Hourglass - Incremental Graph Processing on Heterogeneous Infrastructures

Pedro Joaquim
pedro.joaquim@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Graphs are data structures that can represent relations among entities and enable the extraction of insightful information from their properties. Graph processing systems, that aim at simplifying the computational analysis of graph structures, typically rely on a vertex-centric approach to ease the development of parallel algorithms. In this approach, the computation is modeled as a function that needs to be repeatedly applied to each vertex of the graph, in an iterative process. The irregular structure of graphs and the dynamic nature of the domains that they model, induce load imbalances both in the work performed at each vertex and the number of updates that each region of the graph receives. Knowledge about this load imbalance could be used to provide a more efficient allocation of resources during the execution of the graph processing task. Many graph processing systems are frequently deployed in Infrastructure as a Service (IaaS) platforms but, unfortunately, are oblivious to the heterogeneity of resources that exist in such environments. This prevents them from performing a suitable match among the resources available and the processing needs, which could bring significant cost savings by avoiding the allocation of expensive resources to graph regions that have low computational demand. In this report we advocate the development of a new graph processing system that is aware of the heterogeneity of underlying resources, to enable it to better adapt to different usage patterns and reduce deployment costs.

Table of Contents

1 Introduction . . . 3
2 Goals . . . 4
3 Graph Management Systems . . . 5
3.1 Graph Databases . . . 6
3.2 Graph Processing Systems . . . 6
3.3 Comparison . . . 7
4 Example Graph Processing Applications . . . 7
4.1 PageRank . . . 8
4.2 Community Detection . . . 8
5 Architectures of Graph Processing Systems . . . 9
5.1 Distributed Graph Processing Systems . . . 9
5.1.1 MapReduce . . . 9
5.1.2 Pregel . . . 10
5.1.3 Pregel Variants . . . 11
5.2 Single Machine Graph Processing Systems . . . 14
5.2.1 Polymer . . . 14
5.2.2 GraphChi . . . 15
6 Incremental Graph Processing . . . 16
6.1 Kineograph . . . 17
6.2 GraphIn . . . 18
6.3 Heterogeneous Update Patterns . . . 19
7 Deployment Issues . . . 20
8 Architecture . . . 21
8.1 Overall Strategy . . . 23
8.2 Master Node . . . 23
8.3 Slave Node . . . 24
8.4 Fault Tolerance . . . 25
8.5 Final Considerations . . . 26
9 Evaluation . . . 26
9.1 Deployment Costs and Performance Analyses . . . 26
9.2 Fault Tolerance Mechanism Analysis . . . 27
10 Scheduling of Future Work . . . 27
11 Conclusions . . . 28

1 Introduction

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.

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 Bulk Synchronous Parallel (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 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.

To support the dynamic aspect of the underlying graphs, that suffer modifications as the time goes by, incremental graph processing models have been proposed. They aim at performing the computations required to reflect the modifications performed to the graph, such as modifying the static properties of vertices/edges or add/remove new vertices or edges, avoiding to recompute all data from scratch on the new graph version. These systems typically analyze a batch of updates identifying the graph regions that are affected by such updates. Then, computation is performed to update the data associated with the inconsistent graph zones, rather than to the entire graph. The updates to the graph often exhibit access patterns where is possible to notice that the distribution of vertices that are modified is not uniform across graph partitions.

Most graph processing systems are deployed on commodity clusters on IaaS platforms. However, these systems are oblivious to the heterogeneity of resources in the deployment environment, ignoring opportunities to reduce the costs by

matching the resources with the computational needs. In this work we advocate the development of a deployment aware incremental graph processing system able to explore opportunities to reduce the number of necessary machines and reduce system costs.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. From Section 3 to Section 7 we present all the background related with our work. Section 3 discusses the properties and objectives of different systems designed to manage graph structures. On Section 4 we present some examples of algorithms that typically run on these systems. Next, Section 5 presents examples of some graph processing systems detailed in the literature. Section 6 introduces the incremental graph processing model and gives examples of systems that follow this model. Section 7 discusses on deployment environments. Section 8 describes the proposed architecture to be implemented and Section 9 describes how we plan to evaluate our results. Finally, Section 10 presents the schedule of future work and Section 11 concludes the report.

