TÉCNICO LISBOA

Leveraging Transient Resources for Incremental Graph Processing on Heterogeneous Infrastructures

Pedro Miguel Marcos Joaquim

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Members of the Committee: Prof. Sérgio Marco Duarte

October 2017

Acknowledgments

I would like to acknowledge my dissertation supervisor Prof. Luís Rodrigues. Without his guidance, share of knowledge and persistent help this dissertation would not have been possible.

During my work I have benefited from the valuable contributions of several members of the Distributed Systems Group of INESC-ID Lisboa. In particular, I am grateful to Manuel Bravo, Miguel Matos, and Rodrigo Rodrigues, for the fruitful discussions and comments during the preparation of this dissertation.

I want to thank all my teachers since the first years of school that contributed to my education and guided me in the search for knowledge.

I would like to thank my parents for their friendship, encouragement, caring and for their understanding and support throughout all these years. Thank you for the education you gave me. Thank you for always being there for me during the good and bad times in my life.

To my girlfriend, thank you for the comprehension during these hard work periods. Thank you for making my life much better, giving me strength when need it the most. Thank you for being the way you are and for supporting me in my decisions.

Last but not least, to all my friends and colleagues that helped me grow as a person and made this journey much funnier. Thank you.

To each and every one of you – Thank you.

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 dissertation 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.

Keywords

Graph Processing, Cloud Computing, Resource Management

Resumo

O processamento incremental de grafos tem hoje aplicações nas mais diversas áreas. Aplicações deste tipo de processamento tem a característica comum de efectuarem um processamento contínuo, necessitando assim de um sistema em operação permanente. Para sistemas em ambientes de infraestrutura na nuvem, este tipo de disponibilidade pode incorrer em custos operacionais muito elevados. No entanto, os fornecedores de infraestrutura na nuvem oferecem recursos efémeros com custos bastante reduzidos face aos recursos reservados, contudo estes podem ser retirados a qualquer momento. Estes recursos, aliados à necessidade de sistemas em permanente operação, apresentam uma grande oportunidade para reduzir significativamente os custos operacionais, não aproveitada por sistemas já existentes.

Neste trabalho apresentamos o Hourglass, um motor de exploração para sistemas de processamento incremental de grafos cujo principal objetivo é reduzir os custos operacionais, tirando partido da heterogeneidade dos recursos efémeros existentes. O sistema tira partido de características da computação a ser realizada, tais como os diferentes padrões de comunicação de cada vértice, para efetuar uma atribuição de partições ciente das suas necessidades computacionais. O Hourglass está constantemente a analisar as condições atuais do mercado alvo para escolher a melhor configuração de máquinas a utilizar. Um conhecimento da computação a ser efetuada permite ao Hourglass escolher os momentos certos para fazer a transição entre máquinas. Isto permite que o sistema opere numa fracção dos custos que seriam necessários caso só máquinas reservadas fossem utilizadas, com um impacto mínimo no desempenho do sistema.

Palavras Chave

Processamento de Grafos, Computação na Nuvem, Gestão de Recursos

Contents

List of Figures

List of Tables

Acronyms

AWS	Amazon Web Services
BFS	Breadth First Search
BSP	Bulk Synchronous Parallel
CPU	Central Processing Unit
CRUD	Create, Remove, Update and Delete
DAG	Directed Acyclic Graph
EBS	Elastic Block Store
GAS	Gather, Apply, Scatter
GCE	Google Compute Engine
I-GAS	Incremental-Gather-Apply-Scatter
IaaS	Infrastructure as a Service
MTBF	Mean Time Between Failures
NUMA	Non-Uniform Memory Access
OLAP	Online Analytical Processing
OLTP	Online Transactional Processing
PSW	Parallel Sliding Windows
SSSP	Single-Source Shortest Paths

1

Introduction

Contents

The recent growth of datasets represented as graphs, including social networks [1], web graphs [2], and biological networks [3], motivated the appearance of systems dedicated to the processing of such graph datasets. These systems leverage the graph structure to ease the development of algorithms that are able to perform the computational analysis of the graph. Graph processing systems also provide an efficient execution model to support computations that, typically, cover the entire graph multiple times using iterative procedures. The information extracted from these computations usually has a significant scientific, social, or business value. For example, recommendation systems [4, 5] leverage on graphs representing relations between users and the products that these users buy (consumer networks) or people and posts that they follow (social networks) in order to identify groups of similar users. Based on this information, these recommendation systems are able to recommend items/posts that the users are more likely to buy/read, increasing revenue. In this dissertation we study frameworks that support the processing of large graphs in cloud computing infrastructures.

1.1 Background

Different architectures to build these systems have been proposed. Distributed graph processing models, such as Google's Pregel [6], propose a vertex-centric approach to specify the algorithms that perform the computation on the graph. According to this approach, the computation is modeled as a *compute* function that needs to be repeatedly applied to each vertex of the graph, in an iterative process known as the BSP execution model. This model is supported by a master/slave architecture, where slaves split the input graph among them in *partitions* and become responsible for performing the computation at each of the assigned partitions. The master node is responsible for coordinating the execution and partitioning the graph in the start of the process. Single machine architectures [7, 8] have also been proposed to address the challenges of the distributed counterparts, such as cluster management and communication overhead. Systems that follow this approach have an identical vertex-centric computational model and either leverage on secondary storage to enable large graph processing or use server class machines with large resources to do so.

Graphs that model most subjects of interest present fast chaning structures that need to constantly analyzed. For example, social networks are in constant evolution as new users join the network or new relations are created. To support the dynamic aspect of the underlying graphs, that suffer modifications as the time goes by, incremental graph processing models have been proposed [9–11]. They aim at keeping the graph structure updated and use incremental algorithms that allow to update the computational values obtained avoiding recompute everything from scratch every time a new graph version is available. These systems typically work in two distinct phases. The first phase consists of batching incoming updates to the underlying graph. The second phase then consists of updating the graph and

executing the incremental algorithm to update the computational values.

Most incremental graph processing systems are deployed on commodity clusters on Infrastructure as a Service (IaaS) platforms managed by public cloud providers, such as Amazon[1] and Google Compute Engine[2]. These platforms offer a wide variety of machines with prices proportional to the provided level of resources. In addition to this, public cloud providers often make available transient resource markets. These markets offer clients the possibility to rent, under dynamic price and availability conditions, spare resources with big discounts[3] over the reserved counterparts with the proviso that they can be withdrawn at any moment. These resources, frequently called transient resources due to their transient and unpredictable life time, offer a good opportunity to significantly reduce deployment costs for systems that can tolerate unpredictable availability conditions for its resources.

Unfortunately, existing incremental graph processing systems are oblivious to the heterogeneity of resources and services in the target environment, ignoring opportunities to reduce the costs by matching the resources with the computational needs. Often these systems consider a homogeneous deployment solution where the type of machine is selected without taking the target dataset and computational task characteristics into consideration, leading to situations where the selected machines are over-provisioned with resources that are paid but not used.

1.2 Hourglass

In order to fill the gap between existing incremental graph processing systems and underlying cloud providers we present Hourglass. This system aims at leveraging the existing heterogeneity in such deployment environments, as well as transient resource usage, to significantly reduce deployment costs. As transient resources markets often have prices based on fluctuations in the demand for certain instances, a strategy that leverages different instance's types (heterogeneous resources) is important to ensure that enough alternatives are available under high demand for certain instances' types. However, the graph processing task has associated some resource level constraints that prevent a clear allocation to any type of instance. 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. Hourglass constantly analyses market conditions to find the best deployment configurations. Knowledge about the computation being performed allows Hourglass to choose the right moment to transition between deployments and allow the underlying system to operate on a fraction of the cost of a system composed exclusively of reserved machines, with minimal impact on the system performance. The obtained results show that the Hourglass system is able to achieve cost reductions up to 80% with

[1] https://aws.amazon.com/
[2] https://cloud.google.com/compute/
[3] https://aws.amazon.com/ec2/spot/pricing/

less than 10% performance degradation over the traditional reserved resources deployments.

1.3 Contributions

The main contributions of this work are the following:

- It presents the first architecture of a deployment engine for incremental graph processing systems, leveraging transient resource usage to reduce operational costs.
- It proposes a new hot and cold separation heat distribution strategy that seeks to take advantage of resource heterogeneity.
- It models resource allocation as an optimization problem. Minimizing the cost function associated to the operational costs.

1.4 Results

The main results of the work are the following:

- A working prototype implementation of the Hourglass framework.
- A detailed evaluation proving how the heterogeneous transient resource usage is able to achieve huge discounts over the traditional reserved resource usage, with minimal impact in performance. The presented evaluation also studies the impact of several system level configurations on the achieved cost savings and performance.

1.5 Research History

This work as been performed in the Distributed Systems Group of INESC-ID Lisboa. During my work I have benefited from the valuable contributions of several members of the group. In particular, I am grateful to Manuel Bravo, Miguel Matos, and Rodrigo Rodrigues, for the fruitful discussions and comments during the preparation of this dissertation.

The work started with some proof of concept experiments where we studied the possibility of using machines with different computational capacities to perform the graph processing task using the hot/cold separation and studied ways of reducing the impact of transient resource usage. Parts of this initial work have been published as a poster in the 9^{o} Simpósio de Informática, Inforum, Aveiro, Outubro de 2017. After this initial work, the Hourglass system evolved to the work that is here presented.

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, the extent evaluation presented in this dissertation was possible due to the grant received from AWS Program for Research and Education.

1.6 Work Organization

The dissertation is organized as follows: Chapter 1 introduces the proposed work. In Chapters 2 and 3 we survey the related work and detail the state of art in graph processing systems and in systems that leverage transient resources, respectively. Chapter 4 presents the proposed solution. Chapter 5 presents the experimental evaluation performed to evaluate the proposed solution. Chapter 6 concludes the work and presents the plan for future work and current limitations.

2

Graph Processing

Contents

In this chapter we overview the the current state of the art about systems that use graphs as their data model. We identify the main characteristics of these systems, pointing out the best uses cases for each one of them. We also give some examples where graphs are used to model real world problems, identifying the main benefits of using the presented systems to solve such problems.

2.1 Graph Management Systems

Graphs are data structures composed of a set of vertices and a set of edges. An edge connects two vertices and captures a relationship between the vertices it connects. This structure has a high expressive power, allowing it to model many different kinds of systems. The rich model provided by graphs creates an elegant representation for problems that would not be easily expressed using other abstactions. Several science areas can benefit from the use of graphs to model the objects of study [12] including, among others, social sciences, natural sciences, engineering, and economics.

We provide a few examples of applications where graphs are useful. On social sciences, *social graphs* can be used to model contact patterns (edges) among individuals (vertices) [13]. *Information graphs*, or *knowledge graphs*, represent the structure of stored information. A classic example of such graph is the network of citations between academic papers [14]. *Technological graphs* can represent man-made networks designed typically for distribution of some commodity or resource, road networks and airline routes are typical examples of such graphs. *Biological networks* represent natural structures like the food web, in which the vertices represent species and edges represent prey/predator relationships [15], another example of biological networks are neural networks [16]. Finally, markets can also be modelled by graphs. Gartner [17] points to consumer web identifying five different graphs on this matter: *social*, *intent*, *consumption*, *interest* and *mobile* graphs. By storing, manipulating, and analyzing market data modeled as graphs, one can extract relevant information to improve business.

The rich model provided by graphs, together with the increased interest on being able to store and process graph data for diferent purposes led to the appearance of computational platforms dedicated to operate over a graph data model. In this dissertation we call these platforms as *graph management systems*. An important feature of many graphs that appear in current applications is their large size, which tends to increase even further as time passes. For example, in the second quarter of 2012 Facebook reported an average of 552 million active users per day. On June 2016, Facebook reports [18] an average of 1.13 billion daily active users. Another example is the largest World Wide Web hyperlink graph made publicly available by Common Crawl [19] that has over 3.5 billion web pages and 128 billion hyperlinks between them. Therefore, most systems that support the storage and processing of graph data need to be prepared to deal with large volumes of data, qualifying them as *Big Data* systems.

Several graph management systems exist, they offer a wide range of different services, including:

storing the graph on a transactional persistent way; provide operations to manipulate the data; perform online graph querying that can perform computations on a small subset of the graph and offer fast response time; offline graph analytics that allow data analysis from multiple perspectives in order to obtain valuable business information, with long operations that typically cover the whole graph and perform iterative computations until a convergence criteria is achieved. Robinson et al. [20] proposed the classification of graph management systems into two different categories:

1. Systems used primarily for graph storage, offering the ability to read and update the graph in a transactional manner. These systems are called *graph databases* and, in the graph systems' space, are the equivalent to Online Transactional Processing (OLTP) databases for the relational data model.

2. Systems designed to perform multidimensional analysis of data, typically performed as a series of batch steps. These systems are called *graph processing systems*. They can be compared to the Online Analytical Processing (OLAP) data mining relational systems.

We further detail each one of these types of systems, identifying the typical services provided by each one of them and pointing out the main differences between the two.

2.1.1 Graph Databases

Graph Databases [20] are database management systems that have been optimized to store, query and update graph structures through Create, Remove, Update and Delete (CRUD) operations. Most graph database systems support an extended graph model where both edges and vertices can have an arbitrary number of properties [21–23]. Typically, these properties have a key identifier (e.g. property name) and a value field.

Graph databases have became popular because relationships are first-class citizens in the graph data model. This is not truth for other database management systems, where relations between entities have to be inferred using other abstractions such as foreign keys. Another important aspect of graph databases systems is the query performance over highly connected data. On relational databases systems, the performance of join-intensive queries deteriorates as the dataset gets bigger. On graph databases, queries are localized to a given subsection of the graph and, as a result, the execution time is proportional to the size of the subsection of the graph traversed by the query rather than the overall graph size. If the traversed region remains the same, this allows the execution time of each query to remain constant, even if the dataset grows in other graph regions.

Table 2.1: Differences between graph databases and graph processing systems.

	Graph Databases	Graph Processing Systems
System Purpose	Provide access to business data	Produce valuable information from the business point of view by analyzing graph structure
Queries Type	Standardized and simple queries that include small graph zones	Complex queries involving the whole graph
Processing Time	Typically fast but depends on the amount of data traversed by the queries	Proportional to the entire graph size which is typically slow
Data Source	Operational data modified through CRUD operations	Graph snapshots from OLTP databases
Typical Usage	Answer multiple queries at the same time	Execute one algorithm at a time

2.1.2 Graph Processing Systems

Many graph processing algorithms, aimed at performing graph analytics, have been developed for many different purposes. Examples of some graph processing algorithms are: subgraph matching, finding groups of related entities to define communities, simulate disease spreading models and finding Single-Source Shortest Paths (SSSP). Some graph database systems also include small collections of such algorithms and are able to execute them. However, since the focus of graph databases is storage, and not processing, database implementations have scalability limitations and are unable to execute the algorithms efficiently when graphs are very large (namely, graphs with billions of highly connected vertices). In opposition, a graph processing system is optimized for scanning and processing of large graphs. Most graph processing systems build on top of an underlying OLTP storage mechanism, such as a relational or graph database, which periodically provides a graph snapshot to the graph processing system for in-memory analysis.

2.1.3 Comparison

In the previous sections, we discussed the main differences between graph databases and graph processing systems, the typical usage of each one of them and how they differ from other types of systems. This information is summarized in Table 2.1. In this work, we will focus mainly on graph processing systems. We will detail the most relevant architectures and design decisions in Section 2.3, discussing their impact on the computational and execution model as well as how they relate to each other. Before that, we will provide some relevant examples of graph processing applications.

2.2 Example Graph Processing Applications

As hinted before, graph processing is a data mining task that consists of analyzing graph data from different perspectives. The results are then used to increase business revenue, cut costs, or enable the system to provide some high level functionality based on the retrieved information. We now present real world examples were this analytical process is used.

2.2.1 PageRank

The PageRank [24] algorithm allows to assess the relevance of a web page, leveraging on the existing links (edges) between pages (vertices). The algorithm has been used by Google to implement its early search engine and is today one of the most known and well studied algorithms for this purpose.

The algorithm estimates how important a page is based on the number and quality of links to that page. It works on the assumption that important websites are likely to receive more links from other websites. For example, consider a web page A, that has pages $T_1, T_2 \ldots, T_n$ pointing to it, the importance of the page A represented by $PR(A)$ is given by:

$$PR(A) = (1-d) + d(PR(T_1)/C(T_1) + .. + PR(T_n)/C(T_n)) \tag{2.1}$$

$C(T_i)$ is the number of links going out of page T_i and d is a damping factor which can be set between 0 and 1. The algorithm is designed as a model of the user behavior, here represented as a "random surfer" that starts on a web page at random and keeps clicking on links from that page or, with a given probability d, the surfer gets bored and starts on another page at random.

This is a classic example of an iterative graph processing algorithm, it starts by assigning the same importance value to all pages, $1/N$ being N the total number of pages, and is then executed iteratively multiple times until it arrives at a steady state.

Although PageRank has become popular as web search algorithm, executed over a graph of pages and links between them, it now finds widespread applications in many other fields: Gleich [25] discusses on the aplicability of such algorithm on graphs of different domains. One of such diverse applications is Biology. A study used a similar algorithm [26] over a graph of genes (vertices), and known relationships between them (edges), to identify seven marker genes that improved the prediction of the clinical outcome of cancer patients over other state of the art methods. Other examples of the application of PageRank to different domains include recommendation systems [4], trend analysis in twitter [27] and others [28, 29].

2.2.2 Community Detection

One of the most relevant aspects of graphs that represent real world domains is the underlying community structure. A graph is composed of vertices' clusters that represent different communities, vertices on the same community have many connections between them, having only few edges joining vertices of different clusters.

Detecting these structures can be of great interest. For example, identifying clusters of customers with similar interests on a purchase network, among products and customers, enables the development of efficient recommendation systems [5]. These systems leverage on similar user purchase histories to recommend new products that the user is more likely to buy increasing revenue. In Biology, protein-protein interaction networks are subject of intense investigations, finding communities enables researchers to identify functional groups of proteins [3].

2.2.3 Single-Source Shortest Paths

Finding the shortest distance between pairs of vertices is a classic problem that finds widespread applications in several domains. Finding the shortest distance between two locations on road networks, calculating average distance between two persons on social networks or finding the minimum delay path on telecommunication networks are some relevant examples of real world applications of this algorithm. Most of these scenarios require the all-pairs shortest paths problem resolution. This problem can be trivially solved by applying the SSSP algorithm over all vertices. However, on large networks the usage of such approach can be impossible due to time constraints.

Previous work [30] proposed a landmark based approach to mitigate the mentioned problem. This approach selects a group of seed vertices that are used as landmarks. The system keeps the distances of all vertices to these landmarks, obtained by running the SSSP algorithm over the seed vertices. When the distance to any pair of vertices is requested, the distance to the landmark vertices is used as estimation of the real distance.

2.3 Architectures of Graph Processing Systems

The characteristics of graph analytical algorithms, which typically involve long operations that need to process the entire graph, together with the increasingly large size of the graphs that need to be processed, demand a dedicated system that is specifically designed to process large amounts of data on an efficient way. We now discuss typical graph processing architectures, analyzing concrete systems and detailing how they are able to process such algorithms over large graphs.

2.3.1 Distributed Graph Processing Systems

We have already noted that most graph processing systems need to be prepared to process large volumes of data. One of the approaches to address this challenge is to resort to distributed architectures and to algorithms that can be parallelized, such that different parts of the graph can be processed concurrently by different nodes. In the next paragraphs, we discuss some of the most relevant distributed graph processing systems.

