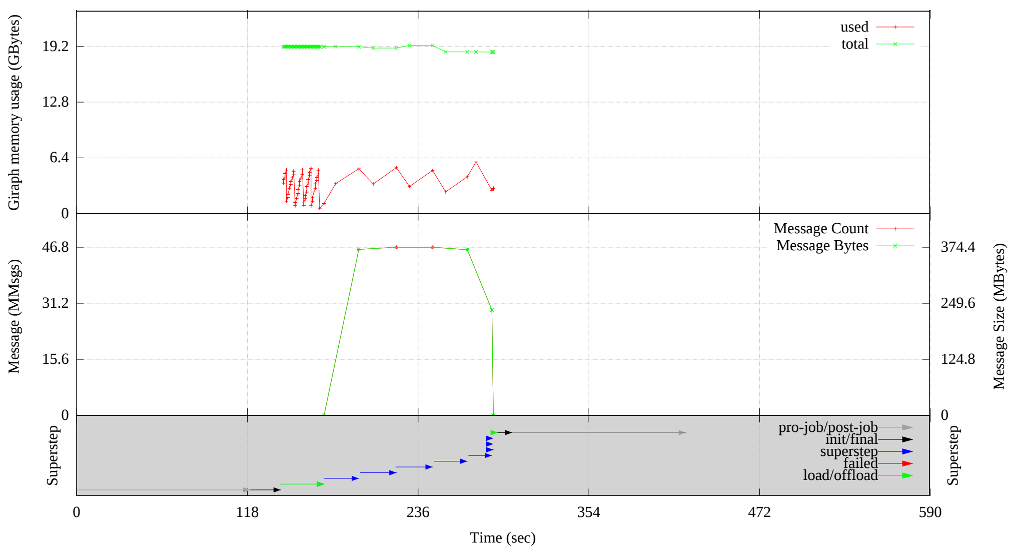


Figure 3: Resource utilization during a Giraph measurement.

gorithm on a specific dataset. Because datasets are managed by the Graphalytics benchmark, no platform-specific changes are needed to support a new dataset; conversely, no new dataset configuration is needed when a new platform is added.

The addition of new algorithms is also supported. Adding a new algorithm requires change in both datasets and platforms. For each dataset, a new set of algorithm-specific parameters needs to be added; a default is provided. For each platform, an implementation of the algorithm must be provided to complete the benchmark.

4.2 Implementation details

We have implemented a Graphalytics prototype in Java, using `.properties` files for configuration. We include support for MapReduce2 and Giraph, running on Apache Hadoop 2.4.1 or later. Our Giraph implementation of algorithms uses the YARN resource resource manager introduced in Apache Hadoop 2, avoiding the overhead of the MapReduce framework. We have also experimented, using earlier prototypes of Graphalytics and a simplified benchmarking process, with MapReduce/Hadoop 0.20, Stratosphere 0.2, GraphLab 2.1, and Neo4j 1.5.

In Listing 1, we show an example of a graph configuration file (lines starting with a double back-slash, “\\”, are comments). Most elements are optional, e.g., there are no parameters for the connected components algorithm (CONN, lines 19 and 20 in the listing) because none are required.

The benchmarking scenario is `graph.ldbc-30`, which correspond to experiments focusing on the generation and validation of LDBC data, scale 30. The generated data is stored as a physical file on the local filesystem where the generator runs, with the file name specified in line 2 of Listing 1. The directedness and the format of the generated dataset are specified in lines 5 and 6, respectively.

