Leveraging Transient Resources for Incremental Graph Processing on Heterogeneous Infrastructures.

Extended Abstract of MSc Dissertation

Pedro Joaquim
Instituto Superior Técnico
Lisboa, Portugal
Advisor: Professor Luís Rodrigues

ABSTRACT

Incremental graph processing as emerged as a key technology, with applications in many different areas. These applications, regardless of their differences, have the common characteristic of continuous processing, thus requiring the system to run on a 24/7 basis. For deployments in public cloud infrastructures, these long term deployments may incur very high costs. However, cloud providers offer transient lived resources, with significantly reduced prices compared to the regular resources, with the proviso that the former can be evicted at any time. Thus, this type of resources is a huge opportunity to significantly reduce operational costs, missed so far by existing approaches.

In this work we present Hourglass, a deployment engine for existing incremental graph processing systems. The system aims at leveraging the resource heterogeneity in public cloud infrastructures, as well as transient resource markets, to significantly reduce operational costs. Hourglass exploits features of the workload, such as the imbalance in vertex communication patterns, to do a resource-aware distribution of graph partitions among the existing machines. By constantly analyzing transient market conditions, it is able to select the cheapest deployment configuration in each moment, allowing a periodic reconfiguration of the system. Knowledge about the computation being performed allows Hourglass to choose the right moment to transition between deployments. This allows the underlying system to operate on a fraction of the cost of a system composed exclusively of reserved machines, with minimal impact in performance.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Graph Processing, Cloud Computing, Resource Management.

1 INTRODUCTION

The growth of graph data, including social networks, web graphs, and biological networks, has driven the emergence of many specialized graph processing frameworks [6, 7, 14, 15]. These systems simplify the implementation of iterative graph algorithms by exposing graph-specific abstractions. Early systems, such as Google's Pregel [15], assumed an underlying static graph structure while the computation is performed. However, many domains of interest, such as commerce, advertising, and social relationships, are highly dynamic and consequently, the graphs that model these domains present a fast-changing structure that needs to be constantly reanalyzed. To support the dynamic aspects of the existing graphs, incremental graph processing frameworks have been proposed [4, 11, 19]. These systems provide decision makers with access to up-to-date information in real-time.

Incremental graph processing systems require a processing infrastructure to be permanently deployed and operating. Unfortunately, the cost of operating an incremental graph processing infrastructure over long periods of time can easily become prohibitively expensive. For instance, considering the experiment reported by facebook [5] to process their trillion edge social graph. On Amazon EC2, using the same 200 reserved machines of type C4.4xlarge, at current prices, incurs a yearly cost of 1 064 600 USD [1].

Luckily, these costs can be significantly reduced if the incremental graph processing system exploits the diversity of resources currently offered by public cloud providers, such as Amazon and Google Compute Engine. First, these platforms offer a wide variety of machines with prices proportional to the provided level of resources. This can potentially enable deployments that better match the resources with the computational needs. Second, providers often make available transient resource markets that offer clients the possibility to rent, under dynamic price and availability conditions, spare resources with big discounts over the reserved counterparts, with the proviso that they can be withdrawn at any moment[2]. These resources offer a good opportunity to dramatically reduce deployment costs for systems that can tolerate unpredictable availability conditions. Unfortunately, existing incremental graph processing systems often consider homogeneous deployment solutions, leading to over-provisioned services with resources that are, most of the time, left idle [13, 23]. Furthermore, their design is not meant to tolerate unpredictable resource evictions.

In this work we present Hourglass, a deployment engine that works as a middleware between the deployemnt environment and the incremental graph processing framework. Hourglass leverages the existing heterogeneity and transient resources to significantly reduce the long term deployment costs. To achieve this goal, Hourglass offers a number of interesting features: (i) it exploits the characteristics of the workload, such as imbalances in vertex communication, to do a resource-aware distribution of graph partitions among the existing resources. This allows Hourglass to resort to a combination of heterogeneous machines with different resources, a strategy that allows to adapt to price fluctuations that result for

[1] https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/
[2] https://aws.amazon.com/ec2/spot/pricing/

peaks of demand for certain instances' types. (ii) Hourglass constantly analyses market conditions and uses knowledge about the computation being performed to find the right moment to transition among deployments and to dynamically adapt to changes in market prices. Combined, these features allow the incremental graph processing system to operate on a fraction of the cost of a system deployed exclusively of reserved machines, with minimal impact on the system performance, despite the evictions. The obtained results show that the Hourglass system is able to achieve cost reductions of 50% with residual impact on performance and up to a 80% cost reductions with a performance degradation less than 10% over the traditional reserved resources deployments.

The main contributions of this work are the following:

- It presents the first architecture of a deployment engine for incremental graph processing systems that seeks to reduce deployment costs by using transient resources.
- This work also describes how the proposed system is able to use heterogeneous resources leveraging specific features of graph processing task, proposing a hot/cold separation strategy.
- It formally describes the optimization problem associated to the target system deployment, considering multiple problem level constraints, and proposes a way of solving it in reasonable time.
- A sound evaluation that shows how the transient resource usage provides better results when multiple instance types are considered over the traditional homogeneous approach. The presented evaluation also studies the impact that several system level configurations have on the achieved cost savings and system's performance.

The rest of the report is organized as follows. Section 2 identifies the challenges in transient resource usage, applied to our context. Section 3 presents the Hourglass system, the solution to the presented problem. Section 4 evaluates the solution. Section 5 presents the related work and Section 6 concludes the work.

