Elasticity in Graph Analytics? A Benchmarking Framework for Elastic Graph Processing

Alexandru Uta[§], Sietse Au[*], Alexey Ilyushkin[*], and Alexandru Iosup[§*]
[§]Vrije Universiteit Amsterdam, [*]TU Delft
{a.uta, a.iosup}@vu.nl, s.t.au@student.tudelft.nl, a.s.ilyushkin@tudelft.nl

***Abstract*—Graphs are a natural fit for modeling concepts used in solving diverse problems in science, commerce, engineering, and governance. Responding to the diversity of graph data and algorithms, many parallel and distributed graph-processing systems exist. However, until now these platforms use a *static* model of deployment: they only run on a pre-defined set of machines. This raises many conceptual and pragmatic issues, including misfit with the highly dynamic nature of graph processing, and could lead to resource waste and high operational costs. In contrast, in this work we explore the benefits and drawbacks of the *dynamic* model of deployment. Building a three-layer benchmarking framework for assessing elasticity in graph analytics, we conduct an in-depth elasticity study of distributed graph processing. Our framework is composed of state-of-the-art workloads, autoscalers, and metrics, derived from the LDBC Graphalytics benchmark and SPEC RG Cloud Group's elasticity metrics. We uncover the benefits and cost of elasticity in graph processing: while elasticity allows for fine-grained resource management, and does not degrade application performance, we find that graph workloads are sensitive to data migration while leasing or releasing resources. Moreover, we identify non-trivial interactions between scaling policies and graph workloads, which add an extra level of complexity to resource management and scheduling for graph processing.**

I. Introduction

To address the big data challenges raised by graph-data and algorithms (trillions of edges and daily processing at Facebook-scale [1]), our community has designed tens of sophisticated graph-processing systems. Although capable and high-performance, these systems use a *static* model of deployment, where the analytics platform runs on a fixed, statically defined set of machines. The static model lacks conceptual fit with the dynamicity of graph applications, and raises pragmatic issues of resource-waste and system-availability. Contrasting the many techniques focusing on dynamic *workload* partitioning [2]–[4] and re-partitioning [5]–[7] across a static infrastructure, in this work we focus on studying a *dynamic model of deployment* for distributed graph-processing. Concretely, we investigate what tools to use for assessing the benefits and cost of *elastic graph analytics*.

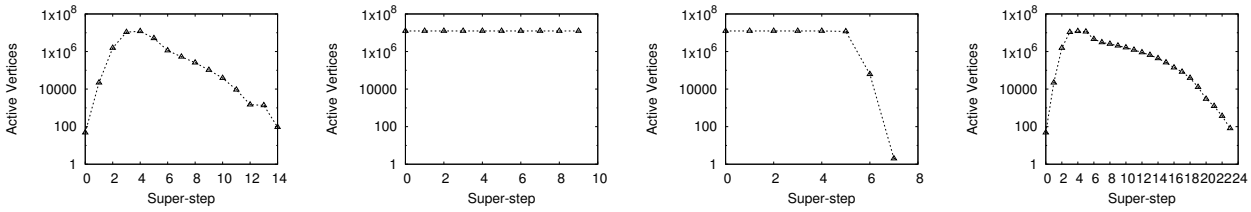
Indicating the increasing need for ever-larger computing resources, graph-processing systems have already been deployed on many types of infrastructure and offered as *analytics platforms*, when commercialized as software and services. So far, industry and academia have demonstrated the use for graph processing on a variety of systems with *static* resource deployment: private clusters [8]–[10], supercomputers [11], heterogeneous CPU-GPU [12], and CPU and (multi-)GPU systems [13], and public clouds [14]–[16].

The use of the static deployment model has negative consequences. Conceptually, graph applications are a poor fit for static, fixed-size infrastructure, because they have often iterative, but highly irregular workloads [5], [7], [17]. Pragmatically, even experts face difficult choices based on empirical approaches to estimate the size of the needed infrastructure; there are serious consequences when the choice is inaccurate, ranging from system-wide crashing when resources are under-provisioned [18], to possibly high resource waste and operational costs when the system is over-provisioned.

State-of-the-art graph analytics systems do not focus on a *dynamic model of deployment*, where infrastructure can grow and shrink to match the needs of the graph-processing application. Elasticity promises to enable many opportunities for improved, more efficient operation, for both public (e.g., cloud) and private infrastructure (e.g., clusters, supercomputers, private-clouds). For public infrastructure, elasticity could reduce operational costs by *reducing resource waste* [19], [20], and improve the ability to meet Quality-of-Service guarantees (such as high performance and availability) by using appropriate auto-scaling policies [21]. For private infrastructure, elasticity could *reduce operational costs* by increasing resource utilization [22], or throttle throughput to meet demands across applications [23].

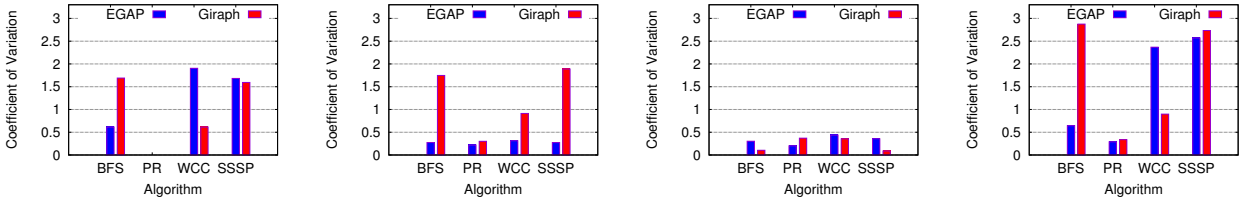
Moreover, because elasticity allows system administrators to change initial commitments of resources, it could be particularly important for the diverse field of graph processing. Instead of estimating the number of machines needed to run specific graph-processing jobs, which is an open challenge [24], harnessing elasticity allows deferring to runtime decisions, when resources are transparently added or removed as needed; thus, elasticity could *reduce the risk of failure and of over-provisioning*. Consequence of the irregular, variable use of resources that characterizes graph-processing (Section II), elasticity promises to be a good conceptual fit by adapting to the workload characteristics.

However promising, elasticity also has potential drawbacks. *Scaling* introduces *resource-management complexity*: how to make a system take smart decisions about leasing or releasing resources (e.g., what, when, for how long)? *How to uncover the costs and benefits of elasticity in distributed graph processing systems?* To answer these research questions, we extend our poster [25] by making the following contributions:



(a) BFS on Datagen-1000. (b) PR on Datagen-1000. (c) WCC on Datagen-1000. (d) SSSP on Datagen-1000.

Fig. 1. Variability in graph-processing algorithms, exemplified through `JoyGraph EGAP`: (a)-(d) depict active vertices vs. algorithm super-step.



(a) Active Vertices. (b) CPU Load. (c) Memory. (d) Wallclock.

Fig. 2. Variability in graph processing systems, exemplified through `JoyGraph EGAP` and Giraph: the coefficient of variation of different metrics, computed per node per super-step for the Datagen-1000 graph. The coefficient of variation is assessed for 4 graph processing algorithms: BFS, PR, WCC, and SSSP.

1) We show strong evidence that typical graph-processing workloads exhibit significant variability in resource utilization (Section II). Our empirical analysis focuses on application- and system-level metrics, which goes beyond what state-of-the-art graph-processing systems consider.
2) We propose the design of a three-layer, modular, elastic graph processing benchmarking framework (Section III). Our framework builds upon industry-grade graph processing workloads, generic and novel system-level autoscalers, and state-of-the-art elasticity metrics. To showcase the framework's ability to capture elastic graph processing behavior, we design and implement a reference elastic graph-analytics system (Section IV).
3) We explore the *benefits and cost of elasticity* through conducting an extensive performance study (Section VI). Our evaluation shows that elasticity mechanisms do not affect application throughput (our reference implementation performance is on-par with JVM-based graph-processing systems). However, migrating data for each lease or release of resources impacts the overall runtime. Furthermore, the auto-scaling policies are able to save resources otherwise wasted.