2.3.1.A MapReduce

The MapReduce framework [31] and its open-source variant Hadoop MapReduce [32] are notable examples of a middleware and of a companion programming paradigm. They aim at simplifying the construction of highly distributed and parallel applications that can execute efficiently on commodity clusters. The MapReduce framework supports two operators, *map* and *reduce*, which encapsulate user-defined functions. A typical execution reads input data from a distributed file system as *key-value* pairs. The input is split into independent chunks which are then processed by the map tasks in a completely parallel manner. The generated intermediate data, also formatted as key-value pairs, is sorted and grouped by key forming groups that are then processed in parallel by the user-defined reduce tasks. The output of each reduce task represents the job output and is stored in the distributed file system.

MapReduce quickly gained popularity because of its simplicity and scalability. In particular, the middleware abstracts many lower-level details and prevents the programmers from having to deal directly with many of the issues that make the management of a distributed application complex, such as haldling failures (which are masked automatically by the framework). It is therefore no surprise that early graph processing systems, such as [33, 34], have attempted to leverage MapReduce for graph analytics. Unfortunately, experience has shown that MapReduce is not well suited for graph applications [35, 36], including experience from Google itself [6], who first introduced MapReduce. The main limitation of such model regarding graph processing is the lack of support for iterative processing. Most of graph processing algorithms are iterative, and most of them require a large number of iterations. However, in order to conduct multiple iterations over a map-reduce framework, the programmer needs to explicitly handle the process and write intermediate results into the distributed file system so that it can be used as input for the next map-reduce job. This approach has poor performance due to the frequent I/O operations and time wasted on multiple jobs' start up.

2.3.1.B Pregel

Pregel [6] is a distributed system dedicated to large-scale graph processing. It has been introduced by Google as a way to overcome the limitations of other general-purpose data processing systems used

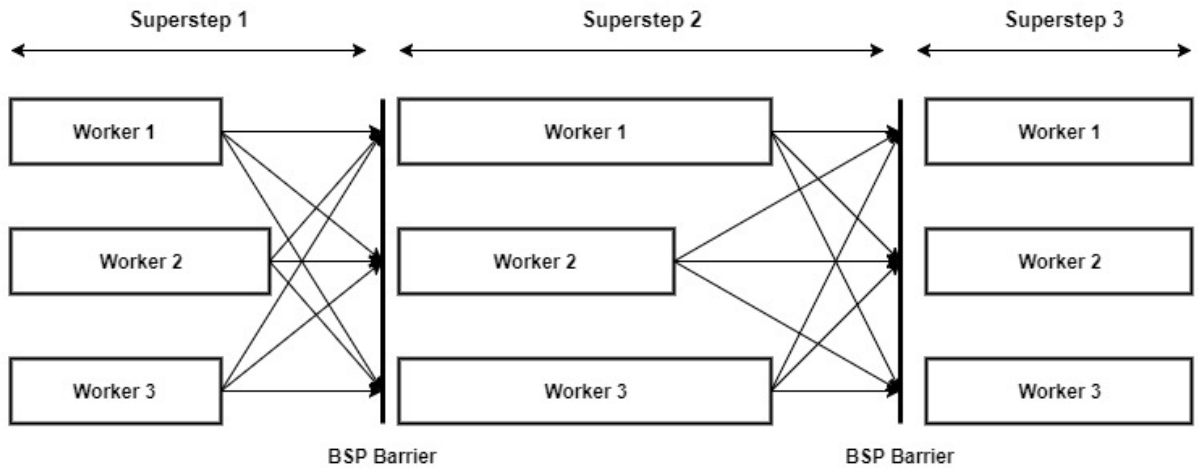


Figure 2.1: Overview of the BSP execution model.

to perform graph analytics. The system introduces a *vertex-centric* approach, also called the *"think like a vertex"* paradigm, in which the computation is centered around a single vertex. The programmer needs to define the algorithm from a vertex point of view, defining a *compute* function that will be executed conceptually in parallel at each vertex.

Pregel's computational model is inspired by Valiant's BSP model [37]. The input graph is partitioned and the computation consists on a sequence of iterations, also called supersteps, in which the partitions are processed in parallel. In each partition, the system invokes the *compute* user-defined function for each vertex in parallel. At each superstep, the function specifies the behavior of the vertices. On superstep S, the *compute* function executed over vertex V is able to: read all messages sent to V on superstep $S-1$; modify the state of V; mutate the graph structure (add or delete vertices/edges); and send messages to other vertices that will be available to them at superstep $S+1$. Typically messages are sent to the outgoing neighbors of V but may be sent to any vertex whose identifier is known. At the end of each superstep, global synchronization barriers are used to make sure that every partition of the graph has finished the current superstep before a new one can start. Figure 2.1 pictures an overview of this execution model.

Depending on the application, some vertices in the graph start the computation in *active* state and are able to deactivate themselves executing the function *voteToHalt*. Deactivated vertices are not considered for execution during the next supersteps unless they receive messages from other vertices and are automatically activated. Computation ends when all vertices have voted to halt. Pregel also introduces *combiners*, a commutative associative user defined function that merges messages destined to the same vertex. For example, when executing the SSSP algorithm, where vertices send messages to inform their neighborhood of their distance to the source vertex, we can use this function to reduce all messages directed to the same vertex into a single message with the shortest distance.

The system has a master/slave architecture, the master machine is responsible for reading the input graph, determines how many partitions the graph will have and assigns one or more partitions to each slave machine. The partitions contain a portion of the graph vertices, all the outgoing edges of those vertices (vertex id of the edge's target vertex and edge's value), a queue containing incoming messages for its vertices and the vertex state (vertex's value and a boolean identifying whether the vertex is active or not) for every vertex in the partition. The master is also responsible for coordinating the computation between slaves. The slave machines, or workers, are responsible for maintaining its assigned partition in memory and perform the computation over the vertices in that partition. During computation, messages sent to vertices on other slave machines are buffered and sent when the superstep ends or when the buffer is full.

The system achieves fault tolerance through checkpointing. At the beginning of a superstep, on user defined intervals of supersteps, the master instructs the slaves to save the state of their partition to a distributed persistent storage. When machines fail during computation, the master detects failed machines using regular "ping" messages and reassigns the partitions of the failed machines to other slaves. Computation is then resumed from the most recent checkpointed superstep.

2.3.1.C Pregel Variants

Since Pregel has been introduced several systems following the same "think like a vertex" paradigm have been proposed [10,38–41]. These systems either provide conceptually different execution models, such as *incremental* execution (discussed on Section 2.4), or introduce some system level optimizations that we now discuss.

Gather, Apply and Scatter: Gonzalez et al. [41] discussed the challenges of analyzing scale-free networks on graph processing systems. These networks have a power-law degree distribution, that is, the fraction $P(k)$ of nodes in the network having k connections to other nodes goes as $P(k) \sim k^{-\gamma}$. Most of the networks presented so far, such as social networks, citations between academic papers, protein-protein interactions, connections between web pages and many others follow these power-law degree distributions.

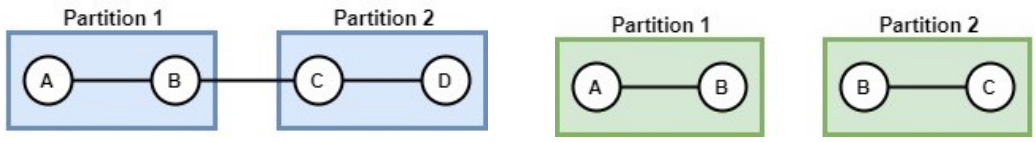
They concluded that a programming abstraction that treats vertices symmetrically, as Pregel does, can lead to substantial work and storage imbalance. Since the communication and computation complexity is correlated with vertex's degree, the time spent computing each vertex can vary widely, which can lead to the occurrence of stragglers. To address these problems, they introduced a new system called PowerGraph [41]. The system adopts a shared-memory view of the graph data, although in reality it is still partitioned, and proposes a Gather, Apply, Scatter (GAS) model where the vertex functionality is decomposed into three conceptual phases: *Gather*, *Apply* and *Scatter*.

The *Gather* phase is similar to a *MapReduce* job, using the *gather* and *sum* user-defined functions to collect information about the neighborhood of each vertex. The *gather* function is applied in parallel to all edges adjacent to the vertex, it receives the edge value, the source vertex value and target vertex value to produce a temporary accumulator. The set of temporary accumulators obtained are then reduced through the commutative and associative *sum* operation to a single accumulator. In the *Apply* phase, the value obtained in the previous step is used as input to the user-defined *apply* function that will produce the new vertex value. Finally, in the *Scatter* phase, the user-defined *scatter* function is invoked in parallel on the edges adjacent to the vertex, with the new value as input, producing new edge values.

By breaking the typical ”think like a vertex” program structure, PowerGraph is able to factor computation over edges instead of vertices, overcoming the problems mentioned before on power-law graph computation. The downside of this model is the restrictiveness imposed to the user. It forces the conceptual *compute* function (gather, sum and apply) to be associative and commutative, new vertex values must always be obtained based on neighbor's values and communication is only performed to the neighborhood of each vertex. In the original Pregel model none of these conditions exist.

Graph Partitioning Techniques: In the original Pregel system, vertices are assigned by default to partitions using a simple hash based method that randomly assigns them. This method is fast, easy to implement, and produces an almost evenly distributed set of partitions.

More sophisticated partitioning techniques have been explored. The primary goal of such techniques is to reduce the communication between partitions. On systems that follow the vertex centric approach, where vertices are partitioned across different partitions, the goal is to reduce the *edge-cut* between partitions. The edge-cut of a given assignment is given by the number of edges whose vertices are assigned to different partitions. On systems that follow the GAS approach, where edges are partitioned across partitions, the goal is to reduce the *vertex-cut* between partitions. A vertex-cut happens when two different edges, that share a single vertex, are assigned to different partitions. Figure 2.2(a) shows the edge-cut of edge between nodes *A* and *B*. Figure 2.2(b) shows the vertex-cut of vertex *B*.



(a) Edge-cut example **(b)** Vertex-cut example

Figure 2.2: Different Partitioning Techniques

Salihoglu and Widom [39] presented an experimental evaluation, on a system similar to Pregel (vertex-centric), where they tried to verify if certain algorithms have performance improvements if ver-

tices are ”intelligently” assigned. They compared the performance metrics of three different techniques, namely: i) the *random* approach; ii) a *domain-based* partitioning scheme, where domain specific logic is used to partition the graph, for example in the web graph, web pages from the same domain can be assigned to the same partition, and finally; iii) the *METIS* [42] software, that divides a graph into a given number of partitions trying to reduce the edge-cut.

The domain-based and the METIS partitioning techniques are expected to improve the system overall performance because they will, in theory, reduce the amount of messages exchanged by different workers, reducing the time wasted on message transmission and data serialization/deserialization. This reduction is achieved by assigning vertices connected by edges to the same partitions given that most messages are exchanged by directly connected vertices. The obtained results showed that, in fact, both techniques reduce the network I/O and runtime on multiple graphs with different algorithms. However, these results cannot be extrapolated to every situation, for example, the METIS approach provides high quality partitions that significantly reduce the communication between workers but it typically requires long preprocessing periods that can nullify the performance gains obtained. Therefore, the METIS technique is well suited for very long operations where the necessary preprocessing time is orders of magnitude smaller than the time spent on algorithm execution. The domain-based is a simplified version of the METIS approach that has a faster preprocessing time, with simple domain specific rules that try to foresee the communications patterns. This approach has fast preprocessing time but often produces inferior quality partitions, than those obtained with METIS, and is not easy to define over different graph domains and algorithms.

GPS [39] also introduces an optional *dynamic repartitoning* scheme. Most systems use an initial partitioning step, like the ones presented before, and vertices remain in those assigned partitions until the end of the computation. This scheme reassigns vertices during computation, by analyzing communication patterns between different workers it is able to reassign vertices in order to improve network usage. This approach decreases network I/O but due to the time spent in the exchange of vertex data between partitions it does not always improve computational time.

Powergraph [41], a system that follows the GAS model, proposes a greedy vertex-cut partitioning algorithm to reduce the vertex-cut of the produced partitions over randomly allocated edges. The algorithm goes as follows: Being the edge (u, v) the next edge to assign to a machine. $M(v)$ the set of machines which vertex v spans, that means that every machine on $M(v)$ has previously been assigned an edge that contains v. The algorithm chooses the edge (u, v) placement following the rules that follow:

Rule 1: if $M(u)$ and $M(v)$ intersect, the edge should be assigned to a random machine in the intersection.

Rule 2: if $M(u)$ and $M(v)$ are not empty and no not intersect, the edge should be assigned to one of the machines of the vertex that has the most unassigned number of edges.

Rule 3: If only one of the vertices has been assigned, choose a machine from the assigned vertex.

Rule 4: If neither vertex has been assigned, assign the edge to the least loaded machine.

Experimental results show that this method is able to reduce vertex-cut by a factor of up to $4\times$ the one obtained by random edge assignment.

Asynchronous Mode: The BSP model provides a simple and easy to reason about execution model where computation is performed as an iterative process. Global synchronization barriers between each iteration (superstep) ensure worker synchronization.

Although synchronous systems are conceptually simple, the model presents some drawbacks. For example, most graph algorithms are iterative and often suffer from the ”straggler” problem [43] where most computations finish quickly but a small part of the computations take a considerably longer time. On synchronous systems, where each iteration takes as long as the slowest worker, this can result in some workers being blocked most of the time waiting for other slower workers to finish, thus causing under-utilization of system resources. Graph algorithms requiring coordination of adjacent vertices are another example where the synchronous execution model is not suitable. The *Graph Coloring* is one of such algorithms, it aims at assigning different colors to adjacent vertices using a minimal number of colors. Some implementations of this algorithm may never converge in synchronous mode, since adjacent vertices with the same color will always pick the same colors based on the previous assigned ones. These types of algorithms require an additional complexity in the vertex program in order to converge [44].

To overcome these problems, asynchronous execution models on graph processing systems have been proposed [7, 40, 45, 46]. In this model there are no global synchronization barriers and vertices are often scheduled to execution in a dynamic manner, allowing workers to execute vertices as soon as they receive new messages. Some algorithms exhibit asymmetric convergence rates, for example, Low et al. demonstrated that running PageRank on a sample web graph, most vertices converged in a single superstep while a small part (about 3%) required more than ten supersteps [45]. The dynamic scheduling of computation can accelerate convergence as demonstrated by Zhang et al. [47] for a variety of graph algorithms including PageRank. The insight behind such statement is that if we update all parameters equally often, we waste time recomputing parameters that have already converged. Thus, using the asynchronous graph processing model, some graph zones that take longer to converge can get executed more often and we can potentially accelerate execution time.

Although the asynchronous execution model outperforms the synchronous one in some situations, that is not always the case. Xie et al. [44] studied the situations that better fit each execution model. They concluded that the asynchronous model lacks the message batching capabilities of the synchronous

model, making the latter more suitable for I/O bound algorithms, where there is a high number of exchanged messages and most time is spent on communication. On the other hand, the asynchronous model can converge faster and favors CPU intensive algorithms, where workers spend most of the time inside the *compute* function. The authors also demonstrated that due to heterogeneous convergence speeds, communication loads and computation efforts in different execution stages, some algorithms perform better with different execution models in different execution states. For example, they shown that the asynchronous mode performs better in the beginning and end of the SSSP algorithm, but synchronous model has superior performance during the middle of execution. This insight motivated the design of PowerSwitch [44], a graph processing system that adapts to execution characteristics and dynamically switches between the asynchronous and synchronous execution model.

2.3.2 Single Machine Graph Processing Systems

While the access to distributed computational resources is nowadays eased through the cloud computing services, managing and deploying a distributed graph processing system is still a challenge. The characteristics of the distributed infrastructure have a large impact on the execution model and force complex design decisions, such as the choice of the right communication paradigm, graph partitioning techniques and fault tolerance mechanisms. Also, in some systems, the communication among the processing nodes can become the main performance bottleneck.

To avoid the challenges above, and to mitigate the impact of the communication latency, recent graph processing systems have been designed to support graph processing on a single machine. Typical implementations either leverage on secondary storage to handle huge graphs or use multicore server class machines, with tens of cores and terabyte memories, to do so. We now analyze some of the graph processing systems are designed for a single machine.

2.3.2.A Polymer

Several multicore machines adopt a Non-Uniform Memory Access (NUMA) architecture, defined by several processor nodes, each one of them with multiple cores and a local memory. These nodes are connected through high-speed communication buses and are able to access memories from different nodes, offering a shared memory abstraction to applications. However, when nodes access memory from other nodes (remote memory), the data must be transferred over the communication channel, which is slower than accessing local memory. Further, the latency on remote accesses highly depends on the distance between nodes.

Polymer [8] is a single machine NUMA-aware graph processing system that provides a vertex-centric programming interface. The system tries to align NUMA characteristics with graph specific data and computation features in order to reduce remote memory accesses. NUMA machines are treated as a

distributed system, where processing nodes act as multi-threaded worker machines, and uses similar execution models to the ones presented before for distributed systems. The graph is partitioned across these nodes in order to balance the computational effort and, given that most accesses are performed to the assigned graph partition, reduce the remote accesses performed by each processing node. To further reduce the remote accesses performed, with the purpose of obtaining graph topology data, the system uses *lightweight vertex replicas*. Following this approach, every partition has a replica of all vertices from all other partitions. These replicas are immutable and only contain partial topology information, such as the vertex degree and some neighboring edges' data.

The system evaluation on a 80-core NUMA machine shows that Polymer outperforms other state-of-the-art single machine graph processing systems, such as Ligra [36] and Galois [48], that do not leverage on NUMA characteristics. Although it is expected that Polymer outperforms state-of-the-art distributed graph processing systems, its evaluation lacks such comparison.

2.3.2.B GraphChi

GraphChi [7] is a system that leverages on secondary storage to allow efficient processing of graphs with huge dimensions on a single consumer-level machine. The system proposes PSW, a novel method for very large graph processing from disk. PSW processes graphs in three stages:

Loading the Graph: In this phase, the vertices of the graph under analysis are split into P disjoint intervals. For each interval, there is an associated shard that stores all edges whose *destination* vertex belongs to the interval. Edges are stored in shards ordered by their *source* vertex. Intervals are chosen in order to produce balanced shards that can be loaded completely into memory.

The graph processing phase is then performed by processing vertices one interval at a time. To process one interval of vertices, their edges (in and out) and their associated values must be loaded from disk. First, the shard associated with the current interval being processed is loaded into memory, this shard contains all in-edges for the vertices in the interval. Then all out-edges must be loaded from the other shards, this process is eased by the fact that edges are stored ordered by their source vertex thus only requiring $P - 1$ sequential disk reads from other shards. Another important aspect of the shards is that edges are not only ordered by their source vertex but also by interval. This means that out-edges for interval $P + 1$ are right after the ones for interval P, thus when PSW starts processing another interval it slides a window over each of the shards. In total PSW only requires P sequential disk reads, a high-level image of the described process is given in Figure 2.3.