2 Goals

Most of the current graph processing systems are oblivious to the heterogeneity of resources that exists in their deployment environment, thus losing opportunities to reduce the associated costs. This work makes a survey of existing graph processing systems, identifying their main characteristics, advantages, and disadvantages, with the aim of finding opportunities to reduce the deployment costs by leveraging knowledge about the heterogeneity of resources. More precisely:

> *Goals:* The main goal of our work is to design and implement a new graph processing system aware of the heterogeneity of resources that may exist in the deployment environment. The system should be able to take advantage of imbalances in computational demand on different graph zones, to better match the resources with the computational needs, thus reducing the costs associated with deployment.

To achieve our goal, is important to identify imbalances in the computational demand, understand the implications of having machines with different resources, or even some machines turned off, while computations are being performed and how to handle this heterogeneity.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a graph processing system that can automatically match machines with heterogeneous resources to the needs of the computations being performed on different regions of the graph; ii) an implementation of the proposed system and iii) an extensive experimental evaluation to analyze the cost reductions it may provide and its impact on the system's performance.

3 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 [7] 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) [8]. *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 [9]. *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 [10], another example of biological networks are neural networks [11]. Finally, markets can also be modelled by graphs. Gartner [12] 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 report we call such platforms *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 reportes [13] 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 [14] 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. [15] 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.

3.1 Graph Databases

Graph Databases [15] 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 [16–18]. 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 on the graph data model. This is not truth in 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 for the execution time of each query to remain constant, even if the dataset grows in other graph regions.

3.2 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.

Table 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 analysing graph structure
Queries Type	Standardized and simple queries that include small graph zones	Complex queries involing 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	One computation at a time

3.3 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 1. In this report, we will focus mainly on graph processing systems. We will detail the most relevant architectures and design decisions in Section 5, 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.

4 Example Graph Processing Applications

As hinted before, graph processing is a data mining task that consists of analyzing graph data from different perspectives and summarizing the results in a form that typically can be 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.

4.1 PageRank

PageRank [19] is an algorithm that 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:

$$PR(A) = (1-d) + d(PR(T_1)/C(T_1) + .. + PR(T_n)/C(T_n))$$

Where $C(T_i)$ 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 [20] discusses on the aplicability of such algorithm on graphs of different domains. One of such diverse application is Biology where a study used a similar algorithm [21], 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 [22] and others [23, 24].

4.2 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 history 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].

5 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 a 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.

5.1 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 parallelised, 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.

5.1.1 MapReduce

The MapReduce framework [25] (and its open-source variant Hadoop MapReduce [26]) is a notable example of a middleware, and of a companion programming paradigm, that aims 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 [27, 28], have attempted to leverage MapReduce for graph analytics. Unfortunately, experience has shown that MapReduce is not well suited for graph applications [29, 30], 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.

5.1.2 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 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 [31]. The computation consists of a sequence of iterations, also called supersteps, in which 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. All 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 and computation is resumed from the most recent checkpointed superstep.

5.1.3 Pregel Variants

Since Pregel has been introduced, several systems following the same "think like a vertex" paradigm have been proposed [32–36]. These systems either provide conceptually different execution models, such as *incremental* execution (discussed on Section 6), or introduce some system level optimizations that we now discuss.

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. Salihoglu and Widom [33] presented an experimental evaluation, on a system similar to Pregel, where they tried to verify if certain algorithms have performance improvements if vertices 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* [37] software, that divides a graph into a given number of partitions, trying to reduce the number of edges crossing partitions (with source vertex and target vertex on different partitions).

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 [33] 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.