The benchmarking scenario runs five algorithms, specified on line 9 of Listing 1. Except for the algorithm for connected components, all other algorithms require one or several parameters. For example, the BFS graph-traversal algorithm requires a source–the graph vertex where the traversal starts–, which is specified on line 12.

```
 1  # Filename of graph on local filesystem
 2  graph.ldbc-30.file = ldbc-30
 3
 4  # Properties describing the directedness and format
 5  graph.ldbc-30.directed = true
 6  graph.ldbc-30.edge-based = true
 7
 8  # List of algorithms supported on the graph
 9  graph.ldbc-30.algorithms = bfs, cd, conn, evo, stats
10
11  # Parameters for BFS
12  graph.ldbc-30.bfs.source-vertex = 12094627913375
13
14  # Parameters for CD
15  graph.ldbc-30.cd.node-preference = 0.1
16  graph.ldbc-30.cd.hop-attenuation = 0.1
17  graph.ldbc-30.cd.max-iterations = 10
18
19  # Parameters for CONN
20
21  # Parameters for EVO
22  graph.ldbc-30.evo.max-id = 2000000000000000
23  graph.ldbc-30.evo.pratio = 0.5
24  graph.ldbc-30.evo.rratio = 0.5
25  graph.ldbc-30.evo.max-iterations = 6
26  graph.ldbc-30.evo.new-vertices = 10
27
28  # Parameters for STATS
29  graph.ldbc-30.stats.collection-node = 12094627913375
```

Listing 1: Sample configuration for an Graphalytics instance.

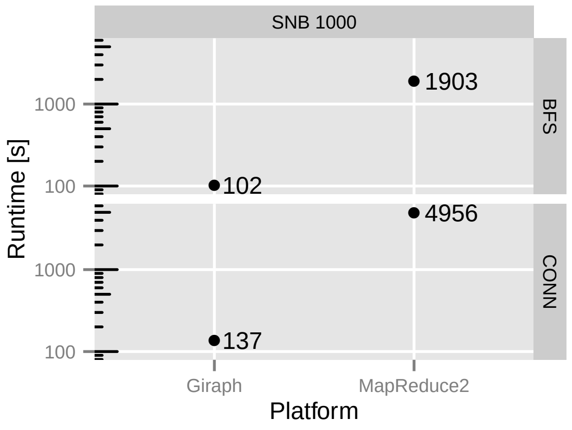


Figure 4: Algorithm runtimes obtained with Graphalytics.

4.3 Experimental Results

We have used the Graphalytics prototype for two broad types of platforms, distributed and GPU-based. Exploring the performance of these platforms is currently a hot topic, with interesting recent results [25, 28, 42].

We ran Graphalytics using the MapReduce2 and Giraph implementations of BFS and CONN on the SNB 1000 dataset. We used, for benchmarking these *distributed graph-analytics platforms*, a 10-worker cluster with a separate management node for the HDFS name node, YARN resource manager, and ZooKeeper. The results are presented in Figure 4. We have reported much more in-depth results for other scenarios, but obtained with much more difficulty in setup and execution of the benchmarks because of the lack of tool similar to Graphalytics, in our previous work [25]. We conclude that Graphalytics is a useful benchmark for distributed platforms.

We tested several popular *GPU-based graph analytics platforms* using the LDBC SNB data generator, although the benchmarking code is not yet fully integrated in Graphalytics. Figure 5 depicts the results for the platforms Medusa (`M` in the figure); Totem (`T-G` for the Totem version using only the GPU as computing resource, and `T-H` for the Totem version using both the GPU and the CPU as a hybrid computing resource); and MapGraph (`MG`). The benchmarking process runs for three GPU-based systems, with GPUs of three different generations and internal architectures. When the integration of this benchmarking process will be complete, Graphalytics will also fully support a variety of graph analytics platforms using GPUs or heterogeneous hardware.

5. BENCHMARKING CHALLENGE: INCLUDING THE ALGORITHM IN THE SUT

Given the extreme scale of the datasets that need to be analyzed, as well as the increasingly complex analysis that needs to be performed, graph analytics has become a high-performance computing (HPC) concern. This trend is probably best proven by the intense activity and fast changes happening in the Graph500[2] ranking, as well as in the adoption of graph traversals as important benchmarks [13] and drivers for irregular algorithms programming paradigms [49].