Parallel Updates: After loading all edges for the vertices in the current interval being processed, the system executes the user-defined functions at each vertex in parallel. In addition to the vertex values,

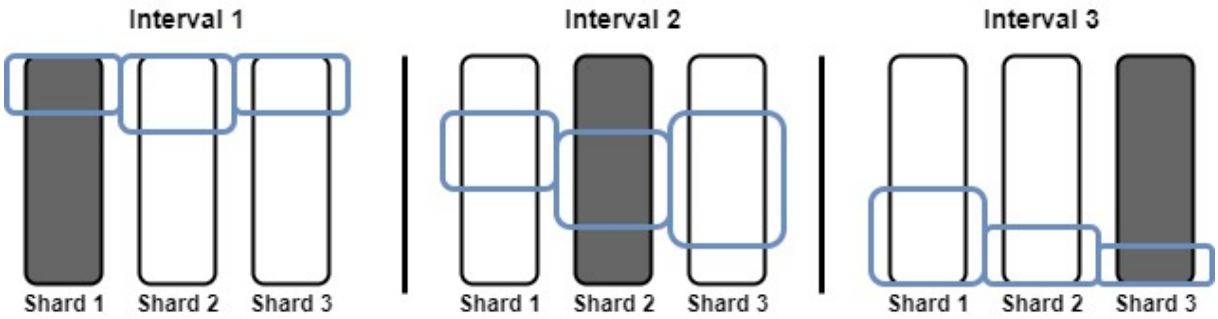


Figure 2.3: Example of the PSW method.

the system also allows for modifications on the edge values. To prevent race conditions between adjacent vertices, the system marks vertices that share edges in the same interval as critical and they are processed in sequence.

Updating Graph to Disk: After the update phase, the modified edge values and vertex values need to be written back to disk in order to be visible to the next interval execution. This phase is similar to the first one but with sequential writes instead of reads. The shard associated with the current interval is completely rewritten, while only the active sliding window of each other shard is rewritten to disk.

Although the system is simple to use, and allows the processing of large graphs on a single commodity machine, its evaluation shows that it performs poorly when compared to state-of-the-art distributed graph processing systems.

2.4 Incremental Graph Processing

The increasing interest in the information gathered from graphs, motivated by its business value, boosted the rise of many graph processing systems. The domains of interest, such as commerce, advertising, and social relationships are highly dynamic. Consequently, the graphs that model these domains present a fast-changing structure that needs to be constantly analyzed. However, most of the systems analyzed so far present a static data model that does not allow changes to the graph data. To cope with dynamic graphs in these systems we have to follow a strategy similar to: (1) perform the computation over a static version of the graph while storing the updates that occur during the computation; (2) apply the updates to the previous graph version; (3) repeat the computation over the new graph. This approach has several drawbacks, for example, it increases the time necessary to see the impact of some graph mutations in the information retrieved from the graph processing step, an extra complexity is necessary to manage the update storage and most of the computations performed in the new graph

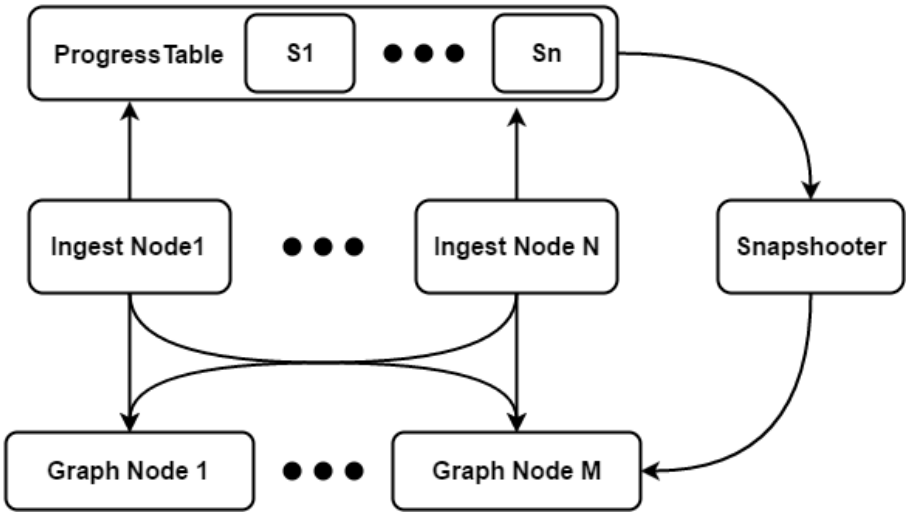


Figure 2.4: Kineograph system's overview.

version are repeated from the previous versions.

To address these issues, a new class of graph processing systems has emerged. Incremental graph processing systems allow graph data to change over time and allow computation to adapt to the new graph structure without recomputing everything from scratch. We now analyze some systems that implement this approach and discuss the system level details specific to it.

2.4.1 Kineograph

Kineograph [10] was one of the first systems to take the incremental approach into consideration. The system is able to process an incoming stream of data, representing operations that modify the graph structure, and accommodate graph algorithms that assume a static underlying graph through a series of consistent graph snapshots.

Figure 2.4 shows an overview of Kineograph. *Ingest nodes* are the system entry point, they process raw data feeds encoding operations that modify the graph. They are responsible for: translating such raw data into concrete graph operations (add or remove nodes/edges) that can span multiple partitions; assign them a sequence number; and distribute the operation to *graph nodes*. Graph nodes are the entities responsible for storing the partitioned graph data in memory and perform the computations. Each ingest node has its own sequence number that it uses to timestamp operations. After timestamping new operations, ingest nodes update their sequence numbers on a central *progress table* and send the operations to the graph nodes that will process them. Graph nodes do not update the graph right away, periodically, a *snapshooter* instructs all graph nodes to apply the stored graph updates based on the current vector of sequence numbers in the progress table. The end result of this commit protocol is a graph snapshot.

Kineograph follows the typical "think like a vertex" paradigm where computation proceeds by processing across vertices and the data of interest is stored along vertex fields. After producing the new graph snapshot, each graph node has its own incremental graph engine. It is responsible for detecting changes in the partition from the last snapshot and trigger user-defined *update* functions to compute new values associated with modified vertices or *initialization* functions to newly created vertices. The system uses the BSP execution model, it proceeds in supersteps and starts by processing the *update* and *initialization* functions on the vertices that were affected by graph modifications. At each modified vertex, if the value changes significantly, the system will propagate the changes to the neighbors, what will schedule them for execution on the next superstep. The propagation proceeds until no status changes happen across all vertices, what designates the end of the computation.

Regarding fault tolerance, the system has several mechanisms to deal with different types of faults. The most relevant for our work, and that we now detail, is the failure of a graph node. The system uses replication to reliable store graph data. The original graph is partitioned in small partitions and each partition is replicated across k graph nodes and can tolerate up to f failures, where $k \geq 2f + 1$. Ingest nodes then use a simple quorum-based mechanism to replicate graph updates, they send the updates to the replicas and consider them reliable stored if they receive responses from at least $f + 1$ replicas. Regarding the computational values at each vertex, the system uses a primary-backup replication strategy where the primary copy of the vertex executes the update function and then sends the results to the replicas. This way it avoids wasting computational resources by executing the update functions at each replica of the partition. If the master replica fails a secondary takes it place without any additional effort.

2.4.2 GraphIn

GraphIn [11] proposes Incremental-Gather-Apply-Scatter (I-GAS), a novel method to process a continuous stream of updates as a sequence of batches. This method is based on the GAS programming paradigm [41] and allows the development of incremental graph algorithms that perform computation on the modified portions of the entire graph data.

The system enables evolving graph analytics programming by supporting a multi-phase, dual path execution model that we now describe. First, the system runs a typical graph processing algorithm over a static version of the graph. It could use any of the above mentioned systems to provide the expected results for this phase. The system then uses an user-defined function, that receives the set of updates received since last time it was executed, and determines which vertices should be considered inconsistent. Typical vertices are the ones affected by edge's addition or removal or direct properties' changes. Sometimes a large portion of the graph may become inconsistent. In this situation, the system does not achieve any performance benefit over static recomputation and may even result in performance

degradation due to the overheads associated with incremental execution. To deal with such situations, the system allows the user do define heuristics that help the system decide between proceeding to incremental execution or perform a static recomputation. For example, in Breadth First Search (BFS), the vertex depth could be used to define a heuristic that chooses to perform a full static recomputation if there are more than N inconsistent vertices with depth lesser than 2 (closer to the root of the BFS tree). If the computation proceeds to the incremental execution, the I-GAS engine proceeds in supersteps by executing the scheduled vertices to execution, initially with the vertices that were marked as inconsistency and then with the vertices that receive messages during computation.

The system evaluation reports speedups of $407\times$, $40\times$ and $82\times$ over static recomputation across several datasets on algorithms like Clustering Coefficient, Connected Components and BFS, respectively, reporting a maximum throughput of 9.3 million updates/sec.

2.4.3 Heterogeneous Update Patterns

Most incremental graph processing systems [9–11, 49], take a stream of update events to build a series of graph snapshots at which incremental graph processing algorithms run on top. These update events can be of different forms, for example, create new vertices, add new edges between existing vertices and change static properties of vertices or edges (such as the weight of a given edge). An interesting observation, that crosses several application domains, is the obtained update patterns where some vertices receive more updates than others. For example, on social networks where graphs model the existing relationships between users, it is expected that the most active users, or the ones with more connections, produce a higher number of update events than those less active or with less connections.

The *preferential attachment* network behavior model [50], also called the *rich get richer* model, models the growth of networks into the power-law degree distributions observed in reality. In this model of network behavior, new vertices that come into the system are more likely to create connections with other vertices that already have a higher number of connections. The same happens with new connections that are created within existing nodes. Easley and Kleinberg [51] discuss the suitableness of such model on social circles and connections between web pages. On these domains, new vertices are more likely to follow other popular vertices (social networks) or create links to other popular pages (web pages network). This further supports the idea that some vertices will receive and produce more updates than others, thus creating different computational needs at different graph partitions.

Ju et al. [9] took this insight into consideration when designing iGraph, an incremental graph processing system. It provides a programming model similar to Kineograph [10] and, in addition, it uses a *hotspot rebalancer*. This entity is responsible for, at user defined intervals, analyze the number of update events that come into the system. It then assigns heat values to each partition, based on the number of updates received by each vertex in the previous time interval. Based on this values, it uses a greedy

algorithm to move vertices between partitions. This seeks to balance the work performed by each slave node creating partitions that receive approximately the same number of updates. The system follows a reduce vertex-cut partitioning technique. So, in addition to the heat values of each vertex, the partition rebalancer also takes into consideration vertex-cut changes when performing reassignments.

2.5 Deployment Issues

The systems studied in this dissertation can be used for a wide-variety of applications with very different time constraints. For example, real-time applications, such as fraud detection [52] or trend topic detection [27], need to extract timely insights from continuous changing graphs. Other systems, such as recommendation systems [4, 5], do not require changes to the underlying graph to be reflected immediately in the produced results, thus allowing longer time intervals between the update of results. Incremental graph processing systems have proven to have the best of two worlds, for fast changing graphs they are faster to reflect changes in the produced results [10, 11] than static recomputation and still can benefit from the update batching on applications without real-time requirements. Thus, these systems have a computational model well suited for continuous deployments, that are important to enable the processing of dynamic graphs, rather than single shot executions over static graphs that are of little use.

Continuously deployed graph processing systems should be able to adapt to the underlying graph dataset computational requirements and avoid wasting resources and consequently money. Most of the graph processing systems, including the ones discussed so far, do not make assumptions regarding the deployment environment. Most of them are not production level systems and it is hard to predict the deployment environments that would be used for that purpose. However, from the existing implementations and evaluation details we observe that a large majority of them use IaaS platforms, such as Amazon Web Services (AWS) [1], for deployment. The flexibility, nonexistent upfront investment or maintenance costs, ease of management and payments based on usage make the IaaS platforms an appealing deployment environment for these and other types of systems.

Dindokar and Simmhan [53] have studied the possibility of reducing the costs of deploying a distributed graph processing system on an IaaS environment to run static graph computations. Their work leverages on graph computations that have different groups of vertices active in different supersteps, such as BFS or SSSP. At the beggining of each superstep, their system analyzes the number of partitions that will be active for that iteration and determines the number of machines necessary to perform the computation, moving partitions between machines if needed and terminating the unnecessary ones. The proposed solution is able to reduce the costs for those types of computations but the extra compu-

[1] AWS: https://aws.amazon.com/

tational effort at the beginning of each superstep, associated with data movement and time to startup or terminate machines, has impact on the computation runtime. Another system limitation is the payments granularity provided by the IaaS platforms. Usually supersteps take seconds or minutes to execute, thus in order to reduce the costs of deployment on a superstep basis the IaaS platform needs to have a granularity of payments in the same order of magnitude as the average execution time of the computation supersteps (minutes or seconds). This allows the system to achieve its goals on platforms with payments per minute of machine usage, such as Microsoft Azure[2], where the evaluation of the system is performed. For platforms with larger payment granularities, such as Amazon that has payments per hour of machine usage, nullify the cost reductions achieved by terminating and starting machines on a superstep basis.

Previous work, both on static and incremental graph processing, ignore the diversity of resources and services made available by cloud providers that offer infrastructure as a service. The unawareness of such conditions translates into lost opportunities to reduce deployment costs. For example, transient resources are much cheaper than the reserved ones, however, they can be withdrawn at any moment without warning. These resources allow systems that can tolerate dynamic availability conditions to reduce costs significantly. These resources can be offered on markets where prices change based on the current demand. For a given instance type, its value is expected to rise under a very high demand and go down as soon as the demand decreases. As different instances have independent supply and demand conditions, the more we consider the more alternatives we have if the price for a given instance rises. Although graph processing systems seem to have neglected the usage heterogeneous resources and different renting services, that is not the case for other types of systems. In the next chapter we introduce some systems that leverage these aspects. We explain how they do it and why the way they do it is inadequate for the graph processing model.

2.6 Summary

In this section we detailed the current state of the art for systems that leverage the graph data model. We identified two types of these systems, graph databases, used primarily for graph storage, and graph processing systems, designed to process analytical jobs over the graph data. Focusing on the latter type, we identified some graph processing applications, such as PageRank and SSSP, giving some real world use cases for these algorithms. We then surveyed the most common architectures for these systems. Distributed settings, typically supported by a master/slave architecture where the graph data is partitioned across different machines, and single machine systems that leverage on secondary storage or server class machines to process large graphs. We detailed the common vertex centric approach,

[2]Microsoft Azure: https://azure.microsoft.com

supported by the BSP execution model, and presented the most common variations, both in conceptual level (edge centric approach) and in execution model (asynchronous model). We then presented the incremental graph processing model. This model is designed to allow the computation over dynamic graphs that are observed in reality. We then pointed out some deployment issues that the existing systems seem to neglect, therefore, missing opportunities to significantly reduce the operational costs.

3

Transient Resource Usage

Contents

In this chapter we give some examples of existing transient resource markets offered by cloud providers. We detail the main characteristics of these markets, pointing out the main advantages and disadvantages that come with transient resource usage. We then give some examples of systems that leverage these types of resources on different matters and also detail why their strategy is inadequate for the incremental graph processing task.

3.1 Transient Resource Markets

Datacenters, in large scale public cloud providers, are well provisioned with resources to handle the most demanding usage conditions. However, such resources are subject to usage spikes that lead to the existence of non-profitable idle resources. To reduce the number of such idle resources, cloud providers allow users to rent them at lower prices with the proviso that they could be withdrawn at any moment. Some relevant examples of markets for these types of resources are described in the following sections.

3.1.1 Amazon EC2 Spot Instances Market

The Amazon EC2 spot market[1] allows users to bid on spare EC2 instances, also called spot instances. The main characteristics of the AWS spot market are:

- It provides resources in a auction based system. More precisely, each spot instance has an associated market value, known to all users, based on the current demand for that type of instance. Users place bids for that instance and the request is accepted if the current market price is bellow the bid.
- The spot instances' market is not a free market, that is, users do not pay the bid value they placed but the current market value for that instance.
- Upon receiving a spot instance, it is terminated under two conditions: (i) the current market value goes above the placed bid and the user is not billed for the last hour of usage or (ii) the user terminated the instance manually and it is billed for the entire hour even if he only used it for a few minutes.
- The price paid for each hour of usage is set as the market value for that instance in the beginning of the billing hour. Therefore, even if the price rises above the initial market price, as long as it stays bellow the placed bid, the user is only billed the value initially set for that hour.
- Once a bid is placed and the spot instance is granted the user cannot change the bid value.

[1] AWS: https://aws.amazon.com/

3.1.2 Google Preemptible Instances

Google's preemptible instances are the equivalent in the Google Compute Engine (GCE)[2] to the spot instances in the AWS environment. These instances are similar to the previously described spot instances with the following differences: The price charged for each preemptible instance is fixed. The system offers a 30 second warning before terminating instances. The maximum contiguous period of time that an user is able to run a preemptible instance is limited to 24 hours.

Generally, GCE avoids preempting too many instances from a single customer and will preempt instances that were launched most recently. This strategy tries to, in the long run, minimize the lost work of each user. The system does not bill for instances preempted in the first 10 minutes of machine usage.

3.2 Leveraging Transient Resources

As described before, cloud providers allow users to use spare resources at very reduced prices with the condition that they can be taken at any moment. Systems that leverage this type of resources have to specifically handle such dynamic resource availability conditions, but are capable of significantly reduce deployment costs. In the following sections we describe relevant examples of systems that leverage transient resource usage. We detail how they adapt the system to effectively deal with the characteristics of the underlying resources.

3.2.1 Pado

Pado [54] is a general data processing engine that leverages idle resources, of over-provisioned nodes that run latency-critical jobs, to run batch data analytic jobs on a mix of reserved and transient resources.

The system work is divided in two different phases, *compilation* phase and *execution* phase. First, in the compilation phase, the system receives as input the description of the job that is going to be processed. The job description is represented as a Directed Acyclic Graph (DAG) of computations where nodes represent computational tasks and edges represent topological dependencies between tasks. For example, if we consider a simple Map-Reduce job, its DAG would have two types of nodes, map tasks and reduce tasks, having directed edges from map tasks to reduce tasks. Still in the compilation phase, the system processes the received DAG to decide which tasks should be assigned to transient or reserved resources. The system focus on assigning the computations that are most likely to cause large recomputation chains once evicted to the reserved resources and the rest to transient resources. The system model assumes tasks to execute on containers that are terminated after completing the assigned

[2] GCE: https://cloud.google.com/compute/

task. The output produced by tasks is moved to containers of other tasks that need it as input and is lost after the container is terminated. Under this assumption, the best candidates to be assigned to reserved resources are the tasks that require more input from other tasks. If a task is evicted, it is necessary to relaunch previous tasks on which the evicted tasks depends, possibly starting long recomputation chains.

In the second phase, the system acquires the reserved and transient containers and assigns the tasks accordingly to the execution plan generated in the previous phase. A master node is responsible for coordinating the entire process and, if necessary, calculate recomputation chains under transient resources' evictions.

Although effective to this offline analytical job, where the input job is well known, the system model would not be effective in the incremental graph processing model. In this model, computation is highly dependent on the type of updates received, making the computation chain unpredictable. Even if possible, this strategy would add an extra overhead to the update batch processing phase. It would be necessary to analyze the entire computation chain and dynamically allocate partitions to transient or reserved machines depending on what vertices are going to be executed for each batch.

3.2.2 Proteus

Proteus [55] exploits transient resources to perform statistical machine learning. This type of systems iteratively process training data to converge on model parameter values that, once trained, can predict outcomes for new data items based on their characteristics. The system uses a parameter server architecture where workers process training data independently and use a specialized key-value store for the model parameters' values. These values are updated through commutative and associative aggregation functions. Therefore, updates from different workers can be applied in any order, offloading communication and synchronization challenges. This simple model allows Proteus to use transient resources to perform the training data work and use reserved resources, free from evictions, to store the current model state. Therefore, if transient resources are evicted no state is lost and the input processing job can be resumed when workers become available again. For the graph processing model, the relation between data and computation is stronger. The computation flow is highly dependent on the graph structure and on computation values' changes. This bound prevents the clear separation between data and computation followed by Proteus [55]. In addition, the graph processing task requires the constant communication between different workers, not necessary in this model.