2 CHALLENGES

The usage of transient resources brings some challenges to the system design. The dynamic price and availability conditions of these resources raise problems that did not exist before. A heterogeneous resource approach, that considers multiple types of machines, further raises questions about how to leverage such diversity. We decided to tackle the Amazon Web Services (AWS) environment, as we believe it to be the most challenging for transient resources usage. The current solution could then be easily translated to other cloud providers with simpler transient resource usage mechanisms. The main challenges identified are the following:

(1) How to handle the dynamic market prices for transient resources. At some moment a machine type can have a discount of 90% over the reserved price and some moments later have a price that is 10× higher than that baseline price.
(2) How to take advantage of the heterogeneous environment when partitioning the initial dataset. Traditional solutions consider only one type of machine and try to split the dataset into a set of partitions that have roughly the same computational effort. In our target setting, that considers machines with different computing powers, there is the possibility of creating partitions that are not balanced in terms of computational effort.
(3) How to minimize the impact on performance due to transient resource usage. Although we expect the usage of transient resources to significantly reduce the deployment costs, the increased probability of failure will impact the system performance.

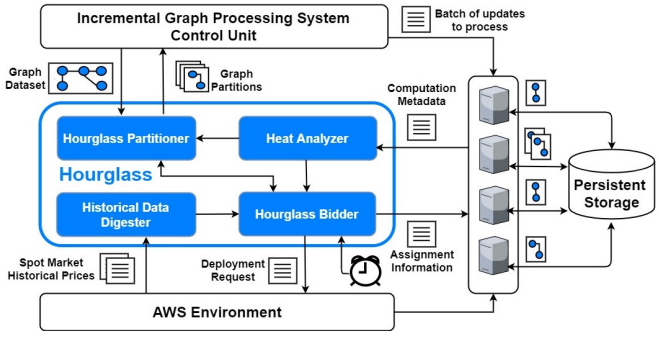

Figure 1: Hourglass **system's overview**

In the following sections we describe the Hourglass system. We detail the main system characteristics that specifically tackle these challenges.

3 HOURGLASS

Hourglass is a deployment engine for incremental graph processing systems. It works as a glue between existing systems and the target deployment environment. As a deployment engine, its goal is to ensure that the underlying incremental graph processing system is deployed in a set of machines that are able to withstand the computational task and, at the same time, find the cheapest deployment configuration that is able to do it. Hourglass leverages heterogeneous transient resources usage in order to significantly reduce the associated deployment costs. The system assumes that these resources are provided by cloud providers on dynamic markets where instances' prices and eviction probabilities are constantly changing. These conditions are frequently analyzed by Hourglass in order to allow it to adapt the current deployment configuration to the new market conditions.

Hourglass has four core components – illustrated in Figure 1 – namely:

- **Heat Analyzer** is responsible to process computational metadata, provided by the underlying graph processing system, in order to calculate the heat of each vertex in the computational task. This data is then provided to *Hourglass Partitioner* to assist in the graph partitioning task and to *Hourglass Bidder* during the deployment process.
- **Hourglass Partitioner** splits the target dataset into a set of partitions, created with certain characteristics meant to improve user-defined goals such as: improve performance; maximize the obtained cost savings; or find a good compromise between the two.

- **Historical Data Digester** reads historical spot market prices from AWS. It uses the historical information to calculate spot instances failure probabilities and estimate costs. This information is then used by *Hourglass Bidder* to choose the deployment configuration.
- **Hourglass Bidder** is the system's component responsible for, at user defined time intervals, find the cheapest assignment for the target set of partitions. Before the beginning of each time slot, *Hourglass Bidder* casts an optimization problem to find the cheapest possible deployment for the current partition's characteristics and spot market prices. The output of this process is a list of machines that need to be requested to the deployment infrastructure and the bid values for each one of them. The result also has the assignment between the existing partitions and the machines available for the next time slot.

3.1 Hourglass Partitioner

Hourglass bases the partitioning problem around the heat distribution. We define heat as the minimum amount of resources that a cluster must have in order to process a given dataset. Once the dataset is divided into partitions, the heat of each partition gives us the minimum amount of resources that a machine must have in order to withstand the computational task associated to that partition. Depending on incremental system's implementation, or on the characteristics of the incremental algorithm being executed, the problem may be memory bound, CPU bound or network bound. The heat should then be measured in a unit of these metrics or in a combination. For simplicity, in the rest of the report we assume the problem to be memory bound and measure the heat in gigabytes.

Once the total heat of the target graph dataset is known, one should decide how the graph will be partitioned. The system should take into consideration the partitioning impact, not only in the system's performance, but also in the possible cost savings achieved.

Improving Execution Time: If the system prioritizes execution time, the created partitions' heat should be as high as the capacity of the largest machine. In detail, as the number of partitions in which we divide the target dataset increases, assuming that the total amount of resources in the system is kept the same, one should also expect the execution time to increase as there is the extra effort of transmitting messages between partitions that are assigned to different machines. Based on this, when the execution time should be prioritized, Hourglass chooses to create partitions as large as the maximum allowed by the available machines. Strategies that seek to maximize local communication between vertices [6, 9, 17] can then be applied to create the partitions with the intended heat value.

Improving Cost Reductions: When leveraging transient resources, the more types of instances we consider the more likely we are to find an instance type that has a significant discount over reserved resources. For example, in the AWS spot market, if one type of spot-instance is under a high demand, its price will rise, possibly even above the price of the on-demand version for that type of instance. When this happens, a good alternative is to assign the partitions to other types of instances that, for that moment, offer a better deal between provided resources and cost.

If we create large partitions, aiming at improving the execution time, we are indirectly hindering the deployment cost as we are reducing the types of instances to which the partitions may be assigned. This means that when the spot price for the instance that we are using rises, we have no alternative other than change to on-demand resources if the spot-price rises beyond that price. If the only goal of the system is to reduce the deployment cost, Hourglass decides to create partitions that can be assigned to all types of instances.

By creating small partitions, that can be assigned to any type of instance, we are not forcing the system to use the smallest possible machines always (worst performance). Under the right market price conditions, if the cost of using large machines is the same of using small machines to achieve the same level of resources, Hourglass prioritizes large resources and assigns multiple partitions to the same machine, achieving the same performance benefits of creating large partitions.