At the same time, the state-of-the-art in high performance computing is massive parallel processing, backed up by a large variety of parallel platforms ranging from graphical processing units (GPUs) to multi-core CPUs and Xeon Phi. Because traditional graph processing algorithms are known for their parallelism-unfriendly features - data-dependency, irregular processing, bad spatial and temporal locality [1] - a lot of work has focused on developing GPU-specific [38, 52, 10, 46], multi-core CPU-specific [2], or even vendor-specific [14, 50] algorithms.

All this work proves an important point: most graph analytics applications can be solved by multiple, different algorithms. These algorithms show very different performance behavior on different platforms *and* on different datasets. Therefore, we argue that not algorithms are the same just because they solve the same problem! Therefore, the selection of the algorithm, which often dictates a pre-selection of the data representation and a filtering of the hardware building blocks, has a huge impact on the observed performance.

To illustrate the importance of the algorithm selection, we present in the remainder of this section a comprehensive evaluation of a large set of 13 parallel breadth-first search (BFS) algorithms built and optimized for GPU execution. A similar study for multi-core CPUs, but not for GPUs, is available in [2]. We point out that the number of algorithms and variations designed for massively parallel architectures such as GPUs is significantly higher, making the results of our GPU study even more interesting.

5.1 Parallel BFS algorithms for GPU execution

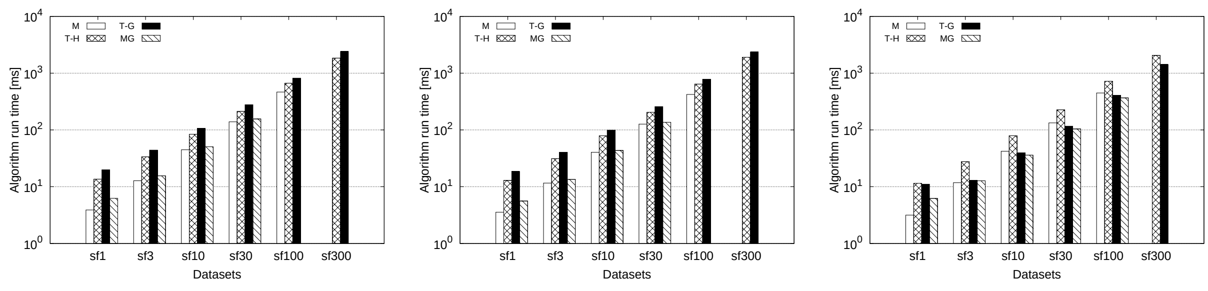
A BFS traversal explores a graph level by level. Given a graph $G = (V, E)$, with V its collection of vertices and E its collection of edges, and a source vertex s (considered as the only vertex on level 0), BFS systematically explores edges outgoing from vertices at level i and places all their destination vertices on level $i + 1$, *if* these vertices have not been already discovered at prior levels (i.e., the algorithm has to distinguish *discovered* and *undiscovered* vertices to prevent infinite loops).

5.1.1 Naïve BFS

In a BFS that accepts an edge list as input, an iteration over the entire set of edges is required for each iteration. By a simple check on the source vertex of an edge, the algorithm can determine which edges to traverse, hence which destination vertices to place in the next level of the resulting tree.

Our parallel BFS works by dividing the edge list into sublists, which are processed in parallel: each thread will traverse its own sub-list in every iteration. Synchronization between levels is mandatory to insure a full exploration of the current level before starting the next one.

[2]http://www.graph500.org



(a) GTX480 (b) GTX580 (c) K20m

Figure 5: Runtimes of GPU-based platforms executing PageRank on different GPUs and LDBC SNB dataset scales.

When mapping this parallel kernel to OpenCL, each thread is mapped to an work-item. As global synchronization is necessary, we implement a two-layer barrier structure: first at work-group level (provided by OpenCL), then between work-groups (implemented in-house). This solution limits the synchronization penalty - see [48],Chapter 3 for more details).