Proteus uses a resource allocation component, BidBrain, to manage the transient resources in the AWS environment. BidBrain's goal is to reduce the cost per unit work. The system analyzes the work that it has to perform and, based on current market values for different instances, the system analyzes historical market prices for those resources and determines the probability of a given instance being

evicted within the hour under a certain bid. With this knowledge, BirdBrain is able to estimate the amount of work that a given instance is able to perform under a certain bid and determine the assignment that is expected to process the work with minimal cost.

Without time constraints to process the training data, the system explores the possibility of free computing. As explained on section 3.1.1, if a spot-instance is terminated before the end of the billing hour, the user is refunded the price it paid for that hour. To capture this possibility, BidBrain estimates the cost (C_i) for a given instance i as:

$$C_i = (1 - \beta_i) * P_i + \beta_i * 0 \tag{3.1}$$

Having P_i as the current market value and β_i as the probability of failure for instance i in the next billing hour. β_i value is estimated from the historical market price data. After estimating the cost for a given instance, BidBrain estimates the expected work it will perform by:

$$W_i = \Delta t_i * \tau * \phi \tag{3.2}$$

Δt_i represents the expected useful compute time of a given instance, estimated based on historical data. τ represents the work performed by unit of time by each instance type, proportional to the number of cores of that instance. ϕ is a scalability overhead decaying factor, empirically defined. The system then estimates the expected cost per work of a given instance i as:

$$E_i = C_i / W_i \tag{3.3}$$

At periodic time intervals, usually at the end of each billing hour, BidBrain builds a list of possible allocations it can do. These allocations consist on pairing different instance types with different bid prices. After constructing the set of possible allocations, the system computes the cost per work for the current allocation and for the current allocation plus each of the possible generated allocations. If one of these allocations reduces the expected cost per work of the current assignment, BidBrain sends the allocation request to AWS.

BidBrain goal is to reduce the expected cost to process a finite workload. This is done by creating an assignment that is expected to reduce the cost per work (Eq. 3.3). This means that at some periods in time the system may acquire more resources, more expensive for that particular hour, if it expects them to reduce the time to process the remaining work 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 BidBrain's optimization goal not well-suited for that particular model. Furthermore, Proteus assumes a relaxed model where no time constraints are required and the unique goal is to reduce the processing costs. This allows moments in time where there are no instances to process

the training data if the expected cost per work is too high. For incremental graph processing systems, their availability must always be guaranteed. Further, Proteus model assumes that any type instance is able to, faster of slower, process the training data (captured by parameter τ on Eq. 3.2). In the graph processing model partitions have computational needs that may prevent them to be assigned to certain instances.

3.3 Summary

In this chapter we presented some relevant examples of transient resource markets. We described spot instances from Amazon, made available on an auction based market, and the preemptible instances from the Google Compute Engine. We identified the main characteristics of these resources, that can be very cheap but present an unpredictable life time. We then gave the example of two systems that leverage this type of resources. Namely, the Pado system, a general data processing engine that uses a mix of transient and reserved resources to process batch data analytic jobs. The system assigns computations that are more likely to cause large recomputation chains once evicted to reserved resources and the rest to transient resources. We also presented Proteus, a system that leverages transient resources to perform statistical machine learning and process the training data. The system presents a bidding component for the Amazon spot market that acquires and releases machines to minimize an optimization function for the deployment cost of the system.

4

Hourglass

Contents

In this chapter we detail the Hourglass system, our solution to fill the gap identified between existing incremental graph processing systems and the target deployment environment. We start by presenting the new challenges that arise from the heterogeneous and transient resource usage. We then present the system's architecture and identify its core modules, specifying their functionality and implementation details.

4.1 Challenges

The usage of transient resources brings some challenges to the system design. The dynamic price and availability conditions of these resources raise new challenges for the system. A heterogeneous resource approach, that considers multiple types of machines, further raises questions about how to leverage such diversity. We decided to tackle the AWS environment, as we believe it to be the most challenging for transient resources. The current solution could then be easily ported to other cloud providers with simpler transient resource usage mechanisms. The main challenges that we identified are the following:

1. How to handle the dynamic market prices for transient resources. At a certain moment a machine type can have a discount of 90% over the reserved price and some moments later have a price that is $10\times$ 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 create balanced partitions for it. In a heterogeneous environment there are machines with different processing capabilities.

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 and the dynamic price conditions, that will probably make the system have to change its deployment configuration periodically, will impact the system performance. This challenge encompasses a sub-challenge of trying to control the eviction rate of these machines.

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

4.2 System's Overview

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

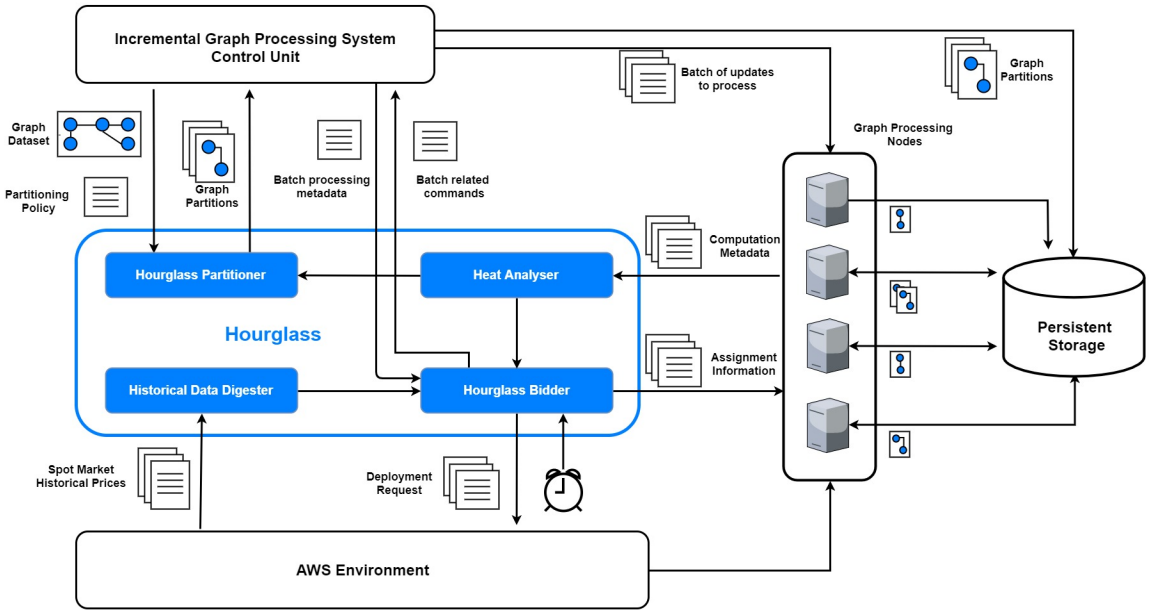


Figure 4.1: Hourglass system's overview

is to ensure the underlying incremental graph processing system to be deployed in a set of machines that are able to withstand the computational task and, at the same time, in 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 assumes the underlying system to follow a typical distributed graph processing architecture [6, 9, 10, 38], where the graph is divided in partitions across different machines that perform the computational task, further called graph processing nodes, and have a centrailized control unit. We also assume the underlying incremental graph processing system to have the typical execution flow [9–11], divided in two periods: (i) The update batching period; and (ii) the update processing period.

Hourglass manages the deployment of graph processing nodes, as they represent the largest share of the operational costs. Graph processing systems may have other entities, with specific tasks, that may be deployed in different machines. For example, Kineograph [10] has ingest nodes, the system's entry point to update operations. Also, the iGraph [9] system has the Hotspot Rebalancer and HotSpot Detector. Hourglass does not take these entities into consideration during the deployment process. All these entities are beyond Hourglass scope and should be managed separately.

The underlying incremental system is a black box to Hourglass. We assume these systems to implement a set of commands (Section 4.2.1) that allow Hourglass to interact with them.

Hourglass has four core components – illustrated in Figure 4.1 – namely:

- **Heat Analyzer** is a component similar to the one in iGraph [9]. It 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. The vertices' heat is a fundamental information in the partitioning task. Hourglass assumes the computational metadata, that allows *Heat Analyzer* to calculate such information, to be already available when the partitioning task is being performed. If not, it is always possible to run the system with a random partitioning technique [6] and reassign vertices later on.

- **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. The duration of each period is measured in hours as Amazon bills on a hour-basis. 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.

In each time interval the system is possibly terminating instances and starting new ones, as it expects them to be a better deployment for the next time period. Instances have ephemeral storages that are lost when they are terminated. This makes it necessary to use an external persistent storage service, accessible to all instances, where instances being terminated checkpoint the current state of their partitions and newly created instances read their assigned partitions from. For the current AWS prototype the Elastic Block Store (EBS)[1] service is used.

In the following sections we describe the mentioned components in more detail.

[1] https://aws.amazon.com/ebs/

4.2.1 Hourglass Interaction Commands

In this section we list the set of commands that a system using Hourglass must provide. We detail for each command its characteristics and which of the Hourglass' components uses it. In the following sections, where all core modules are detailed, we explain how they use this commands and for what purpose. The commands are the following:

- **Obtain computation metadata.** The graph processing nodes must allow Hourglass to request metadata associated to the graph processing computational task. This metadata includes the number of updates received by each partition and information about the communication performed to and from each machine. This information is used by the *Heat Analyzer* component (Section 4.4).
- **Request batch information.** The control unit of the incremental graph processing system must allow Hourglass to request information about the current batch of updates being created. The system must also provide information about previously processed batches, such as their size and processing time. This information is used by *Hourglass Bidder* to predict the expected processing time based on batch characteristics. More details given on Section 4.6.3.
- **Request to delay or anticipate batch.** The control unit is expected to be the entity responsible for controlling the batch processing. Hourglass may find it useful to anticipate or delay such processing (Section 4.6.3). Therefore, it should be possible to Hourglass to request the incremental system to anticipate or delay the start of the next batch processing.
- **Request graph processing nodes to checkpoint current state.** As will be detailed on Section 4.6 and Section 4.7, checkpoint the current state to persistent storage is a very important task. Graph processing nodes must allow Hourglass to decide when this task should be done.
- **Revert to a specific partition checkpoint.** The fault tolerance mechanisms followed by Hourglass, described on Section 4.7, may require certain partitions to revert to a previously created checkpoint. Graph processing nodes must allow Hourglass to request such state changes.
- **Reprocess a specific batch of updates.** Also due to the fault tolerance mechanisms (Section 4.7), Hourglass may require the system to redo the processing of a certain batch of updates. This should be allowed by the incremental graph processing system control unit.

4.3 Hourglass Partitioner

The incremental graph processing system is able to delegate the graph partitioning process to Hourglass. Previous work [41, 56, 57] focused on the importance of creating partitions that minimize the

communication between machines and improve the locality of vertex communication. By reducing remote communication, these approaches are able to improve the computational task execution time. For example, Powergraph [41], that follows the GAS model, focus on reducing the number of partitions spanned by each vertex (vertex-cut) has it involves a synchronization step between the replicas of each vertex. GrapH [57] introduces a partitioning strategy aware of distinct traffic volumes across different edges, normally edges between vertices that are executed more often, and different network link costs between different machines. The goal of the system is to avoid frequent communication over expensive network links. However, previous work neglected the impact that the created partitions can have in the deployment costs.

As demonstrated on Section 5.4, performance and deployment cost are objectives that are maximized under different circumstances. Partitions that improve one of them usually worsen the other. Hourglass allows the user to specify if the produced partitions should maximize performance, cost reductions or try to find a good compromise between the two. Hourglass' partitioning technique is based on heat. In the following sections we explain how we define heat and how it is used for the partitioning process. We also explain the properties that influence the execution time and deployment cost.

4.3.1 Partition Heat

Machines that perform the graph processing task influence the computation based on their available resources. The vertex-centric approach, that centers the computation around vertices, or the GAS model, that focus computation around edges, provide highly parallelizable execution models. Therefore, the execution time to process a given partition is reduced as the number of Central Processing Unit (CPU) cores available increases. Memory is another important resource for the graph processing task. Graph processing systems usually create in-memory representations of the assigned partition to perform the computational task. This computational task creates huge amounts of information, typically proportional to the number of edges, that needs to be propagated across vertices, usually in the form of computational messages. The high number of messages created, that need to processed in the computational task, further increases the amount of memory necessary. Incremental graph processing systems also receive update operations, representing mutations to the underlying graph structure, that need to processed and therefore also require an in-memory representation.

So, different types of machine's resources can impact the system's ability to process a given dataset in different ways. Due to specific application characteristics and system implementation details, the computational task associated to the graph processing phase can be bound by one type of these resources. For example, if we found memory to be a more restrictive factor (memory bound), if a machine that is assigned a given partition does not have the minimum amount of memory necessary to withstand the computational task the computation either aborts, due to insufficient memory, or the system as to use an

out-of-core approach, such as [7], that severely hinders the system's performance. So, for this specific case we must consider the machine memory capacity when performing the partitioning assignment.

Hourglass specifies the heat of a partition, also referred as the partition size, as the minimum level of resources that a machine must have in order to withstand the computational effort associated to the graph processing task for that partition. These heat values can then be measured, for example, in gigabytes if we consider a memory bound application, dedicated CPU cores, if a CPU intensive application is considered or even a combination of different types of machine resources. For simplicity, and also due to the characteristics of our implemented prototype of an incremental graph processing system, in the rest of the report we assume the computational task to be memory bound and measure the heat (partition size) in gigabytes. The partition heat is not only related to the number of vertices and edges that it contains. There are other computational factors that influence the minimum necessary amount of memory to process a partition.

Based in our experiments we find that the total heat of a dataset, given by the sum of the heat of all its partitions, is kept roughly the same as it is splited into a different number of partitions. This means that the total ammount of resources in the system has to be the same, regardless of the number of used machines. Currently, the heat of a dataset is measured empirically. However, we believe that it can determined as a function of the number of vertices, number of edges, application type, graph structure and the individual heat of the vertices in that dataset. The heat of each vertex is determined in runtime by Hourglass following a strategy similar to the one used by [9]. The heat of each vertex is influenced by the number of updates it receives and by the number of messages it receives and sends during the incremental computation. As the incremental function executed in each vertex is the same, the execution time of each vertex should be proportional to the number of messages received and sent.

4.3.2 Heat Distribution

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 Batch Execution Time: If the system prioritizes execution time, the created partitions' heat should be as high as the capacity of the largest machine allows. 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 batch execution time to increase. For example, if we consider a dataset with a total of 30Gb heat. One machine with 30Gb memory and 8 vCPU's, processing a partition with 30Gb heat, is expected to outperform two machines with 15Gb memory and 4 vCPUs each, processing the same partition divided in two partitions with 15Gb heat each. This is

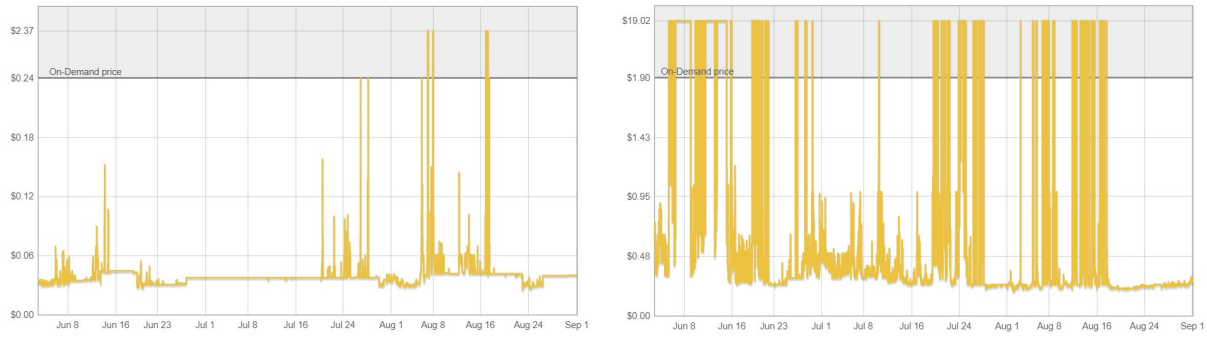
due the fact that the total amount of resources in the system is exactly the same (30Gb memory and 8 vCPU's) and there is the extra effort of transmitting messages between partitions that are assigned to different machines.

As the number of partitions increases, the number of messages that need to be transmitted across different machines also increases. This leads to an increasing deterioration of the system's performance. Although the strategies mentioned on Section 4.3 substantially reduce the communication done between partitions, as they seek to create partitions that maximize local communication, the quality of such partitions also decreases as the partitioning factor increases. For example, for an edge-based partitioning, Sun et al present a partitioning analysis [58] that shows how the vertex replication factor (vertex-cut) increases as the number of partitions in which we split the original dataset increases. For a vertex-based partitioning, Slota et al. do a very detailed analysis [59] and show how the edge cut ratio (ratio of edges whose vertices are on different partitions) significantly increases as the number of partitions also increases, for different datasets and partitioning methods. In short, these partitioning techniques that seek to reduce communication between machines, help to improve the system's performance over other approaches, like random partitioning, but do not prevent it from deteriorate as the number of partitions in which the original dataset is partitioned increases.

Based on this, when the batch execution time should be prioritized, Hourglass chooses to create partitions as large as the maximum allowed by the available machines.

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 changing to on-demand resources if the spot-price rises beyond that price. Furthermore, instances with large memory capacities usually present the spot market prices that change the most. Figure 4.2(a) and Figure 4.2(b) show the spot market prices of EC2 C4.XL instances, with 4 vCPUs and 7.5 Gb memories, and C4.8XL instances, with 36 vCPUs and 60Gb memories, respectively, over a period of three months. These figures show how the spot market price for the largest instance is considerably more unstable than the price for the other instance type. So, by creating large partitions we are not only reducing the number of possible instances to which they can be assigned, but also



(a) Spot market prices for C4.XL instances.

(b) Spot market prices for C4.8XL instances.

Figure 4.2: Spot market prices for C4.XL and C4.8XL instances over a period of three months

assigning them to instances whose prices are constantly changing and, therefore, do not always provide a good discount over the on-demand 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. In Section 5.3 we explain the process to select the ideal partition heat to maximize savings.

Although Hourglass creates small enough partitions that can be assigned to any type of instance, this does not mean that the system, under the right market price conditions, will not choose to deploy such partitions on large machines that improve execution time. By creating small partitions we are only saying to Hourglass Bidder that we prioritize the cost reductions over the performance improvement. So, if the cost of using only large machines is the same of using only small machines to achieve the same level of resources, Hourglass prioritizes large resources and assigns multiple partitions to the same large 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. This would allow them to be assigned to a larger number of instance types and still use large enough partitions to improve execution time.

Hourglass explores a different approach to achieve a better compromise between cost and execution time. Previous work [9–11] has focused on creating partitions with equal heat values in order to balance

Table 4.1: Description of datasets used in the communication analysis

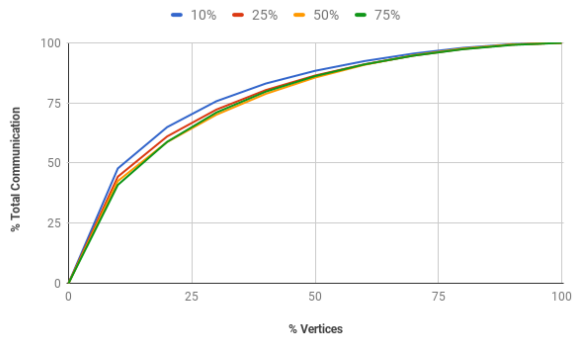
Dataset	#Nodes	#Edges	Description
Social Network [61]	3,072,441	117,185,083	Orkut social network
Web Network [62]	875,713	5,105,039	Google web graph
Road Network [62]	1,965,206	2,766,607	California road network

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.