Hot/Cold separation, a good compromise between cost and execution time: As explained before, execution time and cost reductions are objectives maximized under different, and contrary, conditions. The system's performance degradation is directly related to the percentage of the total communication that is not performed locally. Larger partitions reduce the amount of communication performed remotely but may reduce the achieved cost savings by limiting the types of machines to which partitions may be assigned. A good compromise between cost and execution time would be to choose an intermediate term between the two objective functions and create partitions with an intermediate heat value.

Hourglass explores a different approach to achieve a better compromise between cost and execution time. Previous work [4, 11, 19] has focused on creating partitions with equal heat values in order to balance the work performed by each machine in the system, assuming all of them to have the same amount of resources. In order to take advantage of the available resources heterogeneity, we advocate the creation of partitions with different heat values, therefore with different computational needs, in order to improve on the execution time and cost that would be obtained by creating partitions with an intermediate, evenly distributed, heat value.

As graphs present irregular structures, one should expect the communication to follow such irregular patterns. This leads to the existance of distinct heat values across different vertices, observed by [11]. Based on this, we classify the vertices in the system as *hot vertices*, that have a huge roll in the system's performance and are responsible for a great part of the communication, and *cold vertices*, that perform almost no communication and have low impact in the overall execution time. Hourglass focuses on creating partitions that either have hot vertices or cold vertices, therefore promoting a *hot/cold separation* of vertices. On one hand, by creating partitions of hot vertices, the system is placing in the same partitions the vertices that communicate the most and the number of remote messages between machines is expected to decrease significantly, improving execution time. These partitions will have high heat values, reducing the possible machines to which they can be assigned. On the other hand, the remaining cold vertices can be partitioned

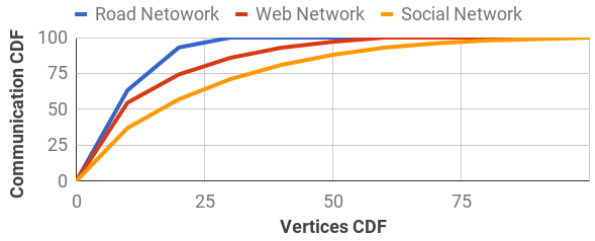

Figure 2: Graph processing communication analysis.

into a higher number of partitions, with low heats, without significantly affecting the system's performance as these vertices are expected to have low impact in the system's performance.

In short, the hot/cold separation approach tries to create a better compromise between cost and execution time by separating hot vertices from cold vertices and creating partitions of different heats (sizes). Partitions with hot vertices will have a higher heat, increasing their deployment cost, but improving the system's performance. Cold vertices can be partitioned into low heat partitions (more partitions) without significantly hindering the execution time and allowing a cheap deployment. In addition to the hot/cold separation, one can still use partitioning techniques that favor local communication. For example, these techniques can be used to choose hot vertices' placement if more than one hot partition is necessary. This will further help to reduce the communication between machines. Besides that, these techniques are often associated with large pre-processing times[16], by limiting the partitioning to hot vertices, as we expect cold vertices to communicate a lot less, we are also reducing the pre-processing time and increasing partition quality by reducing the dataset size [20, 21].

However, in order to this hot cold separation to be useful, the number of hot vertices should be smaller than the number of cold vertices. A lower percentage of hot vertices will reduce the number of partitions with high heat, necessary to improve execution time, and increase the number of partitions with low heat, that reduce deployment costs. As explained before, communication is the ruling factor for vertices' heat. Figure 2 presents a communication analysis performed over three datasets of different sizes and types, namely a social network [26], a webpages network [12] and a road network [12]. The figures show the relation, on a vertex-centric approach, between the percentage of the number vertices (x-axis) and the percentage of the total communication performed during the computation (y-axis). The presented results are for a the Single-Source Shortest Paths (SSSP) application with a 25% percentage of vertices receiving updates. A more detailed evaluation was performed with more application types and different percentage of vertices receiving updates. These are not shown here due to space constraints. However, the presented results are representative of the results obtained for the other experiments.

Based on this results, we can see that a small set of vertices is responsible for most part of the communication. This evidence suggests that the conditions for a hot/cold separation approach to work are met.

3.2 Heat Analyzer

The *Heat Analyzer* component receives metadata information from the computational task being performed to calculate the heat values of different vertices. The heat of a vertex is given as a function of the amount of updates it receives and the communication it performs during the incremental computation task.

The current implementation of this component uses a simple, yet effective, approach that defines the heat of a vertex simply by counting the number of updates it receives and counting the number of messages it sends and receives during the incremental computational task. This information is crucial to the *Hourglass Partitioner* component, as it takes a major role in the partitioning task.

This component also keeps track of the current dataset heat, given by the sum of its partitions' heat and empirically measured by analyzing each machine load information. This information is then used by the *Hourglass Bidder* to create valid assignments between machines and partitions, preventing the outburst of machine capacity.

3.3 Historical Data Digester

On Amazon, spot-instances work on a auction based system. Evictions occur when the current market price for a given instance type is above the bid placed for that same instance. Previous work [8, 10, 24] has consistently used techniques based on historical data of spot market prices to predict evictions probabilities or the expected time before an eviction occurs under a certain bid value. This information is crucial to control, to some extent, the evictions occurring in the system. A way of predicting the probability of failure for a given instance under a certain bid is essential to ensure the system's availability and the computational task progress.

This component provides two services that are used by *Hourglass Bidder* during the deployment selection phase. In this section we present the two services and detail how they are implemented. On Section 3.4 we explain how they are used.

The historical data is organized by instance type, it is defined by a group of datapoints representing spot market price changes. These datapoints contain the spot market price and the timestamp when such market value become effective.