Our results - presented below in Section 5.2 - show that this approach, although naive, can outperform more optimized versions for specific datasets. Moreover, when comparing it with the naive vertex-centric BFS version included in the Rodinia benchmark [13], the results remain very interesting: in some cases, the vertex-centric version performs better, while in others it is significantly worse than our naive edge-based version [58].

5.1.2 BFS at Maximum Warp

The major performance penalty where running massively parallel traversals on GPUs is work imbalance. This imbalance can appear due to the inability of the programming model to express enough fine-tune options. In [52], the authors present a way to systematically deal with this lack of fine-grain tuning and demonstrate it on a new BFS solution. Specifically, they propose a warp-centric solution, with different parallel stages. This approach enables load-balancing at the finest granularity, thus limiting the penalty of coarser load imbalanced solutions.

We have implemented this approach and varied its granularity. Our results show that , for certain graphs, this algorithm delivers a significant performance boost when compared with the alternatives, but for other datasets it is outperformed by more optimized versions.

5.1.3 The Lonestar suite

The LonestarGPU collection[3], presented in detail in [10], includes a set of competitive BFS implementations specifically designed to use the processing power of GPUs. The brief descriptions of these algorithms is presented in Table 2.

We point out that each of these algorithms uses different GPU constructs to implement the same BFS traversal. However, depending on the dataset that is being benchmarked, the performance impact of such optimizations varies, showing better results for one or another variant.

[3]http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

Table 2: Brief description of the LonestarGPU BFS algorithms.

lonestar	A topology-driven, one node-per-thread version.
topology-atomic	A topology-driven version, one node-per-thread version that uses atomics.
merrill	Merrill's algorithm [45].
worklist-w	A data-driven, one node-per-thread version.
worklist-a	A flag-per-node version, one node-per-thread version.
worklist-c	A data-driven, one edge-per-thread version using Merrill's strategy.

Table 3: The datasets selected for our BFS exploration.

Graph	Vertices	Edges	Diameter	90-percentile Diameter
as-skitter	1,696,415	11,095,298	25	6
facebook-combined	4,039	88,234	8	4.7
roadNet-CA	1,965,206	5,533,214	849	500
web-BerkStan	685,23	7,600,595	514	9.9
wiki-Talk	2,394,385	5,021,410	9	4

5.2 Experiments and results

We have run 13 different BFS solutions - our edge-based naive version, the maximum warp version with varying warp size (1,2,4,8,16, and 32), and the 6 algorithms from LonestarGPU - on three different GPUs - a GTX480, a C2050, and a K20m. In this section, we only present our results from the newest GPU architecture in this series, namely the K20m Kepler-based machine. For the full set of results, we redirect the reader to [44]. The datasets we have selected (from the SNAP repository [40]) are presented in 3. We also note that for the `as-skitter` and `roadNet-CA` graphs, we have used both the directed and undirected versions of the graph.

Our results on the K20m GPU are presented in Figures 6 (for the relative performance of the kernels only) and 7 (for the relative performance of the full BFS run, including the data tranfers).

We make the following observations. First, the difference between the best and worst version for each dataset can be as large as *three orders of magnitude*, which means that the choice of algorithm for a given application must be carefully made by the user, prior to measuring and analyzing performance, or be included in the SUT and measured as part of system.