II. Variability in Graph Processing

In this section, we show evidence that the static deployment scheme of graph processing systems, when processing typical graph processing workloads, leads to variable resource consumption. If not counter-acted, for example through the elastic scaling approach proposed in this work, this variability could lead to significant resource waste or, worse, to system crashes due to under-provisioning of resources.

The de-facto procedure for quantifying imbalances in graph-processing is measuring the variability of *graph-related* metrics. Typically, when making a case for elasticity [26], [27] in graph processing, only graph-related metrics are considered, such as *active vertices* (i.e., the vertices that are exchanging messages during a super-step, which represents the graph algorithm unit of computation during a single iteration), or *messages exchanged* per super-step. Many complementary studies present an analysis on how the number of active vertices affects the algorithm efficiency. Direction-optimizing BFS [5] is a technique motivated by the authors' finding variable number of active edges, per iteration. Mizan [7] analyzes runtime per iteration, finds large imbalance. GraphReduce [28] finds large variability in the number of active vertices (frontier size) for two datasets and three algorithms. High workload imbalance appears even between multiple designs and implementations of the same algorithm running the same workload [17]. In comparison, in this section we further analyze the impact of workload imbalance on the consumption of *system-level* resources.

Main finding: We find that considering only *active vertices* as a measure for computation imbalance is insufficient, as it does not capture the impact of the graph computation at the *system-level*. Our results show that even algorithms that do not suffer from active vertices variability (e.g., Pagerank - Figure 1b) are impacted by significant system-level metrics (e.g., CPU load, memory, wallclock time) variability. The results in Figure 2 indicate the consequences can be significant imbalance in the utilization (consumption) of resources. Therefore, *active vertices* is a metric that does not translate proportionally into *system-level* metrics. This indicates there is a need for dynamic mechanisms that can elastically grow and shrink the number of active resources (e.g., machines) to run graph-processing.

Experiment: We ran a set of experiments using as input the Datagen scale factor 1000 graph (Datagen-1000). The graph was generated using the Datagen tool [29]. For our use-case, we used only the user vertices and the edges, excluding the other Datagen data (i.e., comments, posts etc.).

The workloads used are typical graph processing algorithms: breadth-first search (BFS), page-rank (PR), weakly-connected components (WCC) and single-source shortest paths (SSSP). These algorithms were run statically, on both JoyGraph reference elastic graph analytics platform (EGAP), and the well-established Giraph system [8]. We ensured that all the results are correct using the Linked Data Benchmark Council (LDBC) Graphalytics [24] validation tool. We describe in detail the design and implementation of JoyGraph EGAP in Section IV.

Metrics: To provide evidence of extensive resource variability, during each algorithm run, we measured: active vertices, CPU load, memory utilization, and wallclock time. Figure 1 plots the number of active vertices per super-step for each algorithm. It is immediately apparent that for BFS (Figure 1a) and SSSP (Figure 1d) there is a lot of room for elastically scaling based on the number of active vertices. In contrast, for PR (Figure 1b) and WCC (Figure 1c) the number of active vertices does not vary much between super-steps. Based on this metric, the only chance of reducing the number of nodes is in the final super-steps of WCC.

Analysis: When analyzing the other three metrics (CPU, memory, and wallclock time) we notice that the workloads exhibit large amounts of variability. In Figure 2, we plotted the coefficient of variation of the metrics measured per worker node per super-step. In this fashion, we capture both the imbalances of the worker nodes within a super-step, and also between super-steps. When analyzing only active vertices, one could infer that for the PR workload there is no chance to exploit elasticity. However, when considering the other metrics, we notice that even PR suffers significant resource variability. Generally, we notice that most algorithms suffer from significant resource variability. Therefore, we can conclude that by considering system metrics in addition to graph-related metrics, there is a large optimization space for applying elastic (auto)scaling policies. We provide an in-depth analysis of such policies in Section VI.

III. JoyGraph Benchmarking Framework

Policy-based elasticity for graph processing has not been explored until now. Recently, we have seen that generic autoscaling policies perform well with scientific workloads, almost on-par with workload-specific policies [21]. Encouraged by such results, we propose to evaluate applying autoscaling policies to graph processing.

In this section, we introduce the design of the JoyGraph benchmark, our framework for benchmarking elasticity in graph analytics. JoyGraph evaluates *any* elastic graph-analytics platform (EGAPs), and compares the results against a reference platform for elastic graph-analytics (see Section IV). JoyGraph is novel: the focus on elasticity distinguishes it from other graph-processing benchmarks, and the focus on graph processing distinguishes it from other elasticity benchmarks. In particular, JoyGraph introduces elasticity mechanisms that consider both application-level (i.e., active vertices), and system-level (i.e., CPU and network load, wall-clock time etc.) metrics.

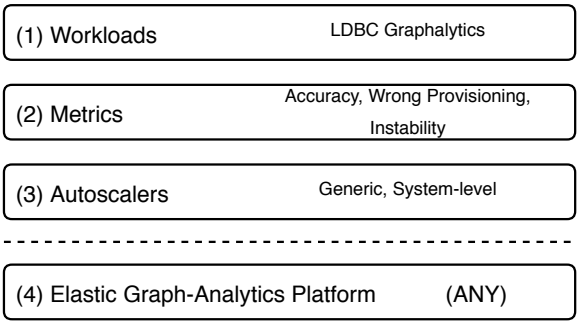


Fig. 3. Overview of the JoyGraph benchmarking framework for elastic graph-processing platforms.

A. Design Overview

Figure 3 depicts the JoyGraph benchmarking framework, which is comprised of: (1) Workloads, e.g., derived from the LDBC Graphalytics workloads [24], (2) Metrics, e.g., elasticity metrics proposed by the SPEC RG Cloud group [21] and performability metrics, (3) Autoscalers, of various type, grouped into two classes, and (4) the Harness for testing any elastic graph-analytics platform.

The JoyGraph benchmarking framework is designed to be generic and provide meaningful results for a wide range of graph analytics engines. Workloads must be representative of both industry and academia state of practice, and must be renewed periodically. Elasticity metrics should be independent of the granularity and type of resources (e.g., physical machines, virtual machines, containers, microservices), and should characterize the autoscaler quality in terms of how much the resource *supply* deviates from the workload *demand*. Finally, the autoscalers need be agnostic of the underlying resources, and minimize the latency of taking decisions to lease or release resources.

Generic graph-analytics engines offer iterative programming models, such as *Pregel* [30], *push-pull* [10], *gather-apply-scatter* [3], *sparse matrix operations* [31], in which users express general-purpose graph computations. Although generic and widely-used in both academia and industry, the systems that implement such models are not *elastic*. Therefore, to explore the costs and benefits of elasticity for graph-analytics, in this work we design a general-purpose, Pregel-based, elastic graph-analytics engine, the JoyGraph EGAP, which serves as a reference implementation of a generic EGAP.

The JoyGraph benchmarking framework serves three main purposes. First, it tries to answer the question of whether elasticity benefits graph-analytics engines, and what are its cost and drawbacks. Second, it provides a framework for in-depth comparisons of future elastic graph-analytics platforms. Third, it provides a reference EGAP implementation.

B. Benchmarking Workload

For the evaluation of EGAPs, JoyGraph uses the workloads of the LDBC Graphalytics [24] benchmark suite for

TABLE I
REAL-WORLD DATASETS IN THE LDBC GRAPHALYTICS WORKLOAD.