Bid Oracle Service: This service, based on historical spot market prices, returns the bid value that should be issued for a given instance in order to met the probability of failure for the intended period duration. The service receives as input the target instance type, its current market price, the target probability of failure and the duration of the period. The process of finding the bid value goes as follows:

(1) The service scans the data finding all datapoints where the target resource type had the current market value. If the collected set of datapoints is not statistically significant, the service does add all datapoints where the market value is equal to the current market value ± 0.01 cents (the price unit on Amazon). This procedure is repeated until a significant number of datapoints are collected.
(2) The service computes all past time periods (for the requested duration) that started with the current market price. This is

done by creating a set, for each of the collected data points, containing that datapoint followed by all other datapoints in the historical timeline that fit into that period.
(3) For each of the sets obtained in the previous step, the service finds the highest price that the resource type reached during that period of time. This price represents the bid price that would have precluded evictions during that specific period. The service then creates an ordered list l, in descending order, with all the obtained values.
(4) The service returns the $\lfloor p * n \rfloor$-th element in l, where $p \in [0, 1]$ is the eviction probability and n is the list l length. This value is the bid price that with a probability $1-p$ would have caused no evictions on the historical periods considered.

Cost Estimation Service: This service is used to estimate the cost of an instance for a given period of time. On Amazon, the cost paid for each instance is defined in the beginning of each billing hour as the current market price of the target instance. This service receives as input the target instance type, its current market price and the duration of intended time period. The result is the estimated cost of that instance for that period. The cost is estimated as follows. As the Bid Oracle service, the Cost Estimator, based on historical data, computes all past time periods that started with the current market price (steps 1 and 2). Then, for each of those periods computes how much renting that type of resource costed. Finally, it returns the average among those costs.

3.4 Hourglass Bidder

Hourglass is responsible for, at user defined time periods, find the assignment for the existing partitions and machines that is expected to minimize the deployment cost. This is selected based on the current market values and always respecting the associated capacity constraints. To solve this problem, Hourglass uses a subcomponent called *Hourglass Bidder* that is responsible for, in the end of each time period, cast an optimization problem to find such deployment configuration.

When creating the assignment between partitions and machines, some aspects, important to ensure the proper functioning of the system and meet user defined goals, need to be considered. The aspects are the following:

- **Eviction Rate:** In the AWS auction system, bids have a direct impact in the probability of failure for a given machine. In order to meet the expected eviction rate, Hourglass allows the user to specify the probability of failure that it is willing to accept. The target bids are then obtained using the bid oracle service (Section 3.3).
- **Partition's Heat:** Partitions have associated heat values that represent the minimum amount of resources that a machine must have in order to withstand the associated computational task to that partition. When assigning partitions, the system must ensure that every machine is assigned a set of partitions that do not burst its capacity.
- **Expected Instance Cost:** As explained in Section 3.3, the price per instance is different from the price paid. The previously described cost estimation service is used to estimate the price of a given instance.

All things considered, the goal of *Hourglass Bidder* is to find an assignment between the existing partitions and the possible instances that may be requested to the deployment infrastructure. This assignment must respect the above mentioned aspects and minimize the expected deployment cost.

3.4.1 *Deployment as an Optimization Problem.*

The deployment problem is formulated as an optimization problem. *Hourglass Bidder* uses a constraint solver to find the best possible assignment in every moment. Considering M as the set of all available machines for time period t and P the set of all existing partitions. The objective of *Hourglass Bidder* in time t is to return the set of boolean variables x_t^{pm}, true if partition p is assigned to machine m on time t and false otherwise. γ_t^m denotes the binary decision variable that becomes 1 if a machine m allocates any partition in time t

More formally, the above described problem is defined as follows:

$$\text{minimize:} \quad \sum_{m \in M} (\gamma_t^m * \hat{\mu}_t^m),\ t \in T, m \in M \tag{1}$$

$$\text{s.t.} \quad x_t^{pm} - \gamma_t^m \leq 0,\ \forall m \in M, \forall p \in P, t \in T \tag{2}$$

$$\sum_{m \in M} x_t^{pm} = 1, \forall p \in P \tag{3}$$

$$\sum_{p \in P} (x_t^{pm} * \sigma_p) \leq \beta_m, \forall m \in M \tag{4}$$

Constraint 1 in the above formulation ensures that if at least one partition is assigned to a machine, the estimated cost of this machine ($\hat{\mu}_t^m$) is considered in the cost computation. Constraint 2 ensures that each partition is assigned to one and only one machine. Constraint 3 ensures that the sizes of partitions (σ_p) assigned to a given machine do not outburst its capacity (β_m). Hourglass uses CPLEX[3] to solve the above formulation. The current implementation allows the system to obtain the solution up to a problem size of 3000 partitions under one minute.

3.5 Fault and Eviction Handling

The Hourglass architecture consists of four components, each representing a single point of failure. The goal of our prototype is to show the potential cost reductions that Hourglass can achieve and it is not implemented to tolerate failures in any of those four components. We consider addressing these concerns orthogonal to the contributions of this paper. Nevertheless, in a real implementation of the system, this components would need to be replicated. Standard machine replication techniques [18] can be employed to make each component robust to failures.

Apart from failures of the Hourglass components, Hourglass has to deal with failures and evictions of the machines that instantiate the graph partitions. The procedure to recover from such failures is similar to the reconfiguration procedure. Thus, when a failure is detected, *Hourglass Bidder* first finds the optimal deployment for the partitions that were instantiated in the lost machines. Then, it triggers the starting phase by booting the new machines and coordinating them, with the incremental graph processing system, how these machines recover the application state. There are

[3] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

two situations to consider. First, if the failure/eviction happens in between computational phases, the recovered partitions simply read the latest checkpoint from the persistent storage. Second, if the eviction/failure occurs during the computational phase. In this case, Hourglass notifies to the processing system that the computational phase is aborted, which notifies the remaining partitions to rollback to the previous consistent checkpoint. Finally, the computational phase is restarted once all partitions are instantiated. We assume that the underlying system has its own fault tolerant mechanisms to ensure that not update is lost until its effects are reflected in a checkpoint. This is a common assumption as this system normally batch updates in a centralized component and only scatter them at the beginning of each computational phase.