Second, there is no best or worst performing algorithm across

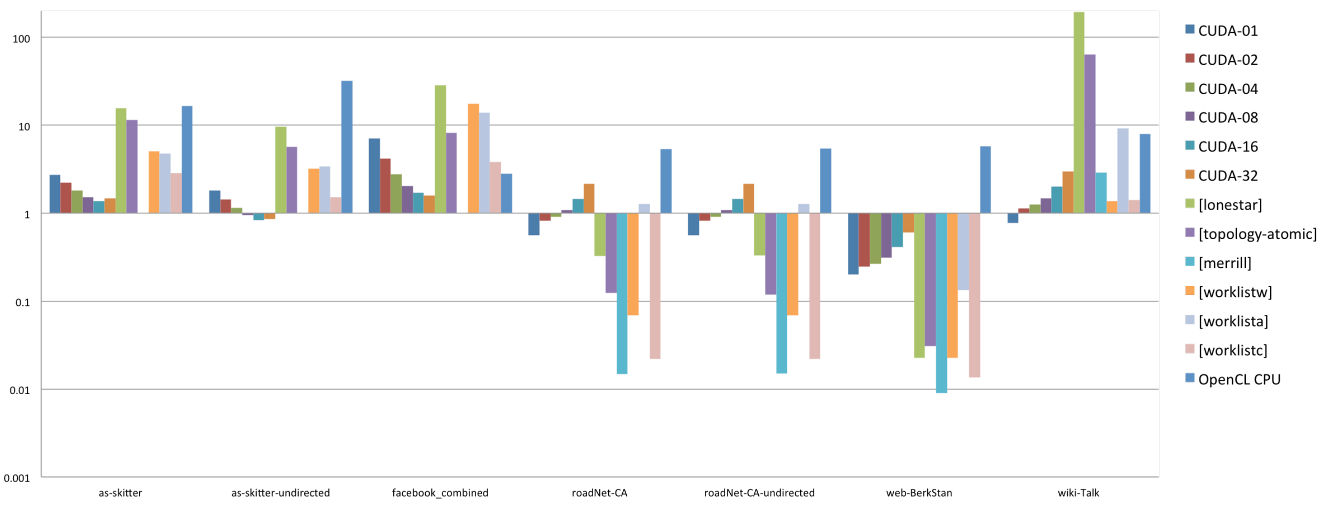


Figure 6: Parallel BFS performance of the kernel for 13 different GPU implementations. The presented speed-up is normalized to the naive edge-based version (reported as reference, at 1).

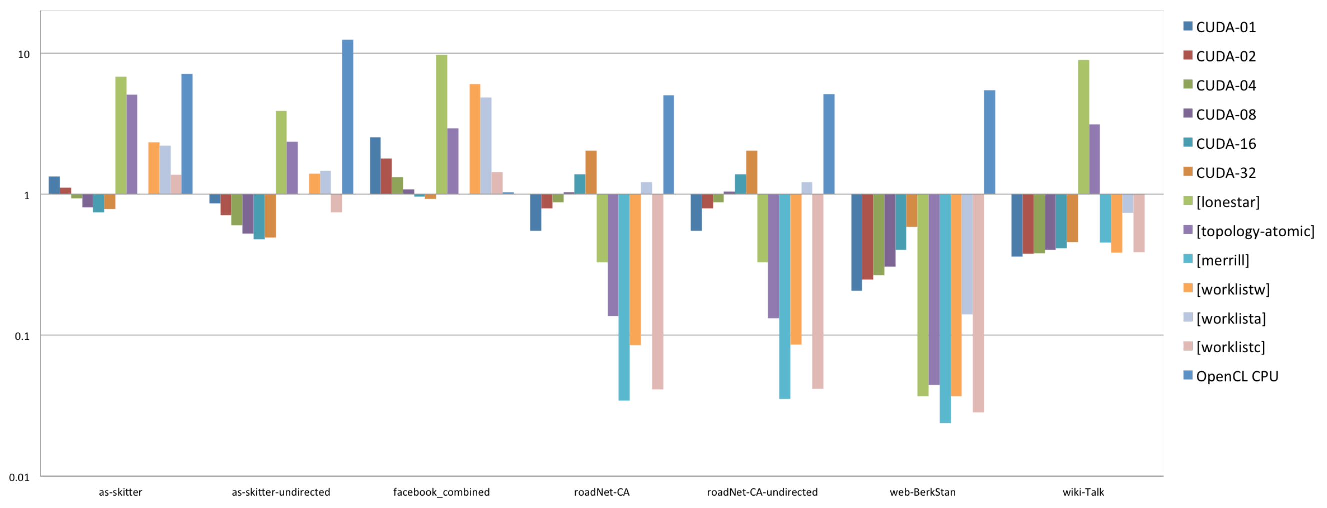


Figure 7: Parallel BFS performance of the full BFS run (including data transfers to and from the GPU) for 13 different implementations. The presented speed-up is normalized to the naive edge-based version (reported as reference, at 1).