Gather, Apply, Scatter (GAS): Gonzalez et al. [36] 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 [36]. The system adopts a shared-memory view of the graph data, although in reality it is still partitioned, and proposes a 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.

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 that every worker finishes the current iteration before a new one can start.

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 [38] 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 [39].

To overcome these problems, asynchronous execution models on graph processing systems have been proposed [40, 34, 41, 42]. 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 [40]. The dynamic scheduling of computation can accelerate convergence as demonstrated by Zhang et al. [43] 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. [39] 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, asynchronous model can converge faster and favors CPU intensive algorithms, where workers spend most of the time inside the *compute* functions. 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 [39], a graph processing system that adapts to execution characteristics and dynamically switches between the asynchronous and synchronous execution model.

5.2 Single Machine Graph Processing Systems

While the access to distributed computational resources is nowadays eased though 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. We now analyze some of the graph processing systems that follow this approach.

5.2.1 Polymer

Several multicore machines adopt a Non-Uniform Memory Access (NUMA) architecture that consists of 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 [44] 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*, where 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 [30] and Galois [45], 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.

5.2.2 GraphChi

GraphChi [41] 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 Parallel Sliding Windows (PSW), a novel method for very large graph processing from disk. PSW processes graphs in three stages:

Loading the Graph: In this method, 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 1.

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

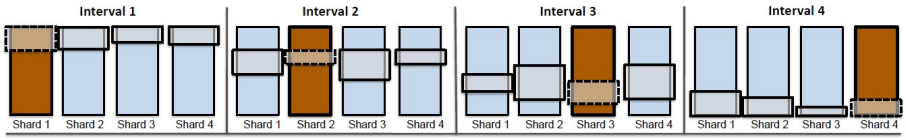

Fig. 1. Example of the PSW method. The vertices are divided into four intervals, each one of them with an associated shard. The computation proceeds by processing each interval at a time, in-edges are read from the current associated shard (in dark color) while out-edges are read from other shards segments (highlighted on top of the other shards), image taken from [41].

parallel. In addition to the vertex values, 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.

6 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 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.

6.1 Kineograph

Kineograph [35] was one of the first systems to take the incremental approach into consideration in the system's design. 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 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 typical 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.

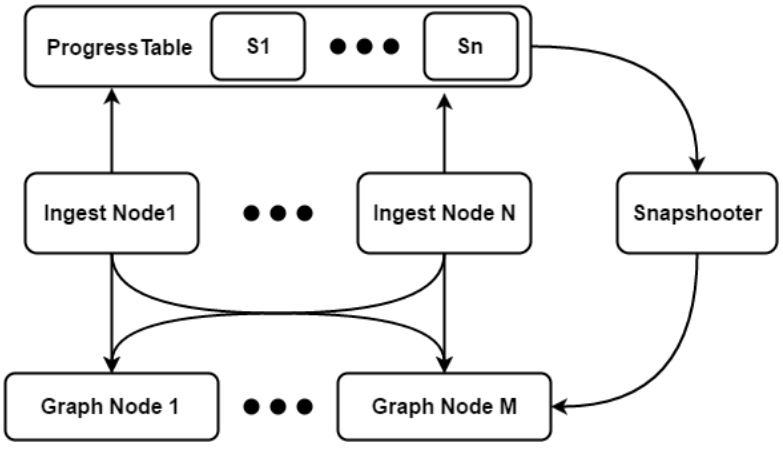

Fig. 2. Kineograph system overview, image adapted from [35].

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.

6.2 GraphIn

GraphIn [46] is an incremental graph processing framework and it 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 [36] 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 a 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. Figure 3 illustrates the process described above.

The system evaluation reports speedups of 407×, 40× and 82× 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.