One of the consequences of failures/evictions is that Hourglass has to handle resources being acquired at different points in time. Thus, the end of a time-slot may not match the end of the billing hour for all resources. Hourglass can handle multiple of these timelines temporarily. Nevertheless, it tries to merge these into a single timeline: when the time-slot is about to end, timeliness for resources whose end of the current billing hour is closer than a given threshold (less than 30 minutes for instance) are merged with the main timeline (the one in which time-slots with the billing hours are synchronized). Of course, this has an impact on deployment cost. Nevertheless, the alternative would be to maintain multiple timelines indefinitely which adds computational overhead. Also, this alternative would negatively impact cost, as the system would be reconfiguring the system by parts and not in one-shot, reducing the quality of the deployment configurations.

4 EVALUATION

Hourglass main goal, as a deployment engine for incremental graph processing systems, is to reduce deployment costs without significantly affect the system performance and availability. As discussed in Section 3, there are several factors, either related to the graph processing task or to the deployment environment interaction, that can impact the outcome results. In this section, we intend to study the impact of several factors, such as heat distribution technique, bidding strategy and others, on the possible savings achieved by Hourglass. Our goal is to understand the best configuration for the system under different market conditions, user constraints and underlying graph processing system characteristics.

4.1 Experimental Setup

The experiments carried out consider five different types of instances from the C4 compute optimized instance family provided by Amazon[4]. All instances run Ubuntu Server 14.04 LTS.

Due to the lack of open-source incremental graph processing systems, in order to evaluate the computational impact of some decisions we created a prototype that resembles a typical vertex-centric implementation, identical to the Kineograph[4] system.

Experiments use two different applications. The SSSP application, that calculates the distance from all vertices to a single vertex, and the TunkRank[22] application, similar to PageRank [2] applied on social network context.

[4]https://aws.amazon.com/ec2/instance-types/

Table 1: Graph dataset description.

Dataset	#Vertices	#Edges	Type
Orkut	3,072,441	117,185,083	Social Network
Synthetic1	4,000,000	400,000,000	Synthetic

Table 1 describes the datasets used in the experimental evaluation. The syntetic graph was obtained using the R-MAT [3] generator with the same parameters used by benchmark graph500[5] to generate scale-free graphs.

4.2 Cost Analysis Simulator

All costs presented on this section were obtained by running different experiments on a simulator created to emulate the characteristics of the target deployment environment rather than experiments on the AWS environment for several reasons. First, prices for spot-instances often change on a daily basis, therefore, in order to obtain meaningful results we should consider time frames which are not smaller than few days, otherwise, the results would be of little interest. Due to the large number of experiments that were carried out, and the limited existing budget, running all experiments with several days per experiment would have been impossible. Also, when comparing different strategies, it is important to consider the same time period, where the price variations are exactly the same, to provide a fair point of comparison.

All costs presented were obtained by simulating the target experiment over a AWS price trace that goes from July 13 to August 9, 2017 (about 650 hours) for the *us-east* region. Unless otherwise stated, the simulator emulates the AWS conditions, the system uses the historical bidding strategy with a 25% probability of failure and one hour time slots, meaning that in the end of every billing hour a new assignment is selected.

4.3 Heterogeneous Transient Resource Usage

Using transient resources, Hourglass should be able to significantly reduce the deployment costs compared to the usage of reserved resources, as the prices for spot-instances can some times reach a 90% discount. In this experiment we intend to show how the usage of multiple instances types further improves the cost savings achieved over only considering a single type of spot-instance.

Figure 3 shows the obtained costs by running the system with a set of 16 partitions with a heat value of 3.75Gb each (60Gb total) for different deployment strategies. *Only OD* strategy considers only on-demand instances to hold graph partitions and is used as the baseline comparison. The *Hourglass* strategy considers the usage of all types of spot-instances and the other strategies consider the usage of only one type, from the above described types, of spot-instances. The figure shows a stacked area graphic that for each strategy shows the cost spent in each type of instance. The simulation, for strategies that consider spot-instances, decides to use on-demand resources if the price for the available spot-instances is above the price for on-demand.

As expected, all strategies that consider spot-instances obtain cost savings from 40% to 75% over the cost of only using on-demand

[5]http://graph500.org/?page_id=12

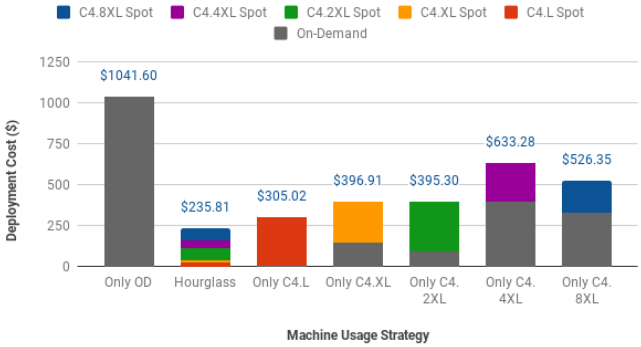

Figure 3: Cost reductions over different machine usage strategies.

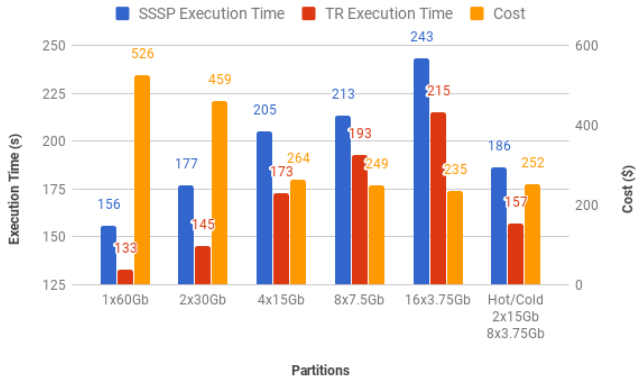

Figure 4: Execution time and cost analysis for the synthetic dataset.

resources. The best cost reductions are achieved by the Hourglass strategy that considers all types of spot-instances. As spot-instances have independent market values, by considering all types of spot-instances, Hourglass is able to efficiently choose replacement instances for machine types whose price went up.