all datasets. This indicates that the relative performance of the algorithms also varies with the input data. This variation demonstrates that no single algorithm can be labeled "the best GPU BFS implementation", and indicates that a complete benchmark should include different BFS versions for a comprehensive evaluation.

Third, and final, taking the data transfer times into account levels the differences between different implementations: this special GPU-related overhead is usually so much higher than the execution of the BFS kernel itself, that the small differences between different algorithms become less visible. We argue that this should be reported, for fairness, when comparing against other platforms (see the OpenCL CPU version in Figure 7), but should be eliminated when benchmarking only GPU-based platforms.

5.3 Practical guidelines

Our study on BFS running on GPUs demonstrates that the algorithms play an essential role in the performance of the application. Given that the increasing variety of modern parallel architectures leads to an increasing variety of algorithms—some portable, some not—for the same application, the traditional benchmarking approach where the application dictates the algorithm must be deprecated. Much like sorting is a problem with many solutions, many graph applications have many parallel and distributed algorithmic solutions. These algorithms need to be first-class citizens in modern benchmarking.

The main challenge for *algorithm-aware benchmarking* is ensuring completeness. In this context, complete coverage, that is, including all possible algorithms, is not feasible in a reasonable time. Defining the most representative algorithms, which is the typical approach in benchmarking, poses the same problems as defining the most representative applications: it requires criteria that are difficult to define without years of practice and experience with each algorithms. Finally, keeping up-to-date with all the new algorithm developments requires an unprecedented level of implementation and engineering efforts.

Therefore, we believe that an algorithmic-aware benchmarking is necessary, but must be a community-based effort. Without the expertise of different researchers in domains ranging from algorithmics to benchmarking and from single-node to cloud-level platforms, we cannot overcome the technical and completeness challenges that arise in graph processing. Without this effort, the benchmarks we build will be constraint by design.

6. CONCLUSION AND ONGOING WORK

The success of cloud computing services has already affected the operation of many, from small and medium businesses to scientific HPC users. Addressing the lack of a generic approach for cloud service benchmarking, in this work we propose a generic architecture for benchmarking IaaS and PaaS clouds. In our generic architecture, resource and job management can be provided by the testing infrastructure, there is support for black-box systems that change rapidly and can evolve over time, tests are conducted with complex workloads, and various multi-tenancy scenarios can be investigated.

We adapt the generic architecture to data-intensive platforms, and design Graphalytics, a benchmark for graph analytics platforms. Graphalytics focuses on the relationship between the input dataset, the analytics algorithm, and the provisioned infrastructure, which it quantifies with diverse performance and system-level metrics. Graphalytics also offers APIs for extending its components, and for supporting more graph analytics platforms than available in our current reference implementation. Experiments we conduct in real-world settings, and with distributed and GPU-based graph analytics platforms, give strong evidence that Graphalytics can provide a unified execution environment for all platforms, and consistent reporting that facilitates comparisons between all possible combinations of platforms, datasets, and algorithms.

Derived from the experience we have accumulated evaluating graph analytics platforms, we identify an important new challenge for benchmarking in clouds: including the algorithm in the system under test, to also benchmark the platform-specific and data-dependent performance of the various algorithms.