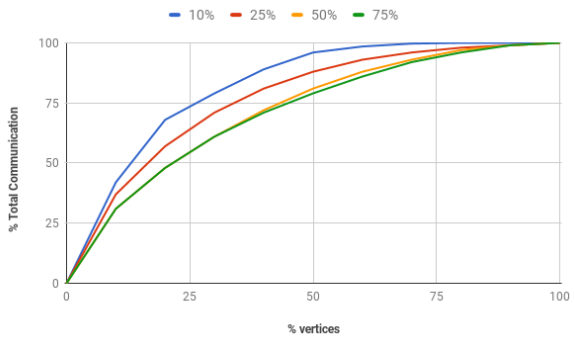
As graphs present irregular structures, one should expect the communication to follow such irregular patterns. This leads to the existance of distinct heat values across different vertices, observed by [9]. Based on this, we classify the vertices in the system as *hot vertices*, that have a huge roll 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 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 Hourglass 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 [60], 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 [58, 59].

However, in order to this hot/cold separation to be useful, the number of hot vertices should be

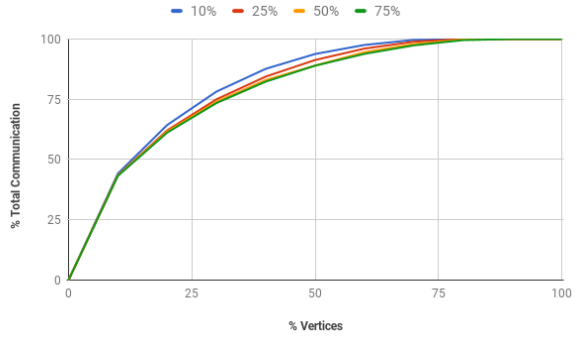


(a) SSSP communication analysis.

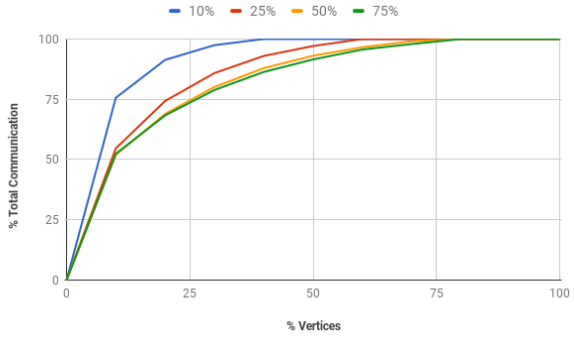


(b) PageRank communication analysis.

Figure 4.3: Communication analysis for a social network graph.

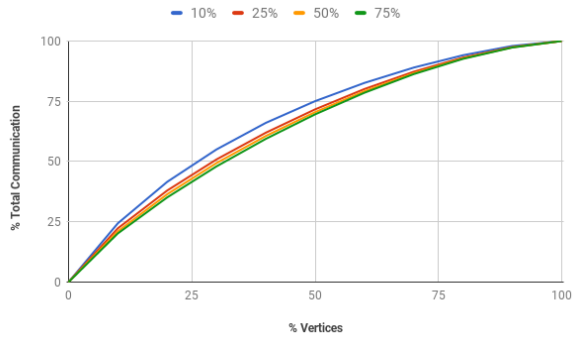


(a) SSSP communication analysis.

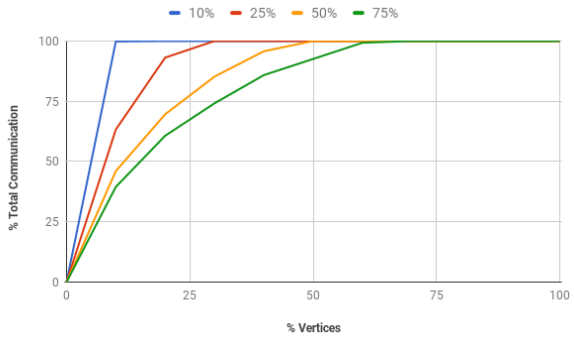


(b) PageRank communication analysis.

Figure 4.4: Communication analysis for a web pages network graph.



(a) SSSP communication analysis.



(b) PageRank communication analysis.

Figure 4.5: Communication analysis for a road network graph.

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 4.3, Figure 4.4 and Figure 4.5 show a communication analysis performed over three datasets of different sizes and types, namely a social network, a webpages network and a road network, described on Table 4.1. The figures show the relation, on a vertex-centric approach, between the percentage of the number of vertices (x-axis) and the percentage of the total communication performed during the computation (y-axis). All datasets were analyzed under two different incremental applications, namely the SSSP and PageRank. The figures represent the execution of one update batch, with randomly generated update operations, for the mentioned incremental applications. The lines in each figure represent different runs where the percentage of vertices that are considered to receive updates are changed. The percentages considered (10%, 25%, 50% and 75%) simulate different update patterns, as systems may have a small subset of vertices receiving most updates or have a more evenly distributed update distribution across vertices. The results shown for each line are the average of 5 runs, where the vertices considered to receive updates are chosen randomly at the beginning. For example, if we take Figure 4.3(b), we can infer that 75% of the total communication is performed by only 25% of the vertices for the PageRank application when updates are distributed to 10% of vertices.

Based on this results, for different dataset types, update distribution patterns and applications, 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.

4.4 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* component to create valid assignments between machines and partitions, preventing the outburst of machine capacity.

4.5 Historical Data Digester

When dealing with transient resources, the eviction of such resources is a major concern. High eviction rates may lead to situations where the computation is not able to proceed, as the system is frequently trying to recover from failures. A system that is able to predict the probability of failure for a given instance under certain circumstances is therefore of big importance. Hourglass' *Historical Data Digester* component is responsible to perform this task.

As explained on Section 3.1.1, spot-instances work on an 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 [55, 63, 64] 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 4.6 we explain how they are used.

4.5.1 Bid Oracle Service

One of the main concerns when requesting spot-instances is to find the right bid to place. Low bid values may protect the system against expensive spot-instances, whose price has risen a lot since the machine was first rent, but may increase the number of evictions. High bid values decrease the eviction rate but may represent large costs for the system. The trade-off between the two represents a compromise between the maximum possible paid price (bid value) and the probability of failure for that instance.

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. Find all datapoints where the target spot instance type had the current market value.
2. If the collected set of datapoints is not statistically significant, for example, it has less than one hundred datapoints, add all datapoints where the market value is equal to the current market value ± 0.01 cents (the minimum price unit on AWS price values). Repeat until a significant number of datapoints are collected.

3. For each datapoint, create a set that only contains it and add all other datapoints that follow it in the historical timeline for the duration that was given as input. Each of these groups of datapoints are past time periods with the same duration as the one that is intended and that also start with a market value similar to the current price.

4. For each set obtained in the previous step, find the highest price in that set. This value represents the bid that should have been issued for that period in order to get to the end of it without any eviction for that instance type.

5. Add all values obtained in the previous step to a list and sort it in ascending order.

6. Return the value that is in the $\frac{P}{100} * N$ nth position in the list obtained in the previous step. Where P is the intended probability of success ($0 \leq P \leq 100$) and N the size of the list. This value is the P-th percentil of the bid values in the list. This means that, for the historical periods considered, if a bid value equal to the obtained value was used, a percentage of P periods would have come to the end without any eviction.

4.5.2 Cost Estimation Service

This service is used to estimate the cost of an instance for a given period of time. As explained on Section 3.1.1, 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 the average cost of similar time periods in the historical data. The process of estimating the result price goes as follows:

Steps 1 to 3 are the same as the ones presented in Section 4.5.1.

4. For each set obtained in the previous step, sum the prices at the beginning of each billing hour, considering the time of the first datapoint as the first hour. These value represents the actual cost paid for instance in that period in the historical data.

5. Return the average of all the values obtained in the previous step.

4.6 Hourglass Bidder

Hourglass is responsible for, at user defined time periods, find the best possible assignment for the existing partitions, based on the current market values and always respecting the associated constraints. To solve this problem, Hourglass uses a sub-component called *Hourglass Bidder* that is responsible for,

in the end of each time period, cast an optimization problem to find the best possible assignment under the current circumstances. This component interacts with the deployment environment to terminate existing resources that are not going to be used in the next time period and request the new ones. Typically, incremental graph processing systems have pause periods of several minutes to batch incoming updates before they are processed. These periods, when no computation is being performed in the graph, are the ideal moment to perform the exchange of partitions between resources. In the next sections we detail the transitioning period and how *Hourglass Bidder* chooses the deployment for the next time period.

4.6.1 Deployment Problem Definition

Before the end of the current time slot, Hourglass has to find the deployment solution for the next time period. This process results in a solution composed of two sets, S_1 and S_2. The set S_1 contains $(T; B; I)$ triplets, meaning that an instances of type T should be requested under bid B and its identifier is I. Set S_2 contains $(I; P_i)$ pairs, meaning that partition P_i is assigned to the machine with identifier I. Set S_1 is used to request the target instances to the AWS environment. Set S_2 is used to inform the newly requested machines their assigned partitions.

When finding the best possible deployment, Hourglass Bidder has to consider several aspects that are important to ensure the proper functioning of the system while meeting user defined goals. The aspects considered are:

- **Eviction Rate:** In the AWS auction system, bids have a direct impact in the probability of failure for a given machine (Section 3.1.1), higher bids lead to lower eviction rates but may significantly increase the price paid. Lower bids increase cost savings but may result in higher eviction rates. Different systems may have different concerns regarding this aspect. Some systems may prefer to maximize cost savings and tolerate the performance degradation due to frequent machine evictions. Others may prefer to reduce evictions and sacrifice some of the possible cost savings achieved. Therefore, the eviction rate that the system is willing to accept has impact in the placed bids for each instance and, consequently, in the deployment cost.

  In order to meet the expected eviction rate, Hourglass allows the user to specify the probability of failure that the system is willing to accept by specifying the maximum probability of failure for each partition. The target bids for the intended probability of failure are obtained using the bid oracle service from the *Historical Data Digester*.

- **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

Table 4.2: Hourglass optimization problem parameter description.

Parameter	Description	Source
M	Set of all available machines	Calculated by Hourglass
P	Set of all partitions	User Defined
T	Set of all time intervals	Slotted time domain (every hour for example)
x_t^{pm}	$\in [0;1]$, 1 if partition p is assigned to machine m on time t	Objective variables to define
β_m	Total capacity of machine m	Obtained from Amazon
σ_p	Heat factor of partition p	Hourglass defines it based on past computations
$\hat{\mu}_t^m$	Estimated cost of machine m for period t	Cost Estimation Service
γ_t^m	$\in [0;1]$, 1 if machine m is going to be used on time t	Objective variable to define

that partition. When assigning partitions, the system must ensure that every machine is assigned a set of partitions that do not outburst its capacity.

- **Expected Instance Cost:** Although bid values influence the eviction probability of a given instance, they do not represent the actual cost paid by the user (Section 3.1.1). The price paid in each billing hour is set as the spot market price in the beginning of the billing hour. If we consider time slots of one hour, the current spot market price for the target instance is a good estimation for the paid price, as it is unlikely that the price changes significantly from the time the machine is selected and the instance is started (moment that Amazon sets the price). The cost estimation service, from the *Historical Data Digester*, is used to estimate the price of a given instance for time slots greater than one hour.

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.

4.6.2 Deployment as an Optimization Problem

The deployment problem is formulated as an optimization problem. Hourglass Bidder uses a constraint solver to find cheapest possible assignment before the start of a new time period. The parameters used to define the problem are described in Table 4.2. 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 values x_t^{pm}, true if partition p is assigned to machine m on time t and false otherwise.

The set of available machines (M) is defined as the minimum set of machines that contains the minimum number of each type of instance that is necessary to hold every partition in the graph. For example, if we consider three types of instances: (i) type S with 5Gb memory; (ii) type M with 10Gb memory; and (iii) type L with 20Gb memory. If we have three partitions with 10Gb heat each. The set M of available machines would contain three machines of type M and two machines of type L. This set contains all the machines necessary to consider all possible permutations of instances to contain the three partitions. Machines of type S are not in the set as it is impossible for them to hold any of the existing graph partitions. The goal is then to choose from the available machines the ones that minimize the deployment cost and are able to hold all partitions.

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

$$\text{minimize:} \quad \sum_{m \in M} (\gamma_t^m * \hat{\mu}_t^m),\ t \in T, m \in M \tag{4.1}$$

$$\text{s.t.} \quad x_t^{pm} - \gamma_t^m \leq 0,\ \forall m \in M, \forall p \in P, t \in T \tag{4.2}$$

$$\sum_{m \in M} x_t^{pm} = 1, \forall p \in P \tag{4.3}$$

$$\sum_{p \in P} (x_t^{pm} * \sigma_p) \leq \beta_m, \forall m \in M \tag{4.4}$$

The x_t^{pm} variable is the key decision variable for this problem. Constraint 1 in the above formulation ensures that if at least one partition is assigned to a machine, the cost of this machine is considered in the cost computation. Constraint 2 ensures that each partition is assigned to one and only one machine. Constraint 3 ensures that partitions assigned to a given machine do not outburst its capacity. Hourglass uses CPLEX[2] to solve the above formulation. The current implementation allows the system to obtain the solution up to a problem size of 4000 partitions under one minute.

4.6.3 Transition Periods

Before the start of a new time slot, *Hourglass Bidder* has to decide the set of machines that are going to be used in the next period. The system keeps track of the end of each period and makes sure it has a deployment solution before it arrives. It terminates machines that are not going to be used in the next time period, before the start of the new billing hour.

The process of terminating existing machines, start new ones and write/read checkpoints to/from persistent storage possibly reduces the time that the system is processing incoming updates. Typically, graph processing systems [9–11] use pause periods between incremental computations to batch incoming update operations. In order to minimize the impact of these transitions, the system adopts a

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

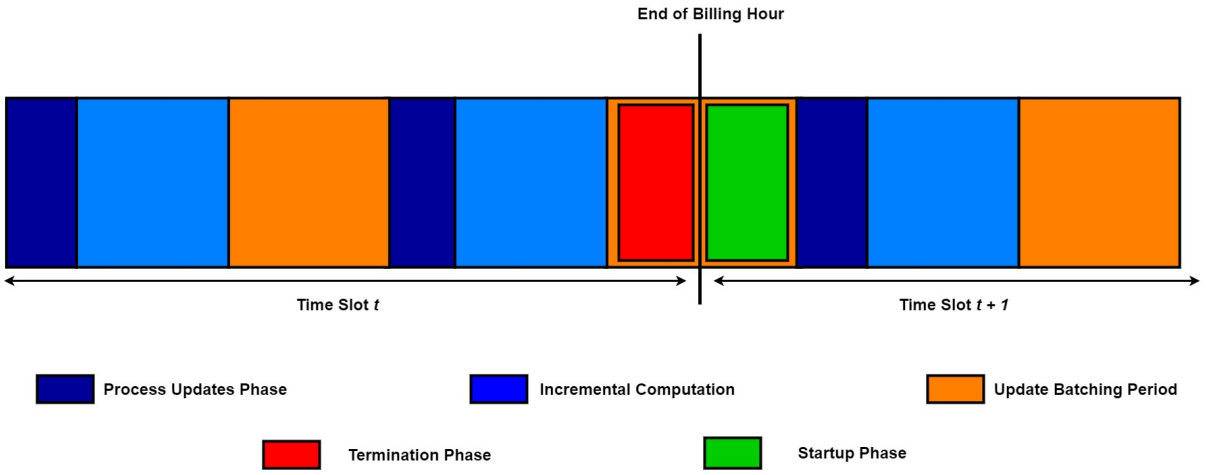


Figure 4.6: Transition process overview

strategy that tries to match these pause periods with the transition period. During the transition phase it is necessary to do the termination phase, where machines checkpoint their current partition state to persistent storage and are terminated. It is important to make sure this termination phase is finished before the beginning of the next billing hour, as Amazon charges the machine hours in the beginning of each billing hour. In the beginning of the new time slot, new machines are started and they read their assigned partitions from persistent storage (startup phase). Figure 4.6 illustrates this process.

Currently, *Hourglass Bidder* uses a strategy that looks into past operations in order to predict the duration of each phase. In the following paragraphs we explain the process.

When to start the termination phase: One of the most important objectives is to make sure instances that are going to be replaced in the next time slot, have checkpointed their partition to persistent storage and have been terminated before the beginning of the next time slot (that matches the beginning of a new billing hour). In order to do so, *Hourglass Bidder* has to start the termination phase in a moment that it believes to have enough time to finish the termination phase, before the next time slot begins.

The number of seconds that the termination phase has to start before the end of the current time slot is given as:

$$\hat{t}_{\text{checkpoint}} + \hat{t}_{\text{termination}} + \epsilon \tag{4.5}$$

Where $\hat{t}_{\text{checkpoint}}$ and $\hat{t}_{\text{termination}}$ represent estimations, based on past executions, of the time it takes to checkpoint the current set of partitions and terminate the instance. To estimate the duration of the checkpoint phase, Hourglass bidder estimates the time that each machine will take to checkpoint their set of partitions and selects the maximum between those estimations as the estimated duration of the checkpoint phase. The variable ϵ represents a time safety margin.

Align pause periods with the transition phase: In order to minimize the impact of those transition periods, it is important to match the pause periods with the transition phase. To match these periods, *Hourglass Bidder* has the possibility to dynamically change when a new computation phase should start. Usually, update batching periods have a fixed maximum duration and a limit in the number of updates that can be batched before a new computation phase starts. They then work on a whichever comes first strategy. *Hourglass Bidder* is able to start a new computation (Section 4.2.1) without any of these conditions being satisfied. By knowing the number of updates that are already batched, the system is able to estimate, based on past operations, the duration of the computational phase. Therefore, if the start of the termination phase is close, the system may chose to anticipate the computation phase if it believes it will make the pause period match the transition phase. As working on estimations may not ensure that the system is able to meet its deadlines, a backup strategy is necessary for those situations where the computation phase is not finished before the end of the billing hour. If the currently ongoing computation phase is not finished by the time it was expected to do so. The system chooses to abort computation and start the termination phase. The computation is then restarted after the new time slot begins.

In a worst case scenario, where the billing hour arrives before some machines are successfully terminated the system pays the next billing hour for the machines being terminated. Although we expect those situations to be rare. For future work, we intend to use a reassignment strategy that finds a new deployment where those machines that failed to meet their deadlines are mandatory.

Multiple time lines: In the beginning, all machines are started at the same time and we consider the start of the billing hour as the start time of the machine with the soonest boot time. As we are leveraging transient resources, although *Hourglass Bidder* tries to reduce the probability of evictions, it is possible that some machines are evicted while others are still available. When this happens, the failed machines have to be replaced. This creates different time lines for the multiple groups of machines, with different billing hours, that need to be considered independently. Therefore, *Hourglass Bidder* starts to process those time lines independently, following the above mentioned process. Figure 4.7 illustrates this situation.