4.4 Batch Execution Time and Cost Analysis

As explained before, for the same dataset, smaller partitions are able to achieve the best cost savings over large partitions. However, as discussed in Section 3.1, as the size of partitions decreases and the potential number of machines performing the computation increases, the execution time is expected to increase, assuming that the amount of resources in the system is kept the same.

We analyzed the deployment cost, batch execution time and messages exchanged during computation to study if the different heat distribution strategies produce the expected results. Figure 4 compares the execution time and deployment cost associated to the synthetic dataset (60GB heat). The figure shows the execution time to process a batch of 10 million updates and perform the incremental computation of both applications, for different numbers

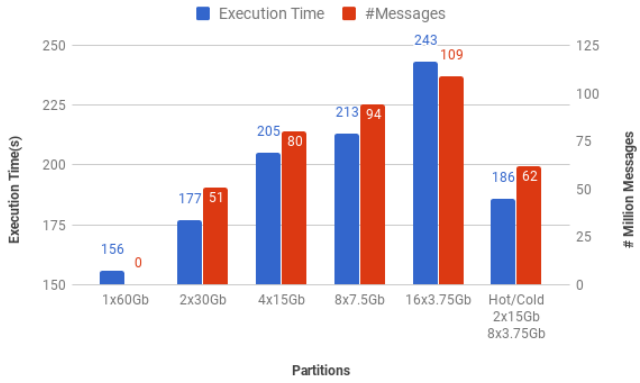

Figure 5: Execution time and messages exchange analysis for the synthetic dataset.

of partitions, namely one 60Gb heat partition, two 30Gb heat partitions, four 15Gb partitions, eight 7.5Gb partitions and sixteen 3.75Gb partitions. The batch execution time was measured using the worst deployment for the computational time, that is, using the smallest possible machine for every partition. The figure also shows the associated deployment cost for each configuration over the duration of the cost simulation (about 650 hours). Results show that, in fact, larger partitions improve batch execution time and increase deployment costs, having small partitions with the opposite trade-off. The hot/cold separation also proves to be a better compromise between cost and execution time than just picking an intermediary partition size. Figure 5 shows the relation between batch execution time and the number of messages exchanged remotely during computation for the SSSP application. Analyzing the results, assuming that all deployment settings have in total the same amount of resources, just divided across into different numbers of machines. It is pretty clear the relation between the execution time and the number of exchanged messages. The results for the hot/cold separation show how the hot vertex placement on the same partitions help to reduce the number of exchanged messages.

4.5 Update Staleness Analysis

In the previous section we evaluated the impact that different heat distribution strategies have in the batch execution time for the worst deployment case. Although these results show that, in fact, the heat distribution impacts the batch execution time the way we predicted, there are other aspects that need to be considered. First, the worst case scenario is not always the deployment configuration being used. Second, the performance of the system is also impacted by other factors, as periodic reconfigurations, spot evictions and state checkpointing. Considering this, we find the batch execution time to be a good performance metric but lacks the temporal continuity necessary to cover the above mentioned aspects. To solve this problem we decided to use another performance metric, the average update staleness over a time period. This metric measures the time passed since an update was received by the system until its effects become reflected in the persistent state of the graph (reliably stored),

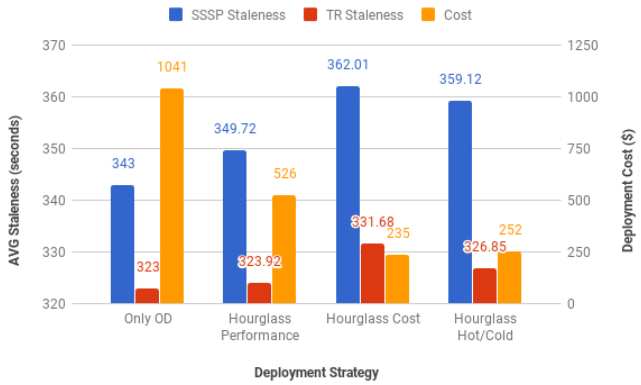

Figure 6: Update staleness and cost analysis for the synthetic dataset.

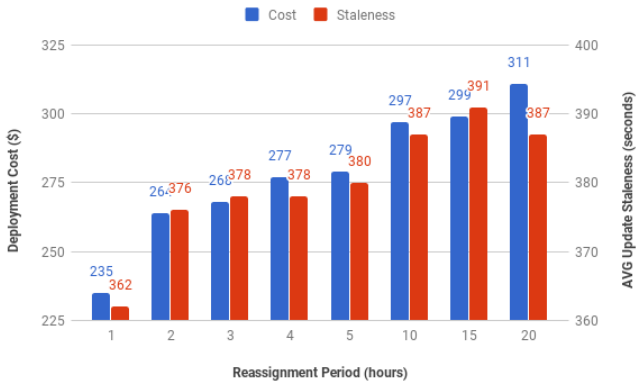

Figure 7: Reassignment impact on performance and deployment cost analysis.

averaged for the amount of updates received during a specific time period. This metric reflects all the above mentioned aspects and is therefore a more complete performance metric.

To measure this metric we extended the previously described cost simulator. The simulator allows us to know, at each moment in time, the machines being used and the assigned partitions to each one of these machines. We can also know from this simulation the failures that occurred and when they occurred. This information, together with data obtained from real executions in the Amazon environment, allows us to do a very good estimation of the update staleness over the same period duration of cost estimation (around 650 hours). To do this, we executed the batch processing phase for every dataset and incremental application in all possible deployment configurations that were observed during the cost simulation process, and for different batch sizes that can accumulate from consequent failures or reassignment phases. Apart from this, we also measured machine's startup times and partition's read and write checkpoint times for all possible partition sizes that were tested.