ID	Name	$\|V\|$	$\|E\|$	Scale	Type
R1(2XS)	wiki-talk [32]	2.39 M	5.02 M	6.9	Real-world
R2(XS)	kgs [32]	0.83 M	17.9 M	7.3	Real-world
R3(XS)	cit-patents [32]	3.77 M	16.5 M	7.3	Real-world
R4(S)	dota-league [33]	0.61 M	50.9 M	7.7	Real-world
D300(L)	datagen-300	4.35 M	304 M	8.5	Synthetic
D1000(XL)	datagen-1000	12.8 M	1.01 B	9.0	Synthetic
G22(S)	graph500-22	2.40 M	64.2 M	7.8	Synthetic
G23(M)	graph500-23	4.61 M	129 M	8.1	Synthetic
G24(M)	graph500-24	8.87 M	260 M	8.4	Synthetic
G25(L)	graph500-25	17.1 M	524 M	8.7	Synthetic
G26(XL)	graph500-26	32.8 M	1.05 B	9.0	Synthetic

(general) graph-analytics platforms. The algorithms that generate these workloads are: breadth-first search (BFS), page-rank (PR), weakly-connected components (WCC), single-source shortest paths (SSSP), local clustering coefficient (LCC), and community detection using label propagation (CDLP). For the latter, we use the label propagation algorithm described in [34].

The `JoyGraph` benchmark uses as input-graphs a subset of the datasets described in LDBC Graphalytics [24]. These datasets are categorized by the Graphalytics benchmark by their scale (order of magnitude, or *scale* of $|V| + |E|$, i.e., the approximation, through log_{10}, of the number of digits of $|V| + |E|$). For simplicity, `JoyGraph` assigns T-shirt sizes to describe its datasets, ranging from 2XS (smallest graphs) to XL (largest graphs). Table I describes the `JoyGraph` datasets. Overall, the datasets span both synthetic and real-world datasets and different orders of magnitude in scale, ranging from XS to XL. Moreover, in accordance with the work of Broido and Clauset, who recently found that scale-free networks are rare [35], `JoyGraph` includes both scale-free (Graph500) and non-scale free graphs (Datagen).

C. Elastic Autoscaling

At the core of any elastic system are its autoscaling policies, which take automatically decisions of elasticity: when, what, and where to scale. Consistently with modern design practices for large-scale distributed systems, the `JoyGraph` benchmark considers a separation of mechanism from policy, and proposes that each EGAP should be able to run any autoscaling policy, albeit, leading to different performance in practice.

The `JoyGraph` benchmark considers explicitly two classes of autoscaling policies: the set of generic autoscaling policies provided by the reference platform (introduced in Section sec:design:elasticity) and a new set of system-level policies we have designed for these experiments. (We envision that more classes will be added later, as the community matures.)

Generic Autoscaling Policies: `JoyGraph` mandates the following set, which is derived from the set of commonly used autoscalers identified by the SPEC RG Cloud group: React [36], AKTE [37], ConPaaS [38],Reg [39], Hist [40]. For a detailed description of the generic autoscalers, we refer the reader to [21].

System-level Autoscaling Policies: The metrics used for system-level autoscaling are derived from our findings presented in Section II.

- CPU Load Policy (CPU): Average worker CPU-load can be used to generate a new partition function to redistribute graph vertices as follows: the worker(s) with the highest load(s) will be distributed if $\lambda < c\sigma$. Here, $c \in (0, 1]$ is a user-defined constant, λ is the average CPU-load in a given worker, and σ is the standard deviation of the average load of all nodes.
- Wallclock Time Policy (WCP): We compute the wallclock time of each worker for each super-step. This is then used to see how much each worker deviates: workers that take more time than the average will distribute part of their vertices to new workers. The number of new workers is calculated as the $\sum 1 - \frac{W_i - \overline{W}}{\overline{W}}$, where W_i is the wallclock of worker i, and $\overline{W}$ is the average wallclock for all workers.
- Network Load Policy (NP): Load unbalances may not always be captured by wallclock time nor by memory usage. The network traffic for certain nodes may deviate from other nodes. Similar to the WCP, we compute instead the network bytes sent of each worker for each super-step: $\sum 1 - \frac{n_i - \overline{n}}{\overline{n}}$, where n_i is the network bandwidth for worker i, and $\overline{n}$ is the average bandwidth for all workers.

D. Elasticity Metrics

`JoyGraph` uses a subset of the elasticity metrics presented by the SPEC Cloud Research Group to evaluate the elastic properties of the system, grouped in three classes.

Accuracy Metrics: The accuracy metrics show how fast the system adapts to the changes in supply and demand. The values of these metrics should be as small as possible. Assuming demand at time t is d_t, and resource supply s_t, the accuracy metrics are defined as follows:

1) a_U: the average *under-provisioning accuracy* metric which measures the area in which the demand exceeds the supply and it is defined as $a_U = \frac{1}{T \cdot R} \sum_{t=1}^{T} (d_t - s_t)^+ \Delta t$, where T is the total duration of the experiment in time steps, and R is the total number of available resources, $(x)^+ = max(x, 0)$, and Δ_t is the time elapsed between two consequent measurements. (We use the formulation $(x)^+$ for defining several other metrics.)
2) a_O: the average *over-provisioning accuracy* measures the area in which the supply exceeds the demand: $a_O = \frac{1}{T \cdot R} \sum_{t=1}^{T} (s_t - d_t)^+ \Delta t$.
3) $\bar{a}_U$: the under-provisioning accuracy normalized by the actual resource demand, $\bar{a}_U = \frac{1}{T} \sum_{t=1}^{T} \frac{(d_t - s_t)^+}{d_t} \Delta t$. The normalized accuracy is particularly useful when the resource demand has a large variance over time, and it can assume both large and small values.
4) $\bar{a}_O$: the normalized over-provisioning accuracy, defined as $\bar{a}_O = \frac{1}{T} \sum_{t=1}^{T} \frac{(s_t - d_t)^+}{d_t} \Delta t$.

Wrong-Provisioning Timeshare Metrics: While the accuracy metrics express the elastic properties of the system in the number (or fraction) of over- or under-provisioned resources, they do not consider how much time the system spends in such states. To distinguish between short but large deviations from long but small deviations, we use *wrong-provisioning*

timeshare metrics t_U and t_O. These two complementary metrics represent the fraction of time in which under- or over-provisioning occurs:

1) $t_U = \frac{1}{T}\sum_{t=1}^{T}(sign(d_t - s_t))^+\Delta t$, where $sign(x)$ is the sign function of x and $(x)^+$ is defined as for a_U.
2) $t_O = \frac{1}{T}\sum_{t=1}^{T}(sign(s_t - d_t))^+\Delta t$.

Instability Metrics: The metrics k and k' capture instability and inertia of autoscalers. Under the assumption that the policy is trying to reach an equilibrium where supply equals demand, a low stability indicates that the system adapts fast to changing supply and demand, and a high stability indicates that the system adapts slowly. For a low complementary stability, it means that the system is fast in removing excessive nodes as demand is decreasing.

1) k: This metric shows the fraction of time the supply and demand curves move in opposite directions $k = \frac{1}{T-1}\sum_{t=2}^{T} min((sign(\Delta s_t) - sign(\Delta d_t))^+, 1)\Delta t$, where $(x)^+$ is defined as for a_U.
2) k': This complementary metric shows the fraction of time the curves move towards each other $k' = \frac{1}{T-1}\sum_{t=2}^{T} min((sign(\Delta d_t) - sign(\Delta s_t))^+, 1)\Delta t$.