In the worst case scenario, we would have a machine for each partition, all with different time lines. This would not only impact the deployment cost but also make it difficult to match the batching periods with the transitioning phase for each time line. However, from our experience, its rare to have more than one extra time line. This is mainly due to two factors. First, the deployment process often chooses groups of machines from the same type with the same bid, as they probably represent the best cost/capacity compromise for that moment in time. For this situation, if an eviction occurs, it will be for the entire group of machines. Also, the process that is followed when evictions occur, detailed in Section 4.7.1, seeks to

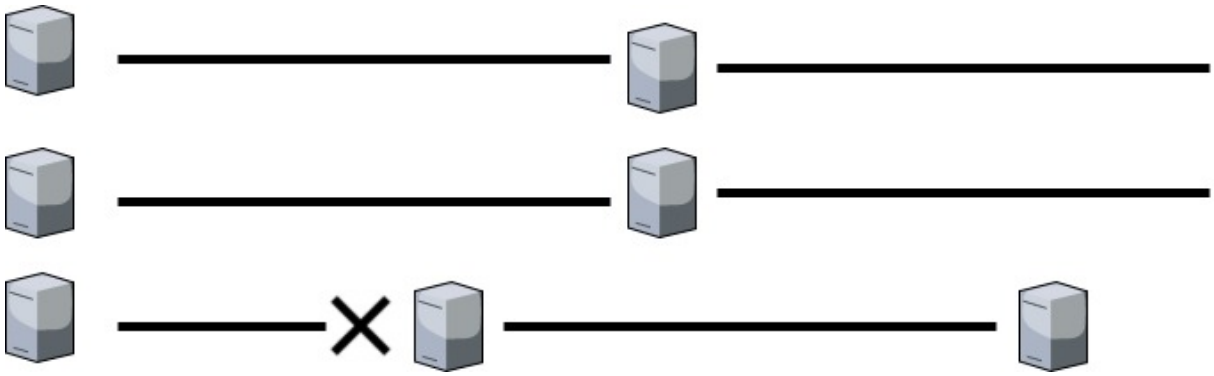


Figure 4.7: Multiple time lines example

assign partitions from those failed machines to machines with spare capacity. This helps to reduce the number of independent time lines.

4.7 Fault Tolerance

Hourglass has two different fault tolerance protocols. One to the Hourglass system itself and another to the machines under its control. As explained in Section 4.2, the system assumes that only the graph processing entities, holding graph partitions and executing the computational task, are deployed in these machines.

Hourglass System 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 its four components. We consider addressing these concerns orthogonal to the contributions of this work. Nevertheless, in a real implementation of the system, this components would need to be replicated. Standard machine replication techniques [65] can be employed to make each component robust to failures.

Graph Processing Nodes Systems that leverage transient resources are more susceptible to machine failures. Considering this, our goal is to cope with evictions without significantly affecting Hourglass' performance and without introducing a significant memory or computationally overhead during steady state. Traditionally, graph processing systems use simple checkpoint based strategies to save the current computational state to persistent storage. As explained in Section 4.6.3, Hourglass already requires machines to save checkpoints to persistent storage during the transition phases.

Considering this, Hourglass uses a checkpoint based fault tolerance protocol to recover the partitions' state. Regarding update operations, these requests are received from client applications and cannot be lost until their effects are reflected on a reliably stored graph state. Hourglass assumes the underlying

graph processing system to have its own fault tolerance protocol, to ensure that no update is lost until it is processed and its effects are reliably stored. There are several ways for this to be done. For example, if we consider a system with an architecture similar to Kineograph [10], where update operations come into the system through nodes dedicated to that same task (ingest nodes). One could consider a replication based strategy where updates are considered reliably stored if replicated with success to $k + 1$ ingest nodes, for a model that tolerates up to k ingest node failures. One could also consider a strategy that stores updates to persistent storage. Although this strategy would possibly have a considerable impact in the system's update acceptance throughput.

4.7.1 Recovery Protocol For Graph Processing Nodes

When these machines fail, either because they were evicted or due to any other type of failure, they have to be replaced in order to continue execution. For this case, the recovery protocol is the following:

- If the system was processing a batch of updates when the failure occurred do:
  1. Reassign the failed partitions.
  2. Revert all partitions' state to the one in the last checkpoint.
  3. Start processing a new batch considering all updates received since the last checkpoint was performed.
- If the system is currently on a pause period and a transition phase is taking place do:
  1. Find all partitions that failed before their last state was saved to persistent storage.
  2. If all partitions have finished their checkpoint do nothing and let the transition process continue normally.
  3. If some partitions were not checkpointed to persistent storage, let the transition process continue, after it finishes, revert all partitions' state to the last checkpoint available for the failed partitions and process all update operations since then.
- Finally, if the system is currently on pause period and no transition phase is taking place do:
  1. Find all partitions that failed.
  2. If no computation occurred since the last checkpoint for every failed partition follow the process to reassign failed partitions and finish.
  3. If the last checkpoint for some of these partitions is not the most recent state, revert all partitions' state to the last checkpoint available for the failed partitions and process all update operations since then.

The process to reassign failed partitions is the following:

1. Check if some of these partitions can be assigned to one of the still available machines. If any, a greedy algorithm is used to select partitions to the existing spare capacity.
2. For the remaining unassigned partitions, follow the process described in Secion 4.6;

The time to recover from faults is reduced if the system performs checkpoints periodically. Typically, these systems have update batching periods large enough to allow the system to checkpoint its current state before a new batch starts to be processed. This allows the system to checkpoint its state every time a new update batch is processed without affecting the computational task. This translates into faster recovery periods and, if an update replication strategy is followed, helps reducing the number of replicas that need to be kept in memory, as they can be discard when their effects are reliably stored.

4.8 Summary

In this chapter we identified the challenges for solutions leveraging transient resources on heterogeneous environments. We then presented the Hourglass system, our solution to fill the gap between existing graph processing systems and the deployment environment. We identified our system's placement and interaction assumptions against the underlying graph processing system. In this chapter we also overviewed the Hourglass architecture and detailed the core components of the system. First, the *Hourglass Partitioner*, responsible for the partitioning task of the initial dataset, based on its heat. We defined the heat of the dataset as the minimum level of machine resources that is required to do the associated graph computational task. We also identified the partitioning characteristics necessary to improve batch execution time, decrease deployment cost or find a good compromise between the two, introducing the hot/cold separation approach for heterogeneous environments. Second, the *Heat Analyzer*, that measures the individual heat of all vertices in the system, based on the computational task communication data and update distribution patterns. Third, the *Historical Data Digester* that uses historical spot market values to predict bid values for target eviction rates, and estimate the cost of a given instance type for a specific period in time. Fourth, we detailed the *Hourglass Bidder* component, responsible for controlling the periodic reconfiguration of the system and choose the deployment configuration. The deployment problem is modeled as an optimization problem, whose objective is to minimize the deployment cost function, subject to the associated partitioning assignment constraints. We also show how we try to minimize the reconfiguration impact by matching these transitioning periods with batching periods. Finally, we presented the used fault tolerance mechanisms based on state checkpointing, frequently used on other graph processing systems.

5

Evaluation

Contents

Hourglass main goal, as a deployment engine for incremental graph processing system, is to reduce deployment costs without significantly affect the system's performance. As discussed on Chapter 4, 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, periodical reassignment and target eviction rate, have on the possible savings achieved by Hourglass and its performance. Our goal is to understand the best configuration for the system under different market conditions, user constraints and underlying graph processing system characteristics.

5.1 Experimental Setup

To perform the experimental evaluation we integrated our system with the Amazon environment. In the following sections we describe the setup used during the evaluation.

5.1.1 Cost Analysis Simulator

All costs presented on this chapter 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 (650 hours) for the *us-east* region. Unless otherwise stated, the simulator emulates the behavior described on Section 3.1.1, it uses the historical bidding strategy with a 25% probability of failure and considers one hour time slots.

5.1.2 Instance Types

The experiments carried out consider five different types of instances from the compute optimized instance family provided by Amazon. Table 5.1 describes the instances used. All instances run Ubuntu Server 14.04 LTS.

Table 5.1: AWS compute optimized instance description.

Instance	vCPU	Mem (GiB)	On-Demand Price per Hour
c4.large	2	3.75	$0.1
c4.xlarge	4	7.5	$0.199
c4.2xlarge	8	15	$0.398
c4.4xlarge	16	30	$0.796
c4.8xlarge	36	60	$1.591

5.1.3 Incremental Graph Processing Applications

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 [10] system described on Section 2.4.1.

Experiments use two different applications described in the following paragraphs.

5.1.3.A Single-Source Shortest Paths

This application intends to keep the updated distances from all vertices in the graph to the source vertex as the graph mutates. Vertices keep as state the current distance to the source vertex, as well as the previous vertex on the shortest path to the source vertex. If the vertex receives the information that another neighboring vertex has a shortest distance it updates the shortest distance and sends the new information to the other neighbors. If the vertex receives the information that either the previous vertex on the shortest path has been removed or the edge to that vertex has been removed or changed its weight, it pulls the distances from all neighbors to choose the shortest distance again.

5.1.3.B TunkRank

TunkRank [66] is an algorithm similar to PageRank [24] that is used to estimate the influence of certain users on social networks. In this model vertices represent users and edges represent 'following' relations between users. The influence of a user X is defined as:

$$\text{Influence}(X) = \sum_{Y \in \text{Followers}(X)} \frac{1 + p * \text{Influence}(Y)}{|\text{Following}(Y)|} \tag{5.1}$$

The used implementation is similar to the one presented on [10].

Table 5.2: Graph dataset description.

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

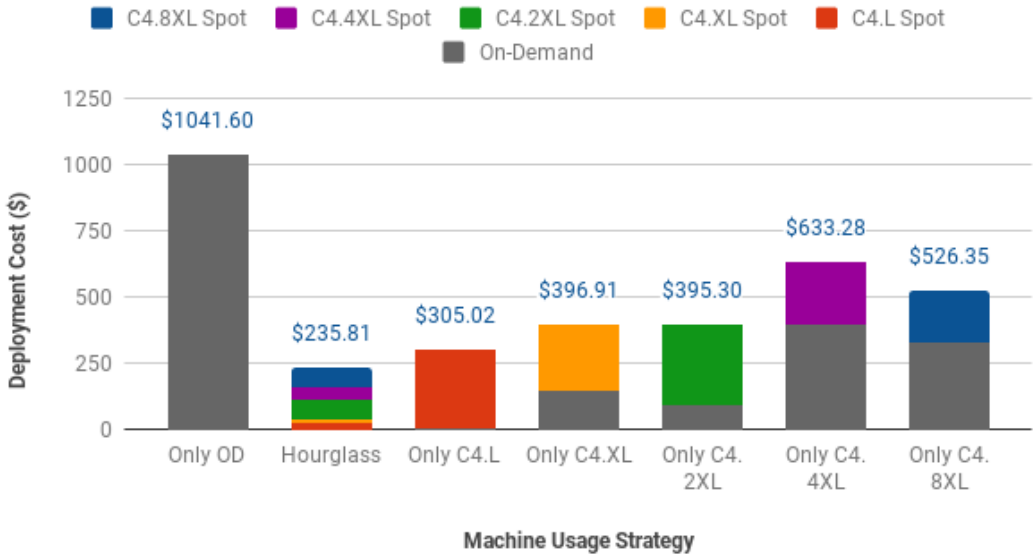


Figure 5.1: Cost reductions over different machine usage strategies

5.1.4 Graph Datasets

Table 5.2 describes the datasets used in the experimental evaluation. The syntetic graph was obtained using the R-MAT [67] generator with the same parameters used by benchmark graph500[1] to generate scale-free graphs.

5.2 Heterogeneous Transient Resource Usage

Using transient resources, Hourglass should be able to significantly reduce the deployment costs compared to the usage of only on-demand 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 spot-instances types further improves the cost savings achieved over only considering a single type of spot-instance.

Figure 5.1 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*

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

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 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.

5.3 Number of Partitions for Homogeneous Heat Distributions

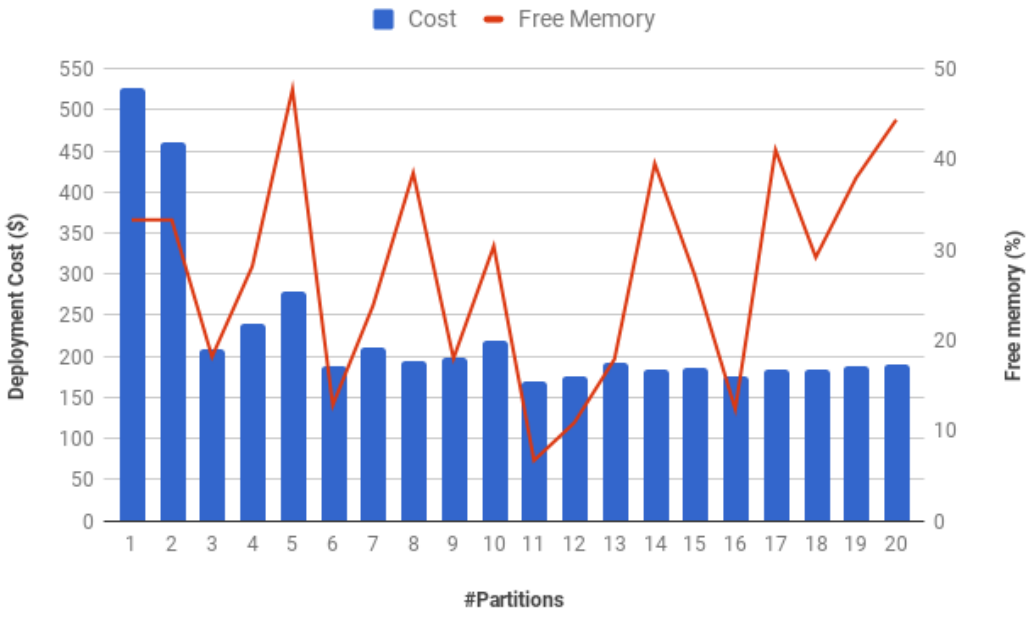
For systems that try to equally distribute the heat of each partition, such as iGraph [9], the number of partitions, and consequently the heat of each one of them, has a huge impact in the possible cost savings achieved. As described on Section 4.3, if the negative impact in performance of larger partitioning factors can be ignored, discussed on Section 5.4, we should split the dataset into partitions assignable to all instance types available, typically small. In this section we try to define the process of choosing the right number of partitions for a given dataset.

Figure 5.2(a) and Figure 5.2(b) show the deployment costs achieved by Hourglass under different partitioning factors for a dataset of size 60Gb and 40Gb, respectively. The red line in each figure shows the percentage of free memory for the entire deployment period and is given by equation 5.2, where T is the set containing all time periods where new assignments were calculated and TOTAL-MEM(t) and FREE-MEM(t) represent, respectively, the sum of all machines' capacity and the sum of all unused machines' capacity for time period t.

$$\frac{\sum_{t \in T} \text{TOTAL-MEM}(t)}{\sum_{t \in T} \text{FREE-MEM}(t)} \tag{5.2}$$

As expected, larger partitioning factors create more partitions with less heat, assignable to more instance types, achieving cheaper deployments. However, increasing the number of partitions does not necessarily mean cheaper deployments, as it increases the number of partitions. The best results are achieved when two conditions are satisfied: (i) the partitioning factor is large enough to create partitions with a heat factor assignable to any type of instance; and (ii) the heat of each partition allows an efficient memory usage for the available machines' capacity. The first condition is easy to met, as we only need to ensure the heat of each partition to be smaller than the heat of the smallest instance, 3.75Gb for our experimental setup. For the second condition we have to find a partitioning factor that creates partitions

(a) Partitioning factor impact on achieved cost savings for dataset with 60Gb Heat.



(b) Partitioning factor impact on achieved cost savings for dataset with 40Gb Heat.

Figure 5.2: Partitioning factor impact on cost.

with a heat factor that allows an efficient memory usage.

Usually, cloud providers offer instance types with resources proportional to their price. For example, the set of machines that we consider have a double the price double the resources relation. For these situations, in order to ensure a 100% machine capacity usage, we only have to use a heat factor per partition that is a multiple or submultiple of the smallest machine memory available. For the dataset with 60Gb, this is achieved using 16 partitions (3.75Gb partitions). For the dataset with 40Gb it is impossible to split it into a number of partitions that have exactly 3.75Gb heat. In this case, one should choose the number of partitions that creates a heat closer to this value, 11 partitions for the 40Gb case. This creates partitions that allow a efficient capacity usage, usually translating into cheaper deployments.

As will be discussed in the next section, increasing the number of partitions can lead to execution overhead. So, in systems that try to equally distribute the heat, one should first chose the maximum number of partitions allowed, either due to execution time constraints or other problem level constraint, and follow the next steps in order to find the ideal number of partitions:

1. Calculate the minimum heat achieved by dividing the graph into the maximum allowed number of partitions.
2. Find the smallest instance able to withstand such heat factor.
3. From 1 to the maximum number of partitions, calculate the heat factor obtained by dividing the dataset into the target number of partitions.
4. Remove partitioning factors that create partitions with a heat higher than the capacity of the machine obtained in step 2.
5. From the remaining partitioning factors, chose the one that creates partitions with the heat value closest to the capacity of the machine selected on step 2.

For example, if we consider the dataset with 40Gb and a maximum number of ten partitions. By following the above steps, one should come to the solution of six partitions, each with about a 6.66Gb heat.

5.4 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 on Section 4.3.2, as the size of partitions decreases, and the potential number of machines performing the computation increases, the execution time is expected to increase, assuming the amount of resources in the system to remain the same.

Figure 5.3: Cost and execution time analysis

Figure 5.3 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 the SSSP application over the synthetic graph, for different numbers of partitions, namely one 60Gb heat partition, two 30Gb heat partitions, four 15Gb partitions, eight 7.5Gb partitions and sixteen 3.75Gb partitions. In this test, and for all others presented in this section, the execution time was measured by 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). The obtained results further support the idea that splitting the dataset into less partitions is better computation wise but worst cost wise.

As discussed on Section 4.3.2, Hourglass tries to achieve a better compromise between cost and execution time by performing a hot/cold separation of vertices, and creating partitions with different heat values. It expects to decrease execution time by reducing the amount of communication performed between machines. This is achieved by co-locating vertices performing most of the communication, hot vertices, on larger partitions and using small partitions for vertices communicating less, cold vertices.

In this section, the tests using the hot/cold separation strategy create partitions of hot vertices, that in total represent around 50% of the total dataset heat with the size of the C4.2XL instance type. The partition size for these vertices was selected as the size of the C4.2XL machines as they have the larger memories that have a relatively stable spot market price as compared to larger instances. The remaining

Figure 5.4: Execution time and messages exchange analysis for the orkut dataset.

Figure 5.5: Execution time and messages exchange analysis for the synthetic dataset.

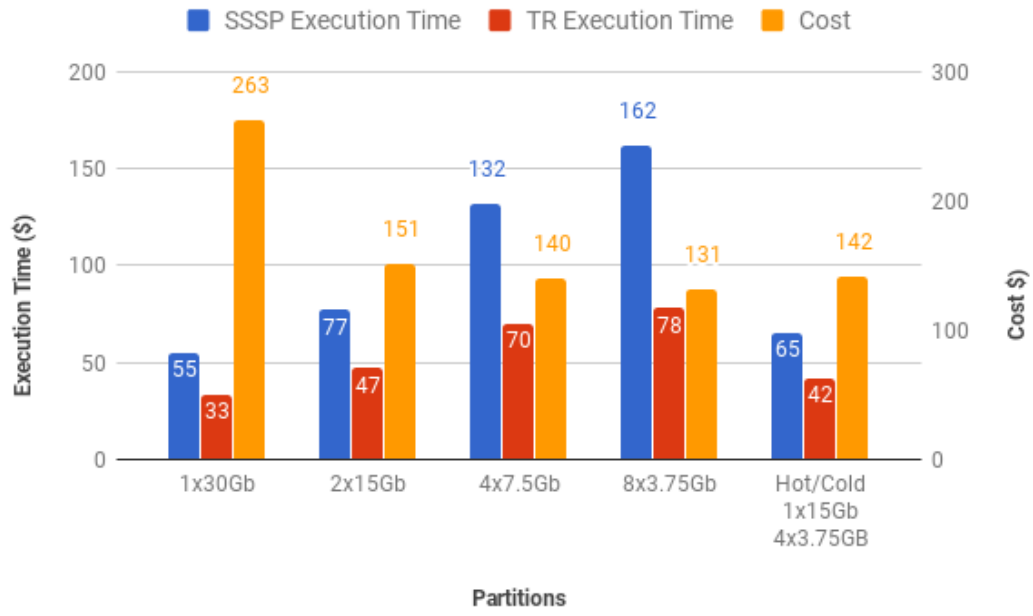
vertices (cold vertices) are divided into partitions with the smallest instance type capacity.