Figure 6 shows the obtained results for the synthetic dataset. Results for the orkut dataset lead to similar conclusions and is not here presented due to space constraints. Analyzing the results, the different partitioning objectives met the initial expectations and follow the same trend as the batch execution times. More precisely, when executing for performance, Hourglass is able to achieve performance results that are not worse than 2%, reaching discounts of 50% over the same baseline cost. When executing to reduce cost, Hourglass is able to achieve cost reductions up to 78% over the baseline cost with no more than 10% performance degradation. The hot/cold separation again shows a good compromise between between cost and execution time.

4.6 Reassignment Frequency Analysis

So far, all experiments considered a reassignment period of one hour. This means that in the end of every billing hour, *Hourglass Bidder* initiates the process to select a new assignment for the next time period. In this section we analyze the impact of selecting different reassignment periods. The results presented in this section are for a

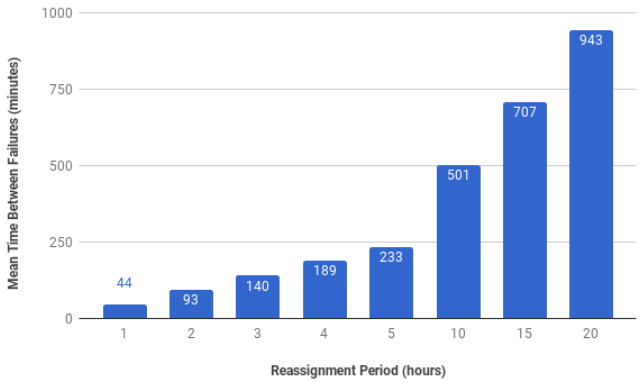

Figure 8: Reassignment frequency impact on the mean time between failures.

60Gb heat dataset deployment, using the partitioning method that improves cost savings, running the the SSSP application.

The reassignment period impacts several aspects. It impacts the deployment cost, as larger reassignment periods reduce the system ability to adapt to price market changes. Figure 7 shows this impact in the deployment cost and also in the system's performance. The performance degradation is mainly due to one factor. For larger reassignment periods, smaller machines have the cheapest estimated costs as they have more stable prices. Therefore, by increasing the reassignment period, we are indirectly prioritizing smaller instances that reduce the system performance.

If increasing the reassignment period degrades both performance and cost one may ask what is the benefit of increasing this period. Selecting the smallest reassignment period seems to be the best solution for all cases. However, the reassignment period duration imposes a possible hard barrier in the system availability, that is, it is necessary to interrupt the system to possibly start new machines. This may have significant impact in performance if this period is

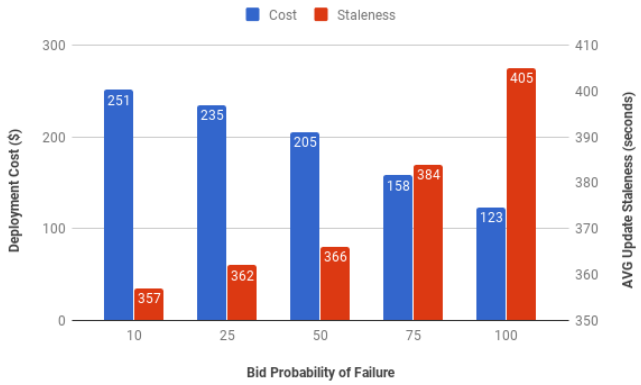

Figure 9: Probability of failure impact on deployment cost and performance.

smaller than the time the system takes to process at least one update batch and checkpoint the current state. This leads to situations where the system is constantly interrupting the ongoing computation and preventing computational progress. So, another important aspect to the system is the Mean Time Between Failures (MTBF), here translated as the mean time between reassignments. This is the time the system has on average without any interference, either due to a spot instance's failure or to the end of the current time slot. Figure 8 shows the obtained MTBF under different reassignment periods. These periods were obtained by running the simulation and include both pauses due to planned reconfigurations and spot failures. We can see that, as the time slot duration increases the MTBF also increases. The reassignment period should be selected so that the MTBF is greater than the time the system usually takes to process at least one update batch and checkpoint the graph state.

4.7 Historical Bid Strategy Under Different Probabilities of Failure

As explained on Section 3.3, the bid value used to request spot instances will dictate if an instance gets revoked or gets to the end of the target time period. Hourglass allows the specification of the target probability of failure for the spot-instances. An approach based on past historical data is then used to find the bid value that matches the target probability of failure.

The probability of failure will impact the eviction rate of spot-instances. In this Section we analyze, for a dataset of 60Gb size running the SSSP application, with partitions that reduce the deployment cost, the impact of different probabilities of failure in performance and cost for the historical bidding strategy. The observed results are representative of the impact that this failure probability has over the mentioned aspects for different datasets, applications, reassignment periods and partitioning methods, not here presented again due to space constraints.

Figure 9 shows the impact that different probabilities of failure (10%, 25%, 50%, 75% and 100%) have in the deployment cost and system's performance. The obtained results are explained by a simple factor, eviction rate. Lower probabilities of failure have less failures that translate into higher MTBF periods and better performance. Higher probabilities of failure induce higher eviction rates, deteriorating the performance due to the increased number of faults. However, these faults reduce the deployment cost due to the free computing that the system gets in the last hour of each machine before the eviction. For systems that can tolerate the performance degradation, a higher probability of failure translates into cheaper deployments. For example, for the analyzed dataset, a probability of failure of 100% is able to achieve a deployment cost that is 88% cheaper than the on-demand baseline and with a performance degradation of 25% over the same baseline.