E. Performability Metrics

Assessing the performance of graph-processing workloads is not a straightforward endeavor. Whilst in high-performance computing domain experts measure FLOPs or bandwidth, deriving such numbers when running graph workloads is both difficult to achieve, and not trivially translated into graph operations throughput. Therefore, in this article we refer to *performability* as a measure of various graph-processing related metrics gathered over a period of time. `JoyGraph` defines the following metrics to evaluate the performability of (elastic) graph-processing workloads:

1) t_p: the processing time of a job in seconds. This is the sum of time spent in every super-step and barrier. It includes time spent on adding and removing resources, but excludes everything else (i.e., loading input, or storing output).
2) t_m: the makespan of a job in seconds. This is the time spent from the submission of the job, until its completion. It includes fetching and provisioning machines, loading input data, running the algorithm, and the time to generate and persist the output.
3) t_e: the elasticity overhead of a job in seconds. This is the time spent, for all super-steps to lease or release resources and migrate data according to the neste partitioning scheme. This metric takes into account parallel data transfers and addition/removal of resources.
4) $\sum t_c$: the total lifespan of all separate node lifespans t_c in seconds, where t_c is the time difference between the start of a process in a node until the release of the node. This metric allows to measure the real cost of running a specific job expressed in node time. The real cost can then be extended to quantify the performance of combinations of graph algorithms and autoscaling policies with respect to a dataset.
5) *VPS + EVPS*: VPS is the graph processing speed in *Vertices Per Second* and *EVPS* is the graph processing speed in *Edges and Vertices Per Second*.
6) $\sum t_e$: the cumulative of overhead in seconds introduced by elasticity for all the used nodes, where the elastic overhead t_e is the time a single node spends waiting for data transfers induced by addition or removal of nodes. This metric does not take into account parallel data transfers and addition/removal of resources.
7) $\sum t_s$: the total time spent processing all super-steps, where t_s is the time spent processing a single super-step (the generation of outgoing messages and the processing of incoming messages) by a set of nodes.

IV. JoyGraph EGAP: A Reference Elastic Graph-Analytics Platform

In this section, we discuss the `JoyGraph EGAP`, which is an elastic graph-analytics platform. Currently, the community does not have a reference EGAP, or even an open-source EGAP it can use for benchmarking purposes. To explore the benefits and drawbacks of elastic graph processing, we design and implement, and include in the `JoyGraph` benchmark, a *reference* EGAP: a fully-elastic graph processing system that can lease or re-lease resources, on-demand, at runtime. The goal of this EGAP is not to optimize performance under elasticity, but to serve as a benchmark for comparisons, stimulating the community to do better.

A. Design Overview

We design the `JoyGraph EGAP` as an iterative GAP, based on the commonly used Pregel-like programming model. The key elasticity mechanism is to grow and shrink the provisioned infrastructure, using the decisions to lease and release resources taken by any of the elasticity policies mandated by `JoyGraph` (described in Section III-C). Figure 4 presents an overview of the `JoyGraph EGAP`, with the following main components:

(1) Master: The *master* node orchestrates the execution of the graph processing workloads on the *worker* nodes. It collects utilization metrics from the workers and takes elastic scaling decisions based on such metrics. Furthermore, whenever a scaling event is triggered, it adjusts the partitioning scheme. Also, using the cluster/cloud resource manager, it is able to provision or decommission resources. The granularity of taking elastic scaling decisions is defined at super-step level, i.e., whenever a super-step finishes. However, `JoyGraph EGAP` could take finer-grained elastic scaling decisions, although in practice we found this to add too much overhead, as generally the super-steps are short-lived (i.e., in the order of seconds, or tens of seconds).

(2) Workers: Worker nodes execute the super-steps of the graph processing workloads. To enable elastic scaling, they collect local metrics (e.g., CPU load, memory utilization, active vertices etc.) which are sent to the master at regular time intervals (i.e., every second). Worker nodes store in memory the *partitions* of the input graph. Whenever an

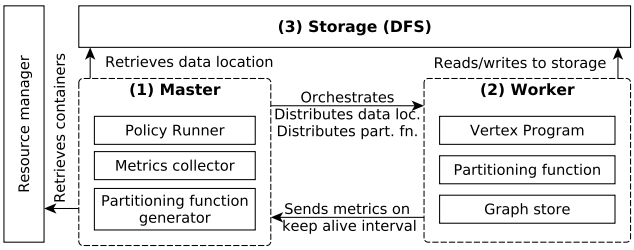


Fig. 4. Overview of the `JoyGraph EGAP` architecture.

elastic scaling event is triggered, the worker nodes receive from the *master* the updated partitioning scheme. Based on this new *vertices-to-workers* mapping, the worker nodes send data to other nodes, to facilitate load balancing when workers are added or removed.

(3) Storage: JoyGraph uses a distributed or shared file system to load the input graph and to store the generated results. The choice of storage system does not affect application performance, as the graph-processing system does not interact with it during the runtime of the graph algorithms. In our experiments (Section VI), the storage system we use is a NFS installation.

B. Elastic Workload Distribution and Nested Partitioning

Graph partitioning is a well-studied field [4], [7], [41]. Although well-balanced partitions (i.e., in terms of storage, workload distribution, network load) are challenging to design on their own, partitioning efficiency can raise additional design challenges: performance decreases even more when a graph is altered, or when workers are added and/or removed at runtime.

Graph partitioning is often achieved using a *hash partitioning* function over the vertex and worker identifiers, e.g., $h(v) = v \mod n$, where n is the number of workers. In JoyGraph, we address partitioning imbalances and robustness while supporting elasticity by means of *nested partitioning*. A *nested hash partitioning* mapping $h_1(vertex) \rightarrow worker$ consists of two parts: (i) a hash-partitioning function $h_0(v) \rightarrow m$, where h_0 is a hash partitioner starting from vertex v and m is a worker-machine, and (ii) a series of mappings $S_i \rightarrow (\{D_i\}, f_i)$, where S_i is the source-machine of the mapping, D_i is the destination-machine of the mapping, and f_i is the fraction of vertices to be distributed from S_i to D_i. Load-balancing can be achieved by carefully choosing f_i, which is done by the *master* (see Section IV-A).

To illustrate how the nested hash partitioning handles elasticity, consider the following example. A workload L is running on JoyGraph on n worker-nodes. At the end of super-step k, the *master* computes how balanced the system is in terms of metric M (i.e., CPU, memory, network, active vertices). If, according to policy P, a scaling event needs to be triggered by adding n_+ worker-nodes, the *master* computes a new *nested partitioning* scheme: for each worker node m_i, a fraction of vertices that need to be migrated, f_i, is computed. Then, the new partitioning scheme is broadcast to all nodes. Based on this new partitioning scheme, the workers transfer vertices to the newly added nodes. The procedure is similar when decreasing the number of nodes.

TABLE II
SELECTED GRAPH ANALYSIS PLATFORMS. ACRONYMS: C, COMMUNITY-DRIVEN; I, INDUSTRY-DRIVEN; D, DISTRIBUTED; S, NON-DISTRIBUTED.

Type	Name	Vendor	Lang.	Model	Vers.
C, D	Giraph [8]	Apache	Java	Pregel	1.1.0
C, D	GraphX [9]	Apache	Scala	Spark	1.6.0
C, D	PowerGraph [3]	CMU	C++	GAS	2.2
I, S/D	GraphMat [31]	Intel	C++	SpMV	Feb '16
I, S	OpenG [42]	G.Tech	C++	Native code	Feb '16
I, D	PGX.D [10]	Oracle	C++	Push-pull	Feb '16

V. EXPERIMENT SETUP

In this section, we present the design of our experiments with EGAPs. We use the `JoyGraph` benchmark to study the effects of elasticity in graph processing, in two experiments:

(1) Non-elastic baseline: we analyze the performance of the reference `JoyGraph EGAP` without elasticity enabled, in comparison with state-of-the-art graph processing platforms (all of which use static-provisioning, so are non-elastic).