Figure 5.4 and Figure 5.5 show the impact of the Hourglass hot/cold heat distribution strategy over execution time and the amount of messages exchanged between machines for the SSSP application in the Orkut and the synthetic dataset, respectively. The figures show in the x-axis the partitions used. The hot/cold strategy uses one 15Gb heat partition and four 3.75Gb heat partitions for the orkut dataset and two 15Gb partitions and eight 3.75Gb partitions for the synthetic dataset. The y-axis shows the execution time in seconds (left axis) and the number of messages exchanged between machines (right axis), in millions of messages. Both figures show how the Hourglass strategy is able to significantly reduce the amount of messages exchanged between machines for both datasets. As vertices communicating the most are in the same partitions, the number of messages exchanged between machines is significantly reduced. Figure 5.4 shows a better reduction in the number of messages exchanged for the orkut dataset as it uses only one partition to hold such hot vertices. For the syntetic dataset, two machines are used for that purpose, increasing the exchanged number of messages.

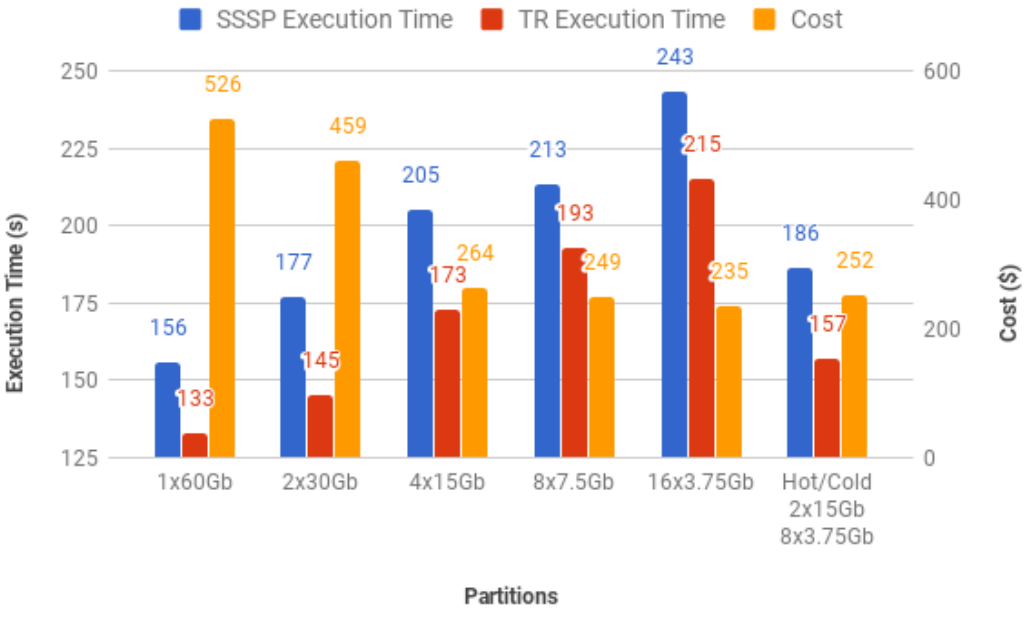
Finally, Figure 5.6(a) and Figure 5.6(b) show the execution time and cost compromise achieved by each heat distribution strategy for the execution of the SSSP and TunkRank applications in the Orkut and synthetic datasets, respectively. The different strategies analyzed are the same as the ones used for the tests presented on Figure 5.4 and Figure 5.5. The y-axis shows the execution time in seconds (left axis) and the cost of deploying such partitions (right axis). The results show that the hot/cold separation strategy presents a the better compromise between cost and execution time than just picking an intermediate partition size. In the Orkut dataset, hot/cold separation offers a performance improvement around 40% and 50%, for the same price, over only using 7.5Gb partitions and 5% more cost savings and up to 15% performance improvement over only using 15Gb partitions. In the synthetic dataset, it offers 5% more cost savings and 10% performance improvement over the intermediate solution.

5.5 Update Staleness Analysis

In the previous section we evaluated the impact of different heat distribution strategies in cost and batch execution time, for the worst deployment case in the latter. Although these results show, in fact, that 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. As explained before, if we partition the dataset into small partitions we can still use large machines if the right market conditions are met. Second, the performance of the system is also impacted by other factors as, for example, periodic reconfigurations, spot evictions and state checkpointing. Considering this, we find the batch execution time to be a good performance metric but it lacks the temporal continuity necessary to cover the above mentioned aspects. To solve this problem we decided to use



(a) Execution time and cost analysis for the Orkut dataset.



(b) Execution time and cost analysis for the synthetic dataset.

Figure 5.6: Execution time and cost analysis.

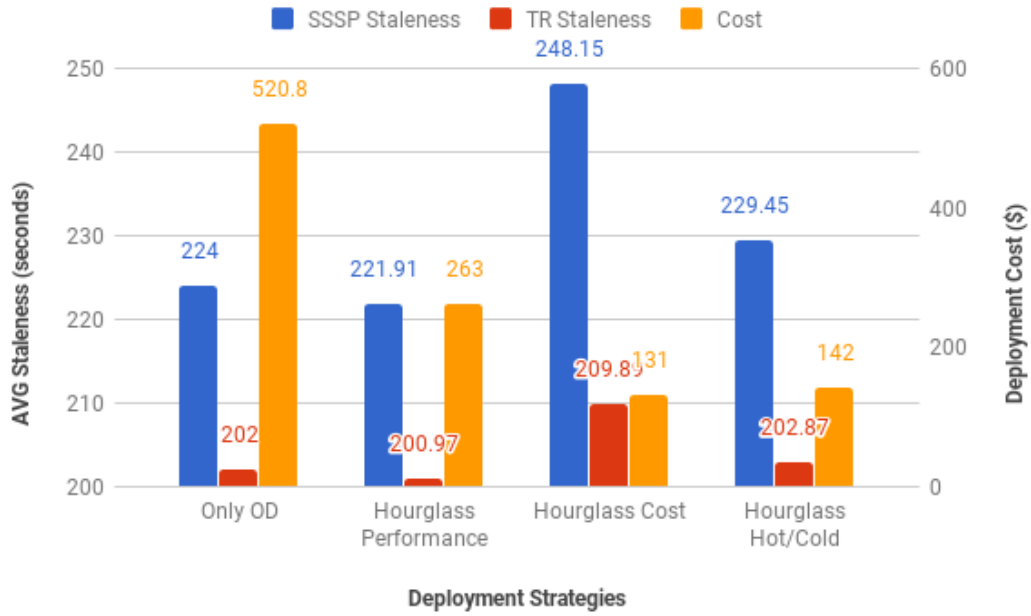
another performance metric, the average update staleness over a time period. This metric measures the time passed since an update was received by the system until its effects become reflected in the persistent state of the graph (reliably stored), averaged for the amount of updates received during a 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 (Section 5.1.1). The simulator allows us to know, at each moment in time, the machines being used and the partitions assigned to each 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 as the 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.

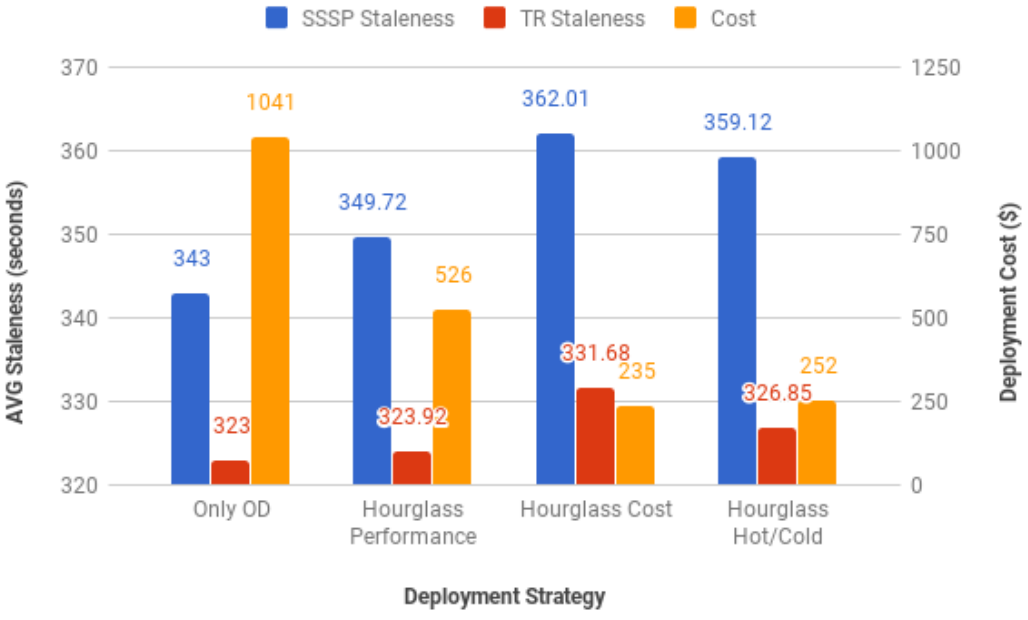
All this information together allowed us to do a simulation process to measure the update staleness for the received updates. Based on the current configuration we know the time a batch takes to be processed and the time it takes to checkpoint the current set of partitions in each machine. If failures occur, the fault recovery protocol, described on Section 4.7.1, is followed based on the system state when the failure occurred.

We measured the average update staleness expected for the Hourglass incremental graph processing system prototype in the two datasets analyzed so far, for the SSSP and TunkRank applications with an update rate of 30k updates per second, for the same time period used for the cost simulation process. The experiment compares the cost and staleness obtained under the three modes of operation of Hourglass, namely the improve performance, reduce cost and hot/cold separation mode, against the baseline approach that only uses on-demand resources. For the simulation process, we set the baseline approach without any type of failure and with the best deployment configuration for the performance.

Figures 5.7(a) and Figure 5.7(b) show the obtained results. We can see that the different modes of operation met the initial expectations and follow the same trend as the batch execution time analysis. More precisely, we can see that when executing for performance, the system is able to achieve results that are no worse than 2% and some times are even better than the baseline approach performance, reaching discounts of 50% over the same baseline cost. The performance improvement over the baseline approach is achieved for the Orkut dataset. This is mainly due to the fact that the baseline approach uses the machine that matches the 30Gb heat of that particular dataset. The Hourglass approach sometimes uses spot instances that have more resources than the minimum required and therefore reduce the



(a) Update staleness and cost analysis for the Orkut dataset.



(b) Update staleness and cost analysis for the synthetic dataset.

Figure 5.7: Update staleness and cost analysis.

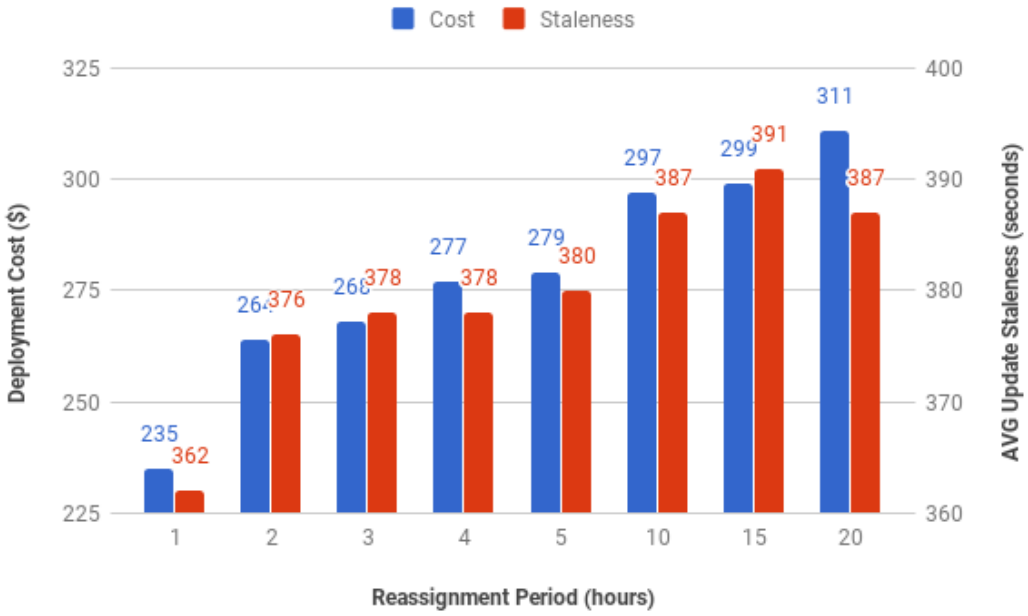


Figure 5.8: Reassignment impact on performance and deployment cost analysis.

batch execution time. The reduce cost mode is able to achieve cost reductions up to 78% over the baseline cost with no more than 10% performance degradation. The hot/cold separation proves to the a good compromise between the two operational modes.

5.6 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 60Gb heat dataset (synthetic) deployment, using the partitioning method that improves cost savings, running the the SSSP application. Similar impacts to the ones presented here on cost and performance were obtained for other datasets, applications and partitioning strategies and are not presented here due to space constraints.

The reassignment frequency impacts several aspects of the system. Namely:

- **Deployment Cost:** Figure 5.8 shows the different deployment costs achieved under different reassignment frequencies. The obtained results show how the deployment cost increases as the reassignment frequency decreases (larger reassignment periods). By increasing the reassignment period we are reducing the system's ability to answer market prices' changes, therefore not using the cheapest possible configurations in every hour.

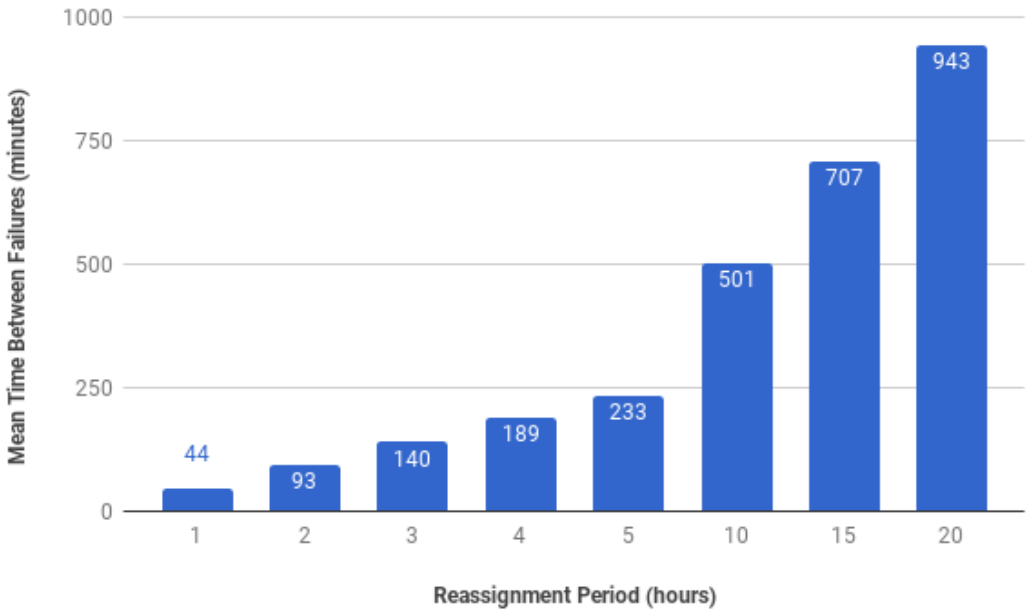


Figure 5.9: Reassignment frequency impact on the mean time between failures.

- **Performance:** Figure 5.8 also shows the system performance degradation as the reassignment period increases. The performance degradation is mainly due to two factors. First, most of the system interruptions happen due to spot failures rather than planned reconfigurations (in the end of the time period). The larger the reassignment period, the more likely is for the system to fail before the end of the current time period. When the system reconfigures on a planned reconfiguration some aspects can be done asynchronously, namely, finding the deployment problem solution based on current market prices and start new machines before the beginning of the new time slot so that when it arrives the new machines are already started and ready to read the persistent state. When failures occur, these things cannot be done upfront and the system needs to wait for the replacement machines to be started, therefore having a greater impact on the system's performance. Also, 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.

- **Mean Time Between Reassignments:** 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 impact on performance if this period is smaller than the time the system takes to process at least one update batch and check-

point 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 5.9 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. This value is important to help choose the best reassignment period. 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.

5.7 Historical Bid Strategy Under Different Probabilities of Failure

As explained in Section 4.5, 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. An approach based on past historical data is then used to find the bid value that matches the target probability of failure. Figure 5.10 shows the effectiveness of the historical approach on different reassignment period durations. It compares the percentage of effective spot failures, observed during the simulation, against the expected probabilities of failure, used in the the historical bid strategy. For this simulation, a dataset of 60Gb heat value is used. The historical data digester is given a two months worth of historical data, used as explained in Section 4.5.

The target probability of failure will impact the eviction rate of spot-instances. This will have an indirect impact on many other aspects. In this Section we analyze, for a dataset of 60Gb heat running the SSSP application with partitions that reduce the deployment cost, the impact of using different probabilities of failure for the historical bidding strategy. We analyze the same aspects that were discussed in the previous section, namely, the deployment cost, performance and MTBF. The observed results are representative of the impact of this failure probability over the mentioned aspects for different datasets, applications, reassignment periods and partitioning methods, that are not here presented again due to space constraints.

Figure 5.11 and Figure 5.12 show, respectively, the impact of different probabilities of failure (10%, 25%, 50%, 75% and 100%) in the deployment cost, system performance and MTBF. 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 that degrade the performance due to the increased number of faults. However, these

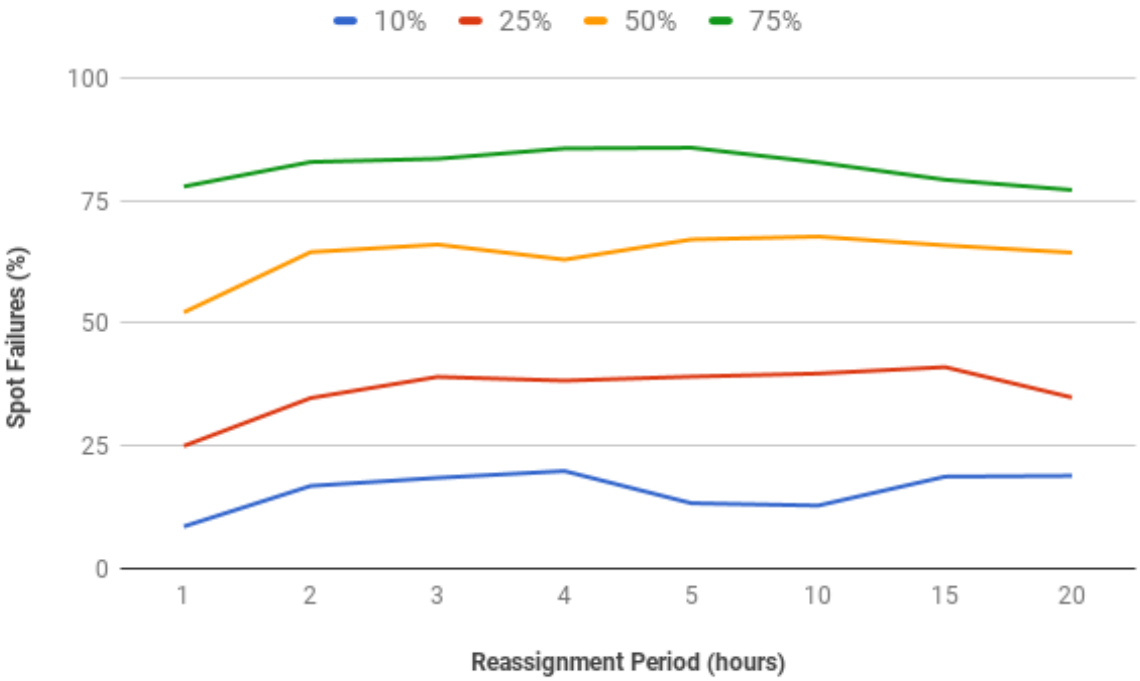


Figure 5.10: Spot failure analysis.