5 RELATED WORK

Graph Processing. Static graph processing systems are designed to run algorithms over a static graph data model. In vertex-centric approaches, such as Google's Pregel [15], computation proceeds iteratively following the Bulk Synchronous Parallel (BSP) execution model [1] using synchronization barriers between iterations. To mitigate the performance degradation caused by the global synchronization barriers of the BSP model, approaches such as GiraphUC [7] use an asynchronous execution model. In this model, vertices process computational messages as soon as they are received, without waiting for synchronous barriers. This model outperforms synchronous approaches for CPU intensive algorithms but degrades in I/O intensive applications due to the lack of batching [25].

Nonetheless, the major shortcoming of static graph processing systems is the assumption that the underlying graph structure does not change. In fact, when the graph structure changes, the algorithms need to be run from scratch which is very inefficient [19]. To overcome the limitations of static graph processing systems, proposals such as Kineograph [4], GraphIn [19] and iGraph [11], allow for incremental graph processing where the underlying graph structure can change and the computation is adjusted accordingly. Nonetheless, despite flexible, incremental graph processing approaches are oblivious to the underlying environment heterogeneity.

Transient Resource Usage. Major cloud providers such as Amazon, Google and Microsoft offer transient resources in their platforms. Transient resources are usually offered at a significant discount over the non-transient counterparts with the downside that these resources can be terminated at any time without notice. It follows naturally that a careful management of transient resources can be an effective way to reduce deployment costs, provided that the dynamic resource availability can be tolerated. This observation, on which the contributions of Hourglass rest, has been exploited before in systems such as Pado [27] and Proteus [8].

Pado [27] is a general data processing engine that leverages idle resources of over-provisioned nodes, to run batch data analytic jobs on a mix of reserved and transient resources. The system receives as input a Directed Acyclic Graph (DAG) of computations representing the job that is going to be processed. The goal is to find the computations that would cause the larger recomputation costs if evicted, and assign these to the non-transient resources and all the others to transient resources. This is effective for offline analytical jobs, where the input is well known and finite, but it does not fit an incremental graph processing model. This is because

the graph computation is highly dependent on the type of updates received, making the computation cost unpredictable.

Proteus [8] exploits transient resources to perform statistical machine learning. It iteratively processes training data to converge on model parameter values that, once trained, can predict outcomes based on the computed model. Proteus' end goal is to reduce the expected cost to process a finite workload which is done by creating an assignment to transient and non-transient resources that is expected to reduce the overall cost per workload. This means that the system may acquire more resources, more expensive for that particular moment, if it expects that it will reduce the time to process the remaining workload and reduce the overall cost of processing the workload. On an incremental graph processing system, the deployment is continuous and the workload is considered infinite, making Proteus's optimization goal unfit for our model.

6 CONCLUSION

In this work we analyzed incremental graph processing systems, that allow the underlying graph structure to change and adapt the computation values obtained from the target algorithm being executed to the graph mutations. The graphs that model domains of interest present a fast changing structure that needs to be constantly analyzed, motivating long lived deployment that become very expensive. We identified that most incremental graph processing systems are oblivious of the underlying deployment environment, missing opportunities to significantly reduce operational costs.

In order to fill this gap between the existing systems and deployment environment we proposed Hourglass, a deployment engine for incremental graph processing systems. The system leverages heterogeneous transient resource usage to significantly reduce the associated operational costs. The system is constantly analyzing market price variations for the target transient resources and, on user defined reassignment periods, selects the cheapest deployment configuration for the next time period.

The obtained results show that Hourglass is able to achieve cost savings of 50% with residual impact on performance and up to a 88% cost reductions with a performance degradation less than 25% over the traditional reserved resources deployments.

Acknowledgments. This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects with references PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013. Also, we would like to acknowledge the AWS Program for Research and Education.

REFERENCES

[1] 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[2] S Brin and L Page. 1998. The anatomy of a large scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1/7 (1998), 107–17.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining.. In *SDM*, Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn (Eds.). SIAM, 442–446.

[4] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, Bern, Switzerland, 85–98.

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815.

[6] Je Gonzalez, Y Low, and H Gu. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), 17–30.

[7] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.* 8, 9 (May 2015), 950–961.

[8] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. 2017. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Belgrade, Serbia, 589–604.

[9] Jiewen Huang and Daniel J. Abadi. 2016. Leopard: Lightweight Edge-oriented Partitioning and Replication for Dynamic Graphs. *Proc. VLDB Endow.* 9, 7 (March 2016), 540–551.

[10] Bahman Javadi, Ruppa K. Thulasiramy, and Rajkumar Buyya. 2011. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC '11)*. IEEE Computer Society, Washington, DC, USA, 219–228.

[11] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. 2016. iGraph: An Incremental Data Processing System for Dynamic Graph. *Front. Comput. Sci.* 10, 3 (June 2016), 462–476.

[12] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR* abs/0810.1355 (2008).

[13] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. Portland, Oregon, 450–462.

[14] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. *CoRR* abs/1006.4990 (2010).

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, Indianapolis, Indiana, USA, 135–146.

[16] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 631–643.

[17] Christian Mayer, Muhammad Adnan Tariq, Chen Li, and Kurt Rothermel. 2016. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS)*. 118–128.

[18] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319.

[19] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. *GraphIn: An Online High Performance Incremental Graph Processing Framework*. Springer International Publishing, Cham, 319–333.

[20] George M. Slota, Sivasankaran Rajamanickam, Karen D. Devine, and Kamesh Madduri. 2016. Partitioning Trillion-edge Graphs in Minutes. *CoRR* abs/1610.07220 (2016).

[21] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, Chicago, Illinois, 16:1–16:10.

[22] Daniel Tunkelang. 2009. A twitter analog to pagerank. Retrieved from http://thenoisychannel.com/2009/01/13/a-twitter-analog-topagerank. (2009).

[23] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Bordeaux, France, Article 18, 17 pages.

[24] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. 2017. Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, Belgrade, Serbia, 620–634.

[25] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. *SIGPLAN Not.* 50, 8 (Jan. 2015), 194–204.

[26] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012).

[27] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Belgrade, Serbia, 575–588.