We are currently implementing Graphalytics and refining its ability to capture algorithmic bottlenecks. However, to succeed Graphalytics cannot be a single-team effort. We have initiated various community-wide efforts via our work in the SPEC Research Group and its Cloud Working Group, and are currently looking to connect to the SPEC Big Data Working Group.

Acknowledgments

This work is supported by the Dutch STW/NOW Veni personal grants @large (#11881) and Graphitti (#12480), by the EU FP7 project PEDCA, by the Dutch national program COMMIT and its funded project COMMissioner, and by the Dutch KIEM project KIESA. The authors wish to thank Hassan Chafi and the Oracle Research Labs, Peter Boncz and the LDBC project, and Josep Larriba-Pey and Arnau Prat Perez, whose support has made the Graphalytics benchmark possible; and to Tilmann Rabl, for facilitating this material.

7. REFERENCES

[1] B. H. A. Lumsdaine, D. Gregor and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 2007.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

[3] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *ISPASS*, pages 2–9. IEEE Computer Society, 2005.

[4] J. N. Amaral. How did this get published? Pitfalls in experimental evaluation of computing systems. LTES talk, 2012. [Online] Available: `http://webdocs.cs.ualberta.ca/~amaral/Amaral-LCTES2012.pptx`. Last accessed Oct 2012.

[5] Amazon Web Services. Case studies. Amazon web site, Oct 2012. [Online] Available: `http://aws.amazon.com/solutions/case-studies/`. Last accessed Oct 2012.

[6] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–12, 1999.

[7] P. Brebner, E. Cecchet, J. Marguerite, P. Tuma, O. Ciuhandu, B. Dufour, L. Eeckhout, S. Frénot, A. S. Krishna, J. Murphy, and C. Verbrugge. Middleware benchmarking: approaches, results, experiences. *Concurrency and Computation: Practice and Experience*, 17(15):1799–1805, 2005.

[8] A. Buble, L. Bulej, and P. Tuma. Corba benchmarking: A course with hidden obstacles. In *IPDPS*, page 279, 2003.

[9] A. Buluç, E. Duriakova, A. Fox, J. R. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams. High-Productivity and High-Performance Analysis of Filtered Semantic Graphs. In *IPDPS*, 2013.

[10] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[11] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, 2010.

[12] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *JSSPP*, pages 67–90, 1999.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *The 2009 IEEE International Symposium on Workload Characterization, IISWC'09*, pages 44–54, 2009.

[14] F. Checconi and F. Petrini. Massive data analytics: The graph 500 on ibm blue gene/q. *IBM Journal of Research and Development*, 57(1/2):10, 2013.

[15] G. Cong and K. Makarychev. Optimizing Large-scale Graph Analysis on Multithreaded, Multicore Platforms. In *IPDPS*, 2012.

[16] E. Deelman, G. Singh, M. Livny, J. B. Berriman, and J. Good. The cost of doing science on the cloud: the Montage example. In *SC*, page 50. IEEE/ACM, 2008.

[17] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Performance Evaluation Review*, 26(4):14–29, 1999.

[18] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[19] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *TPCTC*, pages 173–188, 2012.

[20] E. Frachtenberg and D. G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *JSSPP*, pages 257–282, 2005.

[21] D. Genbrugge and L. Eeckhout. Chip multiprocessor design space exploration through statistical simulation. *IEEE Trans. Computers*, 58(12):1668–1681, 2009.

[22] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

[23] Graph500 consortium. Graph 500 benchmark specification. Graph500 documentation, Sep 2011. [Online] Available: `http://www.graph500.org/specifications`.

[24] J. Gray, editor. *The Benchmark Handbook for Database and Transasction Systems*. Mergan Kaufmann, 2nd ed. edition, 1993.

[25] 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. In *IPDPS*, 2014.

[26] Y. Guo and A. Iosup. The Game Trace Archive. In *NETGAMES*, pages 1–6, 2012.

[27] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking graph-processing platforms: a vision. In *ICPE*, pages 289–292, 2014.