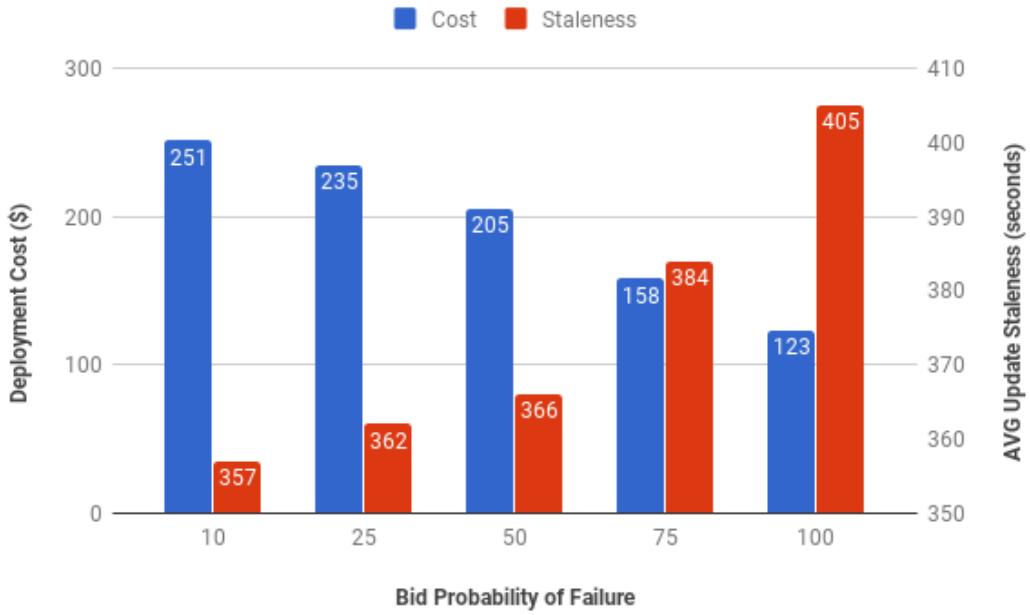


Figure 5.11: Probability of failure impact on deployment cost and performance.

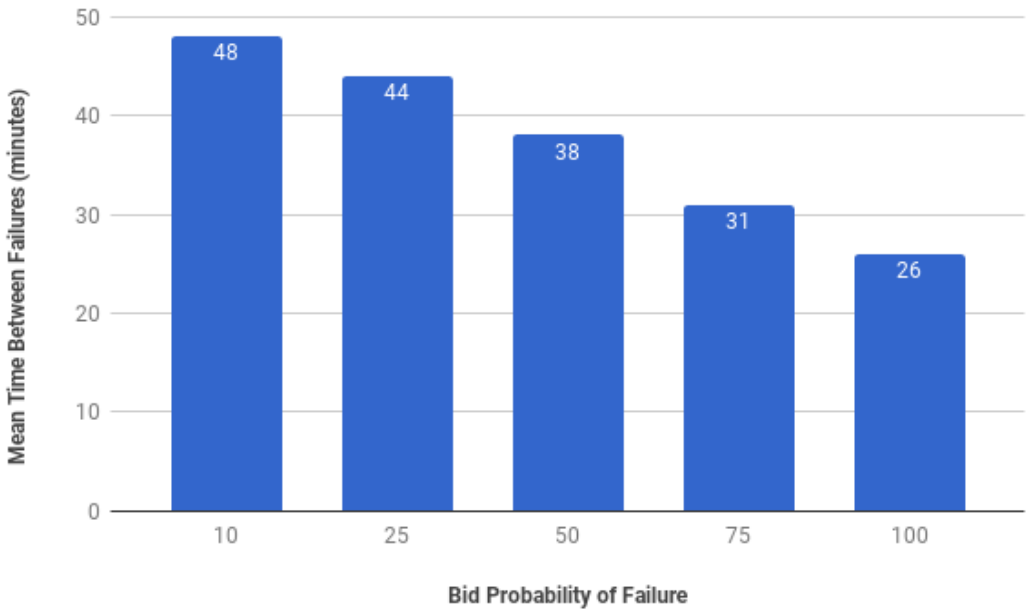


Figure 5.12: Probability of failure impact on mean time between failures.

faults reduce the deployment cost due to the free computing time 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 (comparing against the values presented on Section 5.6).

5.8 Summary

In this section we presented a complete evaluation of the Hourglass system. We started by explaining the experimental setup, describing the datasets, applications and some simulation processes. The first experiment presented shows the importance of considering multiple resource types when using transient resources. The Hourglass approach presents the best results over any other of the presented strategies as it considers more instance's types. The more instance's types the deployment process considers the more alternatives the system has to replace machines that become very expensive.

The next experiment analysis the impact of the partitioning factor over the deployment cost for homogeneous heat distribution strategies. The experiment concludes that the deployment cost is minimized for a partitioning factor that allows partitions to be assigned to all instance's types and that maximizes the target machines' capacity usage.

In the following experiment we analyze the batch execution time and cost under different partitions' sizes. The experiment proves that larger partitions improve execution time but reduce the obtained cost savings and that smaller partitions do the opposite trade-off. The experiment also shows how the hot/cold separation approach, that takes advantage of the environment heterogeneity, is able to reduce the overall communication, presenting a better compromise between cost and batch execution time than just picking an intermediate homogeneous partition size.

Next we studied the update staleness, a performance metric used to analyze the system on a complete deployment setting. The results show that the system is able to achieve 50% cost reductions and less than 2% performance degradation over the baseline on-demand when executing for performance. It also shows that the system is able to achieve a 78% discount over the baseline price with a performance degradation bellow the 10% margin when executing to minimize costs.

The next experiment studies the impact of selecting different reassignment periods in the system's performance, cost and MTBF. The results show that less frequent reassignments worsen performance and reduce the cost savings. However, they may be necessary if the system requires a large MTBF period.

Finally, we analyzed the impact of different probabilities of failure on the bidding strategy over the system's performance and cost. The results show that higher probabilities of failure have a negative impact in performance but are able to reduce deployment costs due to the free computing event. Results show that the system is able to increase the cost reductions over the on-demand baseline approach up to 88% with a performance degradation of 25%.

6

Conclusion

Contents

6.1 Conclusions

In this dissertation we surveyed the existing work on graph processing systems. These systems are designed to improve the execution of graph algorithms, typically executed over large datasets. We identified two types of these systems, static graph processing systems, that consider fixed graph structures, and 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 domains of interest are very dynamic and the graphs that model these domains present a fast changing structure that needs to be constantly analyzed, motivating the usage of incremental graph processing systems over static graph systems. We identified that most incremental graph processing systems are oblivious of the underlying deployment environment, missing opportunities to significantly reduce operational costs, as other systems in different matters have done.

In order to fill this gap between the existing systems and the deployment environment we proposed Hourglass, a deployment engine for incremental graph processing systems. The system leverages heterogeneous transient resource usage in order to significantly reduce the deployment costs associated to the deployment of these systems. The system is constantly analyzing market price variations for the target transient resources and, at user defined reassignment periods, selects the best deployment that is expected to be the cheapest one for the next time period. Hourglass analyzes the target dataset heat and allows its partitioning using different heat distribution strategies that seek to meet system level goals such as: improving execution time; maximizing cost reductions; or find a good compromise between the two using a new hot/cold separation strategy. In order to to maximize the cost savings achieved, we propose one method to select the right number of partitions in which to divide the target dataset for a homogeneous heat distributions strategy.

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 88% cost reductions with a performance degradation less than 25% over the traditional reserved resources deployments. The evaluation performed on the execution time and deployment cost showed that these are maximized under different, and contrary, partitions' characteristics. These results also show how the hot/cold separation is able to achieve a good compromise between cost and execution time. This strategy improves execution time by creating high heat partitions for vertices that communicate the most, therefore increasing communication locality and improving execution time, and low heat partitions for cold vertices that allow cheap deployments.

6.2 System Limitations and Future Work

In the following paragraphs we explain some system limitations and how we intend to work on them for future work.

- **Other Fault Tolerance Mechanisms:** Currently, Hourglass requires incremental graph processing systems using it to use a checkpoint based mechanism, to allow state transfer between different assignments and recover from faults. Other systems may use other mechanisms, such as replication [10], that can be studied to integrate the Hourglass model, facilitating the system integration with other existing systems. These different mechanisms are expected to offer different trade-offs between cost and performance.
- **Study conciliation between hot/cold separation and intelligent partitioning techniques:** Currently Hourglass uses a random vertex assignment strategy to assign hot vertices to hot partitions. As explained on Section 4.3.2, the hot/cold separation is expected to work well together with partitioning techniques that seek to reduce remote communication between vertices by creating partitions that minimize the vertex replication or edge-cut ratio. Applying such techniques only to partitions with hot vertices is expected to further reduce the amount of messages that are exchanged and reduce the large pre-processing times reported for this techniques [60] by reducing the initial dataset to the hot vertices, that are the ones doing the greater part of the communication and where it really matters to improve local communication. We seek to study these aspects and see the impact on performance.
- **Different spot markets:** The performed evaluation is done on a specific market for a given region. A possible improvement is to consider multiple spot markets, on different availability regions and geographic locations, even possibly from different cloud providers with different conditions. Increasing the number of markets is a way of increasing the number of alternatives when some spot-instances are under high demand and have expensive prices. If a geo-distributed solution has to be considered, a partitioning solution similar to [57] has to be considered.

Bibliography

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[2] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, “The web as a graph: Measurements, models, and methods,” in *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*, ser. COCOON’99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 1–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765751.1765753

[3] S. Zhang, H. Chen, K. Liu, and Z. Sun, “Inferring protein function by domain context similarities in protein-protein interaction networks.” *BMC bioinformatics*, vol. 10, no. 1, p. 395, 2009. [Online]. Available: http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-10-395

[4] M. Gori and A. Pucci, “ItemRank: A random-walk based scoring algorithm for recommender engines,” in *IJCAI International Joint Conference on Artificial Intelligence*, 2007, pp. 2766–2771.

[5] P. Reddy and M. Kitsuregawa, “A graph based approach to extract a neighborhood customer community for collaborative filtering,” *Databases in Networked Information Systems*, pp. 188–200, 2002. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-36233-9{_}15{%}5Cnhttp://link.springer.com/chapter/10.1007/3-540-36233-9{_}15{%}5Cnhttp://link.springer.com/content/pdf/10.1007/3-540-36233-9{_}15.pdf

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[7] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884

[8] K. Zhang, R. Chen, and H. Chen, “Numa-aware graph-structured analytics,” *SIGPLAN Not.*, vol. 50, no. 8, pp. 183–193, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858788.2688507

[9] W. Ju, J. Li, W. Yu, and R. Zhang, “igraph: An incremental data processing system for dynamic graph,” *Front. Comput. Sci.*, vol. 10, no. 3, pp. 462–476, Jun. 2016. [Online]. Available: http://dx.doi.org/10.1007/s11704-016-5485-7

[10] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, “Kineograph: Taking the pulse of a fast-changing and connected world,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12. New York, NY, USA: ACM, 2012, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168846

[11] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, *GraphIn: An Online High Performance Incremental Graph Processing Framework*. Cham: Springer International Publishing, 2016, pp. 319–333. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43659-3_24

[12] M. E. J. Newman, “The structure and function of complex networks,” *Siam Review*, vol. 45, no. 2, pp. 167–256, 2003.

[13] A. Rapoport and W. J. Horvath, “A study of a large sociogram,” *Behavioral Science*, vol. 6, no. 4, pp. 279–291, 1961.

[14] D. J. De Solla Price, “Networks of scientific papers,” *Science*, vol. 149, no. 3683, p. 510, 1965. [Online]. Available: http://garfield.library.upenn.edu/papers/pricenetworks1965.pdf{%}5Cnhttp://www.scopus.com/inward/record.url?eid=2-s2.0-0000432228{&}partnerID=tZOtx3y1

[15] J. Cohen, F. Briand, and C. Newman, *Community food webs: data and theory*, ser. Biomathematics. Springer Verlag, 1990, includes bibliographical references.

[16] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner, “The structure of the nervous system of the nematode Caenorhabditis elegans.” *Philosophical Transactions of the Royal Society of London*, vol. 314, no. 1165, pp. 1–340, 1986. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/22462104

[17] Gartner, “The Competitive Dynamics of the Consumer Web: Five Graphs Deliver a Sustainable Advantage,” 2012. [Online]. Available: https://www.gartner.com/doc/2081316/competitive-dynamics-consumer-web-graphs

[18] F. Inc, “Facebook reports second quarter 2016 results,” 2016.

[19] Common crawl. http://commoncrawl.org/. Last Accessed: November 2016.

[20] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.

[21] Neo4j. https://neo4j.com/. Last Accessed: November 2016.

[22] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer, "Weaver: A High-Performance, Transactional Graph Store Based on Refinable Timestamps," in *VLDB*, 2015, pp. 852–863. [Online]. Available: http://arxiv.org/abs/1509.08443

[23] Orientdb. http://orientdb.com/orientdb/. Last Accessed: November 2016.

[24] S. Brin and L. Page, "The anatomy of a large scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1/7, pp. 107–17, 1998.

[25] D. F. Gleich, "PageRank beyond the Web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2014. [Online]. Available: http://arxiv.org/abs/1407.5107{%}5Cnhttp://epubs.siam.org/doi/10.1137/140976649

[26] C. Winter, G. Kristiansen, S. Kersting, J. Roy, D. Aust, T. Knösel, P. Rümmele, B. Jahnke, V. Hentrich, F. Rückert, M. Niedergethmann, W. Weichert, M. Bahra, H. J. Schlitt, U. Settmacher, H. Friess, M. Büchler, H.-D. Saeger, M. Schroeder, C. Pilarsky, and R. Grützmann, "Google goes cancer: Improving outcome prediction for cancer patients by network-based ranking of marker genes," *PLOS Computational Biology*, vol. 8, no. 5, pp. 1–16, 05 2012. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pcbi.1002511

[27] J. Weng, E. P. Lim, J. Jiang, and Q. He, "Twitterrank: Finding topic-sensitive influential twitterers," *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010)*, pp. 261–270, 2010.

[28] Z. Nie, Y. Zhang, J.-R. Wen, and W.-Y. Ma, "Object-level ranking (PopRank)," *Proceedings of the 14th international conference on World Wide Web - WWW '05*, p. 567, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1060745.1060828

[29] D. Walker, H. Xie, K.-K. Yan, and S. Maslov, "Ranking scientific publications using a model of network traffic," pp. P06 010–P06 010, 2007.

[30] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: ACM, 2009, pp. 867–876. [Online]. Available: http://doi.acm.org/10.1145/1645953.1646063

[31] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[32] Apache hadoop. http://hadoop.apache.org/. Last Accessed: November 2016.

[33] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.14

[34] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science and Engg.*, vol. 11, no. 4, pp. 29–41, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2009.120

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010. [Online]. Available: http://arxiv.org/abs/1006.4990

[36] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," *PPoPP*, pp. 135–146, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442530

[37] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[38] Apache giraph. http://giraph.apache.org/. Last Accessed: November 2016.

[39] S. Salihoglu and J. Widom, "GPS : A Graph Processing System," *SSDBM Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pp. 1–31, 2013.

[40] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 950–961, May 2015. [Online]. Available: http://dx.doi.org/10.14778/2777598.2777604

[41] J. Gonzalez, Y. Low, and H. Gu, "Powergraph: Distributed graph-parallel computation on natural graphs," *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 17–30, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf

[42] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[43] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963491

[44] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, “Sync or async: Time to fuse for distributed graph-parallel computation,” *SIGPLAN Not.*, vol. 50, no. 8, pp. 194–204, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858788.2688508

[45] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: A framework for machine learning and data mining in the cloud,” *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354

[46] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, “Asynchronous large-scale graph processing made easy,” in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013. [Online]. Available: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper58.pdf

[47] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Priter: A distributed framework for prioritized iterative computations,” in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC ’11. New York, NY, USA: ACM, 2011, pp. 13:1–13:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038929

[48] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” *SOSP ’13*, pp. 456–471, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2517349.2522739

[49] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, “Time-evolving graph processing at scale,” in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES ’16. New York, NY, USA: ACM, 2016, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2960414.2960419

[50] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. October, pp. 509–512, 1999.

[51] D. Easley and J. Kleinberg, “Networks , Crowds , and Markets : Reasoning about a Highly Connected World,” *Science*, vol. 81, p. 744, 2010. [Online]. Available: http://books.google.com/books?hl=en{&}lr={&}id=atfCl2agdi8C{&}oi=fnd{&}pg=PR11{&}dq=Networks+,+Crowds+,+and+Markets+:+Reasoning+about+a+Highly+Connected+World{&}ots=LxUX{_}qqGxp{&}sig=AIUBAVQSiqW6jxnRaIlbW2S2498

[52] L. Akoglu, H. Tong, and D. Koutra, “Graph based anomaly detection and description: A survey,” *Data Min. Knowl. Discov.*, vol. 29, no. 3, pp. 626–688, May 2015. [Online]. Available: http://dx.doi.org/10.1007/s10618-014-0365-y

[53] M. Pundir, M. Kumar, L. Leslie, I. Gupta, and R. Campbell, *Supporting on-demand elasticity in distributed graph processing*. United States: Institute of Electrical and Electronics Engineers Inc., 6 2016, pp. 12–21.

[54] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun, "Pado: A data processing engine for harnessing transient resources in datacenters," in *Proceedings of the EuroSys*, 2017.

[55] 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 *Proceedings of the EuroSys*, 2017.

[56] J. Huang and D. J. Abadi, "Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 540–551, Mar. 2016. [Online]. Available: http://dx.doi.org/10.14778/2904483.2904486

[57] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, "Graph: Heterogeneity-aware graph computation with adaptive partitioning," in *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 118–128.

[58] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 16:1–16:10. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079097

[59] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," *CoRR*, vol. abs/1610.07220, 2016. [Online]. Available: http://arxiv.org/abs/1610.07220

[60] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 631–643. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/malicevic

[61] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012. [Online]. Available: http://arxiv.org/abs/1205.6233

[62] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008. [Online]. Available: http://arxiv.org/abs/0810.1355

[63] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "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, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 620–634. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064220

[64] B. Javadi, R. K. Thulasiramy, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, ser. UCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 219–228. [Online]. Available: http://dx.doi.org/10.1109/UCC.2011.37

[65] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[66] D. Tunkelang, "A twitter analog to pagerank." Retrieved from http://thenoisychannel.com/2009/01/13/a-twitter-analog-topagerank, 2009.

[67] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004, pp. 442–446. [Online]. Available: http://dblp.uni-trier.de/db/conf/sdm/sdm2004.html#ChakrabartiZF04