(2) Impact of elasticity on performance: we perform an in-depth analysis of the performance-effects of auto-scaling policies, which determine, at runtime, the amount of nodes used by the `JoyGraph EGAP` to run graph-processing workloads.

Implementation and experimentation effort: We have implemented a prototype of the `JoyGraph` reference EGAP in Scala, using approximately 11K LoC, in 4 person-months. We have used for it the generic autoscaling policies provided by the SPEC RG Cloud Group, and implemented the new system-level policies. We have conducted experiments, first for testing purposes and later full-scale, in another 3 person-months.

A. Systems Under Test: Two Classes of Elastic and Four Classes of State-of-the-Art Graph-Processing Systems

We equip the reference `JoyGraph EGAP` with the two sets of autoscaling policies introduced in Section III-C, generic and systems-level. This effectively allows us to evaluate two classes of EGAPs.

We compare EGAPs with the selection of (non-elastic, generic) GAPs summarized in Table II: 6 state-of-the art GAPs systems, grouped in 4 classes. The class of GAPs that are community-driven and distributed (*C,D*) is the most represented, in part because these are open-source systems available to everyone. For all generic GAPs, the results we report here are taken from the LDBC Graphalytics study [24]. For a more detailed description of these state-of-the-art platforms, we refer the reader to the same study.

B. Hardware and Software Environment for Experiments

The results presented in this section are generated by conducting real-world experiments on one of the clusters of DAS-5, a multi-cluster system designed for computer-science experiments [43] and extended in 2016 to fit big data experimentation. The infrastructure is exclusively reserved for

our experiments, that is, we report results obtained without the interference of a background workload.

Our infrastructure consists of 68 compute nodes equipped with dual 8-core Intel E5-2630v3 (two hyperthreads per core) CPUs, 64 GB memory, and 4 TB HDD. The nodes are interconnected with a 54-Gbps FDR InfiniBand. The cluster nodes are running CentOS 7.2, Linux Kernel 3.10. The cluster manager used for leasing and releasing resources is SLURM. The JVM used to run the JVM-based platforms, including `JoyGraph EGAP`, is OpenJDK 1.8.

VI. Results using the JoyGraph Benchmark

In this section we present an in-depth analysis on the performance of the elastic prototype for graph-processing, JoyGraph, according to the experimental setup presented in Section V. Our main findings can be summarized as follows:

MF1 Elasticity does not degrade application throughput: the performance of `JoyGraph EGAP` and of JVM-based graph-processing systems is comparable.

MF2 `JoyGraph EGAP` can run non-trivial graph-processing workloads (e.g., LCC), where several state-of-the-art platforms fail.

MF3 Elastic scaling policies are able to exploit non-trivial (e.g., wallclock time, CPU, network) resource variability to improve resource management and load balancing.

MF4 Elasticity introduces a high communication overhead when leasing or releasing resources.

MF5 Elasticity is able to reduce resource utilization provided that communication time is reduced by at least 50%.

MF6 The interaction between the autoscaling policy and the graph-analytics workload is non-trivial. Consequently, elasticity adds a new level of complexity in resource management and scheduling for graph processing, which requires further, more in-depth study.

A. `JoyGraph EGAP` *vs. State-of-the-Art Graph-Processing Systems*

In this section we assess the behavior of the `JoyGraph EGAP` under various scenarios: dataset variety, algorithm variety, strong horizontal scalability, and weak horizontal scalability. The results obtained with this set of experiments support **MF1**.

Dataset Variety. The performance of the `JoyGraph EGAP` is comparable to the JVM-based graph-processing platforms. We ran the BFS and PR algorithms on 6 different graphs of increasing sizes on 1 compute node. We compare the performance of `EGAP` with Giraph [8], GraphX [9], Powergraph [3], Graphmat [31] (both single node and distributed), OpenG [42], Oracle PGX and its distributed counterpart, PGX.D [10]. Figure 5 presents the results of this experiment. We report the full makespan time, that includes loading the graph, running the algorithm, and saving the results. The results show clearly that `JoyGraph EGAP` performance is comparable to the JVM-based graph processing platforms (Giraph, GraphX), while the native platforms exhibit better performance, as expected.

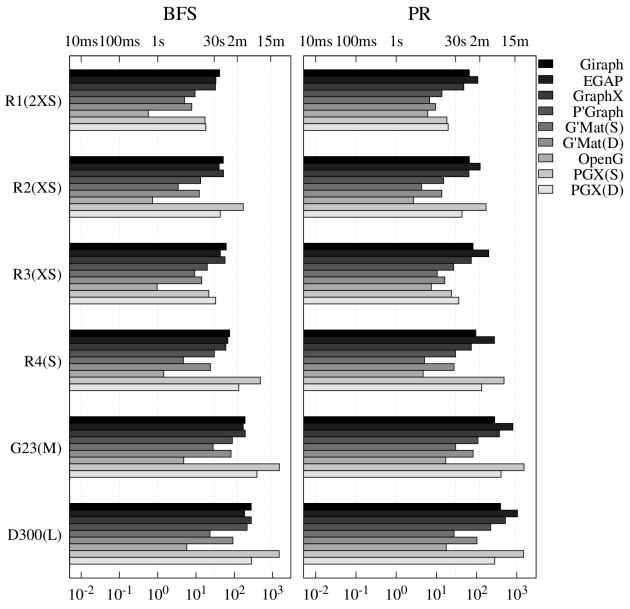


Fig. 5. Dataset variety: Dataset impact on makespan, for BFS and PR.

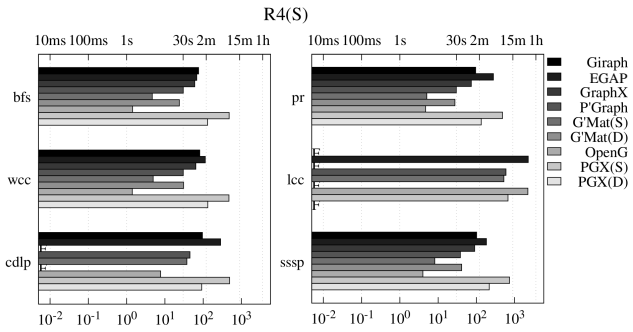


Fig. 6. Algorithm variety: algorithm makespan on R4.

Algorithm Variety. `JoyGraph EGAP` is able to run non-trivial graph-processing workloads (i.e., LCC), while state-of-the-art platforms fail (**MF2**). We ran all the LDBC Graphalytics algorithms (BFS, WCC, CDLP, PR, LCC and SSSP) on the R4 graph. We compare `EGAP` against the platforms mentioned previously on one compute node. Figure 6 plots the results. For most algorithms, `EGAP` performance is comparable to the JVM-based platforms (Giraph, GraphX) and comparable to PGX and PGX.D on most algorithms. `EGAP` is able to run LCC, as opposed to Giraph, GraphX, GraphMat (distributed) and PGX.D on the R4 graph.

Strong Horizontal Scalability. `JoyGraph EGAP` exhibits good horizontal scaling behavior. We performed a strong horizontal scalability analysis for all graph processing platforms on the D1000 graph while running BFS and PR. The platforms are deployed on 1-2-4-8-16 nodes. Figure 7 plots the results. We notice that the best-performing platforms are GraphMat and PGX.D. These platforms are not only the fastest, but also exhibit a good scaling behavior. In contrast, `EGAP` performance is comparable only to Giraph and GraphX. In our environment, GraphMat crashed on 4 machines due to an unresolved issue in the used MPI implementation.