[28] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[29] A. Iosup. Iaas cloud benchmarking: approaches, challenges, and experience. In *HotTopiCS*, pages 1–2, 2013.

[30] A. Iosup and D. H. J. Epema. GrenchMark: A framework for analyzing, testing, and comparing grids. In *CCGrid*, pages 313–320, 2006.

[31] A. Iosup, D. H. J. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, and R. Yahyapour. On grid performance evaluation using synthetic workloads. In *JSSPP*, pages 232–255, 2006.

[32] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Par. Dist. Syst.*, 22(6):931–945, 2011.

[33] A. Iosup, R. Prodan, and D. Epema. Iaas cloud benchmarking: Approaches, challenges, and experience. In X. Li and J. Qiu, editors, *Cloud Computing for Data-Intensive Applications*. Springer Verlag, New York, USA, 2015.

[34] A. Iosup, R. Prodan, and D. H. J. Epema. Iaas cloud benchmarking: approaches, challenges, and experience. In *SC Companion/MTAGS*, 2012.

[35] K. R. Jackson, K. Muriki, L. Ramakrishnan, K. J. Runge, and R. C. Thomas. Performance and cost analysis of the supernova factory on the amazon aws cloud. *Scientific Programming*, 19(2-3):107–119, 2011.

[36] R. Jain, editor. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons Inc., 1991.

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

[38] G. J. Katz and J. T. K. Jr. All-pairs shortest-paths for large graphs on the gpu. *23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55, 2008.

[39] LDBC consortium. Social network benchmark: Data generator. LDBC Deliverable 2.2.2, Sep 2013. [Online] Available: `http://ldbc.eu/sites/default/files/D2.2.2_final.pdf`.

[40] J. Leskovec. Stanford network analysis platform (snap). *Stanford University*, 2006.

[41] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[42] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[43] P. Mell and T. Grance. The NIST definition of cloud computing. National Institute of Standards and Technology (NIST) Special Publication 800-145, Sep 2011. [Online] Available: `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. Last accessed Oct 2012.

[44] C. d. L. Merijn Verstraaten, Ana Lucia Varbanescu. State-of-the-art in graph traversals on modern arhictectures. Technical report, University of Amsterdam, August 2014.

[45] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012.

[46] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.

[47] M. Oskin, F. T. Chong, and M. K. Farrens. Hls: combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, pages 71–82, 2000.

[48] A. Penders. Accelerating Graph Analysis with

Heterogeneous Systems. Master's thesis, PDS, EWI, TUDelft, December 2012.

[49] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

[50] X. Que, F. Checconi, and F. Petrini. Performance analysis of graph algorithms on p7ih. In *ISC*, pages 109–123, 2014.

[51] I. Raicu, Z. Zhang, M. Wilde, I. T. Foster, P. H. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *SC*, page 22. ACM, 2008.

[52] T. O. S. Hong, S. K. Kim and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. *Principles and Practice of Parallel Programming, PPoPP'11*, 2011.

[53] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[54] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, 2006.

[55] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. E. Taylor, and X. Wu. Performance projection of hpc applications using spec cfp2006 benchmarks. In *IPDPS*, pages 1–12, 2009.

[56] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP*, 2013.

[57] J. Spacco and W. Pugh. Rubis revisited: Why j2ee benchmarking is hard. *Stud. Inform. Univ.*, 4(1):25–30, 2005.

[58] A. L. Varbanescu, M. Verstraaten, C. de Laat, A. Penders, A. Iosup, and H. Sips. Can portability improve performance? an empirical study of parallel graph analytics. In *ICPE*, 2015.

[59] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 612–619, 2012.

[60] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On Triangulation-based Dense Neighborhood Graphs Discovery. *VLDB*, 2010.

[61] N. Yigitbasi, A. Iosup, D. H. J. Epema, and S. Ostermann. C-meter: A framework for performance analysis of computing clouds. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009*, pages 472–477, 2009.