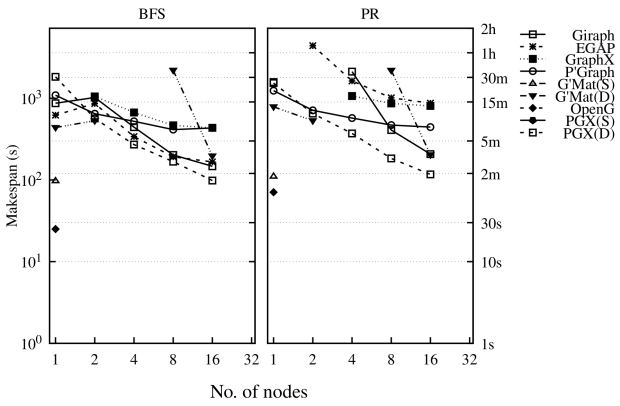


Fig. 7. Strong Horizontal Scalability: BFS and PR makespan on D1000 vs. number of nodes.

B. Elastic Graph Processing

We present the performance evaluation of the autoscaling policies using the elasticity metrics introduced in Section V. Our experiments are performed on the graph500-25 dataset using the BFS, and PR workloads. We start all the elastic runs on 4 worker nodes. During runtime, the minimum allowed number of worker nodes is 1, while the maximum is 20. We compare all the elastic runs with static baseline experiments that runs on 4, 10, and 20 machines. The results obtained in this set of experiments support **MF3**.

Elastic BFS Highlights. Table III presents the performance metrics for all autoscaling policies while running the BFS algorithm. For the same experiment, Table IV presents the autoscaling metrics derived from the performance metrics.

The processing time and makespan vary significantly between the autoscaling policies. The worst performing autoscalers are the generic ones, while the graph-specific policies leverage better running times. We notice that the generic autoscalers do spend little time under-provisioned, but also that the over-provisioned time is relatively high. This indicates that *scaling out* is cheap, but *scaling in* is more expensive. The graph-specific policies do not over-provision much, but the under-provisioning time is relatively high. In terms of stability, all policies exhibit a highly unstable behavior. This can be contributed to the BFS behavior: the number of active vertices increases significantly in the beginning, leading to a constant increase in demand; towards the end of the algorithm, the number of active vertices decreases significantly, leading to a large decrease in demand.

Elastic PR Highlights. Table V presents the performance metrics for all autoscaling policies while running the PR algorithm. For the same experiment, Table VI presents the autoscaling metrics derived from the performance metrics.

The autoscaler processing time and makespan do not vary as much as for BFS. As opposed to BFS behavior, the worst performing autoscalers are the graph-specific policies. It is important to notice that in the case of PR, only three autoscaling policies generate elastic scaling events: React, AKTE and WCP. The others do not trigger any elasticity due to the PR characteristics: all vertices are active during the entire duration of the algorithm. Contrary to the BFS behavior, no policy over-provisions. The three policies that do trigger elastic scaling (i.e., React, AKTE, and WCP) exhibit a large amount of under-provisioning, and also instability.

TABLE III
PERFORMANCE METRICS FOR GRAPH500-25 AND ALGORITHM BFS.

	t_p	t_m	t_e	$\sum t_c$(s)	kVPS	kEVPS	$\sum t_e$(s)	$\sum t_s$(s)
Static-20	29	76	0	1,316	576	35,938	0	329
Static-10	47	104	0	1,008	359	22,409	0	329
Static-4	147	300	0	1,445	115	7,230	0	506
React	525	675	461	6,473	32	2,027	3,345	324
AKTE	2,106	2,258	1,980	15,630	8	505	10,499	324
ConPaaS	957	1,116	872	10,882	17	1,112	6,950	327
Reg	701	862	618	10,454	24	1,526	7,268	313
Hist	563	708	500	7,545	30	1,890	4,986	316
NP	433	583	254	3,399	60	3,784	621	518
CPU	445	593	272	3,346	38	2,391	396	552
WCP	747	898	551	5,878	23	1,496	998	525

TABLE IV
AUTOSCALER METRICS FOR GRAPH500-25 AND ALGORITHM BFS.

	a_U	a_O	$\bar{a}_U$	$\bar{a}_O$	t_U	t_O	k	k'
React	0.11	1.27	0.00	0.82	0.12	21.19	36.98	58.95
AKTE	0.06	1.01	0.00	20.25	0.28	6.27	60.72	38.18
ConPaaS	0.07	2.02	0.00	38.52	0.07	12.79	34.94	62.59
Reg	0.12	2.71	0.00	54.21	0.13	16.93	44.82	51.18
Hist	0.13	2.91	0.00	58.18	0.14	19.39	52.36	43.61
NP	0.22	0.02	0.00	0.00	4.40	0.36	24.67	18.21
CPU	1.16	0.00	0.00	0.00	23.23	0.00	49.64	23.25
WCP	0.97	0.55	0.00	0.00	19.47	10.90	59.35	39.53

TABLE V
PERFORMANCE METRICS FOR GRAPH500-25 AND ALGORITHM PR.

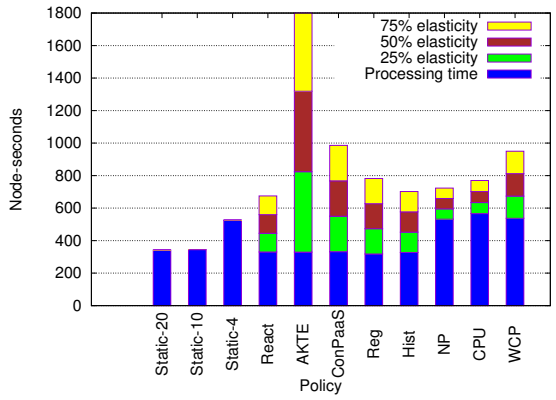
	t_p	t_m	t_e	$\sum t_c$(s)	kVPS	kEVPS	$\sum t_e$(s)	$\sum t_s$(s)
Static-20	186	236	0	4,668	91	5,712	0	3,094
Static-10	385	443	0	4,734	44	2,764	0	3,080
Static-4	1,327	1,483	0	7,355	12	801	0	5,033
React	1,107	1,253	514	13,548	15	961	5,110	3,775
AKTE	1,116	1,268	529	13,710	15	953	5,248	3,793
ConPaaS	1,334	1,488	0	7,378	12	797	0	4,958
Reg	1,302	1,460	0	7,236	13	817	0	4,951
Hist	1,395	1,551	0	7,699	12	762	0	5,104
NP	1,308	1,462	0	7,246	13	814	0	4,967
CPU	1,294	1,452	0	7,199	13	822	0	4,912
WCP	1,660	1,814	457	13,712	10	641	1,014	4,730

TABLE VI
AUTOSCALER METRICS FOR GRAPH500-25 AND ALGORITHM PR.

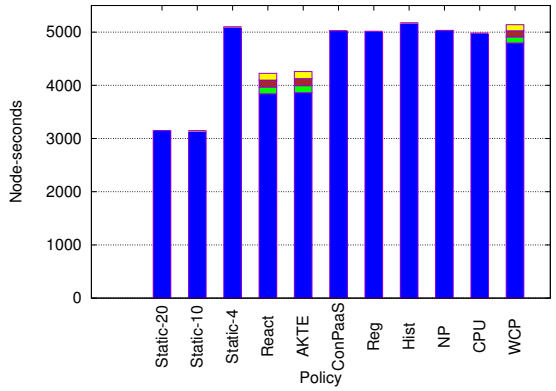
	a_U	a_O	$\bar{a}_U$	$\bar{a}_O$	t_U	t_O	k	k'
React	4.46	0.00	0.00	0.00	44.64	0.00	53.33	45.26
AKTE	4.37	0.00	0.00	0.00	43.72	0.00	54.20	44.32
ConPaaS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Reg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hist	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CPU	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
WCP	2.25	0.00	0.00	0.00	39.18	0.00	30.16	40.23

C. The Cost and Benefits of Elasticity

With the reference implementation, moving data when leasing or releasing resources is a synchronous mechanism: computation is stopped until the the data is migrated. This generates large amounts of (communication) overhead for the auto-scaling policies that trigger many scaling events (**MF4**). This behavior is depicted in the column t_e of Table III and Table V for BFS and PR, respectively. In Figure 8, we plot the resource utilization for BFS (Figure 8a) and PR (Figure 8b) in *node-seconds*, i.e., the amount of time `EGAP` used its resources. Furthermore, for the autoscalers, we depict the processing time and the 25th, 50th, and 75th percentiles of elasticity time. Provided that the data migration mechanism is further optimized, and reduced by at least 50%, the resource utilization for both BFS and PR can be improved



(a) BFS Resource Utilization.



(b) PR Resource Utilization.

Fig. 8. `JoyGraph EGAP`: the cost of elasticity.

via elastic scaling policies (**MF5**). This could be achieved via non-blocking [44], or hot-migration [45] data movement mechanisms.

Moreover, as static processing scalability is not linear (e.g., Static-20, and Static-10 use a similar amount of resources), as a user, or system administrator it is not trivial to determine an optimal configuration. Figure 8 shows that the general autoscaling policies achieve a similar resource utilization to the static deployments on BFS, and a competitive resource utilization for PR. To conclude, all general autoscalers achieve a good processing time for BFS, while for PR only React and AKTE. The NP, CPU and WCP policies are too conservative, and trigger too few elastic scaling events, thus processing the workload on lower numbers of resources, which reduces the throughput.

D. Toward Uncovering Elasticity Laws in Graph-Processing

We showed in Section II that there is more non-trivial resource variability to exploit than simply using *active vertices*, as state-of-the-art suggests. We showed that even though the number of active vertices is constant during runtime (i.e., as is the case of PR), the usage of systems-level metrics can highly vary. The opposite is also true, as is the case of BFS: while the active vertices show significant variability, the wallclock time, or memory do not vary as much. We validate this behavior by plotting the supply-demand curves of BFS and PR under the WCP and AKTE autoscaling policies. Figure 9 depicts the results.

For the BFS workload, while the AKTE policy (Figure 9b) scales abruptly from 4 to 20 nodes and back, there is not much variability after super-step 5. Moreover, the WCP policy (Figure 9a) does not even identify a large increase in demand, scaling only up to 8 nodes. For the PR algorithm, both WCP (Figure 9c) and AKTE (Figure 9d) identify large increases in demand and trigger many scaling events, even though the PR algorithm does not generate any variability in the number of active vertices during runtime.

This shows that we are far from completely understanding the behavior of graph-processing workloads and systems, supporting **MF6**. There is much intrinsic resource variability due to both algorithm and system behaviors that we need to harness via fine-grained autoscaling mechanisms and policies to build more scalable, load balanced, and performant systems.

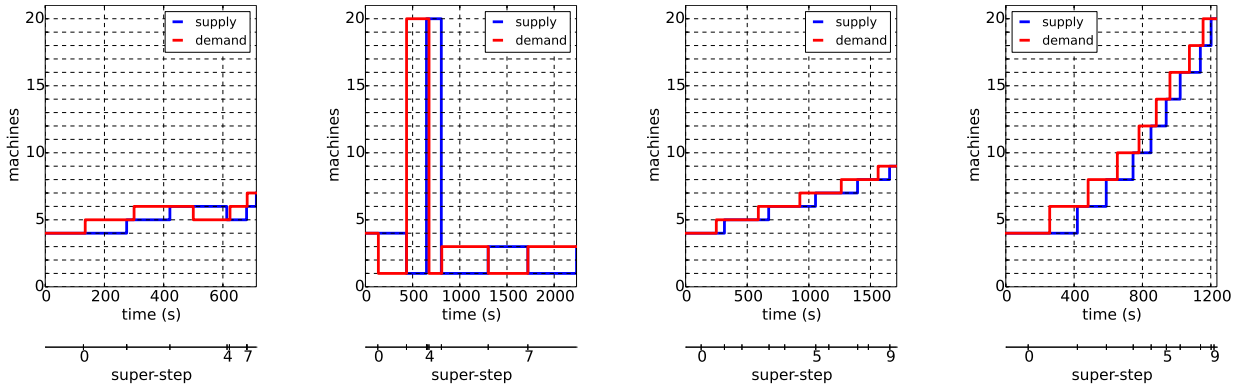
Similarly to our conjecture, in the past, that the performance of graph-processing systems depends non-trivially on datasets, algorithms, and underlying execution platform [24], we conjecture from the data presented in this section that uncovering the laws that govern elasticity in graph-processing systems is also non-trivial. Proving this conjecture is outside the scope of this work.

VII. Related Work

In this section we discuss work related to `JoyGraph`: benchmarks for graph processing, and three systems-related categories, cloud-optimized graph processing, dynamic partitioning, and general elastic data processing frameworks.

a) Benchmarks for Graph Processing: Closest to our work, LDBC Graphalytics [24] is a state-of-the-art benchmark supported by the industry and academic community processing graphs on commodity hardware. Extending LDBC Graphalytics significantly, `JoyGraph` focuses on elasticity, proposing new metrics, elasticity policies, and a reference `JoyGraph EGAP`. `JoyGraph` proposes the same features in contrast with Berkeley GAP [46], or Graph500, which is currently the de facto standard for benchmarking GAPs running on supercomputing hardware; `JoyGraph` uses a more diverse set of algorithms and datasets than Graph500, also adhering to the recent finding of Broido and Clauset: most real-world graphs are not scale-free [35].

b) Cloud-optimized Graph Processing: Recent work on graph processing proposes adapting such systems to the cloud: Surfer [16] Pregel.NET [26], and iGiraph [27]. Also, Li et al. introduce a performance model [47] for Pregel-like distributed graph processing on clouds. This predictor suggests cost effective static VM deployment with appropriate VM instance types. Such systems focus on enabling cost-effective deployments on clouds while using static setups. In contrast, JoyGraph takes a more general approach, being infrastructure agnostic. Its elasticity policies can be extended with cost-effective scaling policies and network-aware partitioning to reduce cost and improve performance in clouds. Furthermore,



(a) BFS + WCP. (b) BFS + AKTE. (c) PR + WCP. (d) PR + AKTE.

Fig. 9. EGAP elastic policies WCP and AKTE on BFS and PR. The position of the discrete super-steps reflects the duration of the elasticity operations.

JoyGraph's native elasticity enables it to take decisions dynamically, during runtime.

c) Dynamic Partitioning: We identified several graph processing systems [7], [48], [49] that dynamically repartition the graph data during runtime. The goal of such systems is to improve performance through reducing the imbalances of either storage, network communication, or computation. Much like JoyGraph, such systems monitor various system metrics during runtime (e.g., active vertices, CPU load, network communication) and take dynamic re-partitioning decisions during runtime. In addition to achieving load balancing through re-partitioning, JoyGraph is also able to scale the number of workers to improve resource utilization.

d) Elastic Data Processing Systems: The problem of elasticity in generic data processing systems has been extensively treated recently. Elasticity has been leveraged to meet on-demand workload variability, reduce monetary costs or improve resource utilization. For achieving this, variants of elastic MapReduce [50], [51] have been developed. Furthermore, more domain-specific elasticity schemes have been proposed: machine learning on Spark [52], elastic stream processing [53] or even scientific workflows [22]. Similarly to such systems, JoyGraph supports user-defined policies for dynamically scaling the number of compute nodes during runtime. Closest to our work, Pundir et al. investigate the challenges of elastically scaling graph computation during runtime [44]. However, they only provide mechanisms for migrating data when adding or removing nodes, and do not investigate any elastic policy.

VIII. Conclusion and Ongoing Work

Graph-processing systems currently use a static deployment model: for each processing job, a set of resources is used from start to finish. In contrast, in this work we investigated a dynamic deployment model. Our benchmark, JoyGraph proposes a framework for assessing the benefits and costs of elasticity in graph processing. JoyGraph builds on top of the LDBC Graphalytics industry benchmark for graph processing, and the SPEC Cloud Group's elasticity metrics, and proposes a reference EGAP.

We find that elasticity offers an interesting trade-off between performance and fine-grained resource management in comparison to the static state-of-the-art alternatives. We characterize with many elasticity-related metrics the performance of the autoscaling policies and find that they offer distinct trade-offs. Moreover, we show that graph processing workloads are sensitive to data migration during elastic scaling, and that the non-trivial interactions between scaling policies and workload add an extra level of complexity to resource management and scheduling for graph processing.

We are just beginning to understand the potential elasticity holds for graph processing, and for big data processing in general. We are currently extending this work in the following directions: we are exploring the situation when elasticity becomes particularly challenging when multiple customers compete for the same infrastructure. We are considering re-evaluating and extending JoyGraph for a more extensive set of (emerging) graph-processing algorithms, addressing property graphs, mutable graphs, and (dynamic) graph-processing workflows.

Reproducibility through Code and Data Availability

Following a decade-long tradition in our research group, of releasing data and source-code, the artifacts of this study are available online, at https://atlarge.ewi.tudelft.nl/gitlab/stau/joygraph

Acknowledgments

This work is supported by the Dutch projects Vidi MagnaData, by the Dutch Commit and the Commit project Commissioner, and by generous donations from Oracle Labs, USA.

References

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *PVLDB*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *SIGKDD*, 2012, pp. 1222–1230.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[4] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema, "Modeling, analysis, and experimental comparison of streaming graph-partitioning policies," *JPDC*, vol. 108, pp. 106–121, 2017.

[5] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing breadth-first search," in *SC*, 2012, p. 12.

[6] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *ICDE*, 2013, pp. 553–564.

[7] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *EuroSys*, 2013, pp. 169–182.

[8] A. Ching and C. Kunz, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Hadoop Summit*, vol. 6, no. 29, 2011.

[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[10] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi, "PGX.D: a fast distributed graph processing engine," in *SC*, 2015, pp. 58:1–58:12.

[11] F. Checconi and F. Petrini, "Massive data analytics: the Graph 500 on IBM Blue Gene/Q," *IBM J. Res. Dev.*, vol. 57, no. 1/2, pp. 10–1, 2013.

[12] Y. Guo, A. L. Varbanescu, D. H. J. Epema, and A. Iosup, "Design and experimental evaluation of distributed heterogeneous graph-processing systems," in *CCGRID*, 2016, pp. 203–212.

[13] S. Heldens, A. L. Varbanescu, and A. Iosup, "Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms," in *Int'l. W. on Irregular Applications: Archi. and Algo.*, 2016, pp. 62–65.

[14] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," in *SIGMOD*, 2010, pp. 1123–1126.

[15] J. Li, S. Su, X. Cheng, Q. Huang, and Z. Zhang, "Cost-conscious scheduling for large graph processing in the cloud," in *HPCC*, 2011, pp. 808–813.

[16] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *SoCC*, 2012, p. 3.

[17] M. Verstraaten, A. L. Varbanescu, and C. de Laat, "Quantifying the performance impact of graph structure on neighbour iteration strategies for pagerank," in *Euro-Par Workshops*, 2015, pp. 528–540.

[18] 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, pp. 395–404.

[19] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *ICDCS*, 2011, pp. 559–570.

[20] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *SC*, 2011, pp. 1–12.

[21] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *ICPE*, 2017, pp. 75–86.

[22] A. Uta, A. Sandu, S. Costache, and T. Kielmann, "MemEFS: an elastic in-memory runtime file system for escience applications," in *e-Science*, 2015, pp. 465–474.

[23] B. Nicolae, P. Riteau, and K. Keahey, "Bursting the cloud data bubble: Towards transparent storage elasticity in iaas clouds," in *IPDPS*, 2014, pp. 135–144.

[24] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson *et al.*, "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," *PVLDB*, vol. 9, no. 13, pp. 1317–1328, 2016.

[25] S. Au, A. Uta, A. Ilyushkin, and A. Iosup, "An elasticity study of distributed graph processing," in *CCGrid*, 2018.

[26] M. Redekopp, Y. Simmhan, and V. K. Prasanna, "Optimizations and analysis of BSP graph processing models on public clouds," in *IPDPS*, 2013, pp. 203–214.

[27] S. Heidari, R. N. Calheiros, and R. Buyya, "iGiraph: A cost-efficient framework for processing large-scale graphs on public clouds," in *CCGRID*, 2016, pp. 301–310.

[28] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: processing large-scale graphs on accelerator-based systems," in *SC*, 2015, pp. 28:1–28:12.

[29] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The LDBC social network benchmark: Interactive workload," in *SIGMOD*. ACM, 2015, pp. 619–630.

[30] G. Malewicz, M. Austern, and A. Bik, "Pregel: a system for large-scale graph processing," *SIGMOD*, pp. 135–146, 2010.

[31] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *PVLDB*, vol. 8, no. 11, pp. 1214–1225, 2015.

[32] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," `snap.stanford.edu/data`.

[33] Y. Guo and A. Iosup, "The game trace archive," in *NETGAMES*, 2012.

[34] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, no. 3, p. 036106, 2007.

[35] A. D. Broido and A. Clauset, "Scale-free networks are rare," *arXiv preprint arXiv:1801.03400*, 2018.

[36] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE, 2009, pp. 281–286.

[37] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012, pp. 204–212.

[38] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 195–204.

[39] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[40] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, p. 1, 2008.

[41] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *SOSP*, 2013, pp. 472–488.

[42] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *SC*, 2015, pp. 1–12.

[43] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[44] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell, "Supporting on-demand elasticity in distributed graph processing," in *IC2E*, 2016, pp. 12–21.

[45] A. Uta, O. Danner, C. van der Weegen, A. Oprescu, A. Sandu, S. Costache, and T. Kielmann, "Memefs: A network-aware elastic in-memory runtime distributed file system," *Future Generation Comp. Syst.*, vol. 82, pp. 631–646, 2018.

[46] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[47] Z. Li, B. Zhang, S. Ren, Y. Liu, Z. Qin, R. S. M. Goh, and M. Gurusamy, "Performance modelling and cost effective execution for distributed graph processing on configurable VMs," in *CCGRID*, 2017, pp. 74–83.

[48] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *SIGMOD*, 2012, pp. 517–528.

[49] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Int'l. Conf. on Sci. and Stat. Data. Mgmt.* ACM, 2013, p. 22.

[50] Z. Fadika and M. Govindaraju, "Delma: Dynamically elastic MapReduce framework for cpu-intensive applications," in *CCGRID*, 2011, pp. 454–463.

[51] A. Gandhi, S. Thota, P. Dube, A. Kochut, and L. Zhang, "Autoscaling for hadoop clusters," in *IC2E*, 2016, pp. 109–118.

[52] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: agile ml elasticity through tiered reliability in dynamic resource markets." in *EuroSys*, 2017, pp. 589–604.

[53] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *SoCC*, 2015, pp. 276–287.