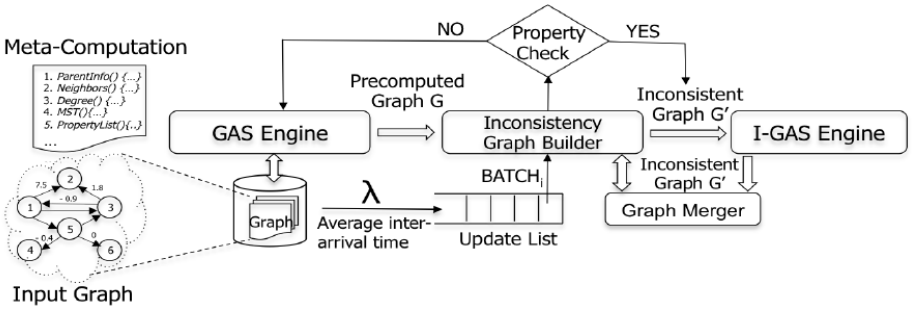

Fig. 3. GraphIn computational process overview, image taken from [46].

6.3 Heterogeneous Update Patterns

Most incremental graph processing systems [35, 46–48], 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 from 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 much more updates than others. For example, on social networks where graphs model the existing relationships between users, 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 [49], 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 with existing nodes. Easley and Kleinberg [50] 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. [47] took this insight into consideration when designing iGraph, an incremental graph processing system. It provides a programming model similar to Kineograph [35] 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.

7 Deployment Issues

The systems studied in this report can be used for a wide-variety of applications with very different time constraints. For example, real-time applications, such as fraud detection [51] or trend topic detection [22], 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 [35, 46] than static recomputation and still can benefit from the update batching on applications without real-time requirements. Thus, these systems have a computational model more 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 use the right number of machines to avoid wasting resources and consequently money. Most of the graph processing systems, including the ones discussed on this report, 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 flexibil-

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

ity, 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.

Single machine architectures on server class machines, with large memories and hundreds of CPU cores, are cheaper to deploy on these platforms than using multiple machines with less resources to achieve the same level of processing capabilities and are usually faster than the distributed counterparts. However, they represent a stronger commitment to the allocated resources making it more difficult for the deployed system to adapt to a greater or lesser need for processing resources. On distributed architectures, increasing or decreasing the number of machines performing the graph computation allows the system to provide better flexibility to dynamic resource demands, which may come from growing datasets or usage spikes. This allows the system to scale to a virtually unlimited level of resources while on single machine architectures it has fixed boundaries.

Dindokar and Simmhan [52] 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 computational 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, but platforms with larger payment granularities, such as Amazon that has payments per hour of machine usage, nullify the cost reductions achieved by turning machines off on a superstep basis.

8 Architecture

In this section we describe the preliminary architecture of *Hourglass*, an incremental graph processing system designed to take advantage of heterogeneous update patterns on dynamic graphs in order to reduce deployment costs on heterogeneous clusters. In these clusters, machines have different costs associated with their usage, typically proportional to the level of resources that they

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

provide. For example, machines with higher processing capabilities are more expensive to rent on IaaS platforms and will probably have higher costs associated with power supply and cooling mechanisms on private clusters.

Hourglass will provide a computational model similar to the ones provided by other incremental graph processing systems [35, 46–48] and, in addition, with the intent of reducing deployment costs, it will analyze the update patterns of each graph region and, by moving vertices with similar update rates to the same set of partitions, create partitions with different computational requirements. After identifying these partitions, the system is able to assign the ones with higher computational requirements (high update rates) to machines with higher computational capabilities. Rarely updated partitions can be assigned to machines with lower processing capacities, that can be turned off most of the time and turned on when necessary, thus reducing costs without affecting performance.

Although this strategy can be used on different types of clusters with heterogeneous resources, we decided to follow the trend of other graph processing systems and develop Hourglass in the AWS environment. Figure 4 illustrates an overview of the system's architecture. It follows a typical graph processing master/slave architecture, being master and slave nodes EC2 instances. In the next sections we describe the implementation strategy that we intend to follow in order to accomplish our goals on the target environment, detailing the different entities that exist in the system and the fault tolerance mechanisms.

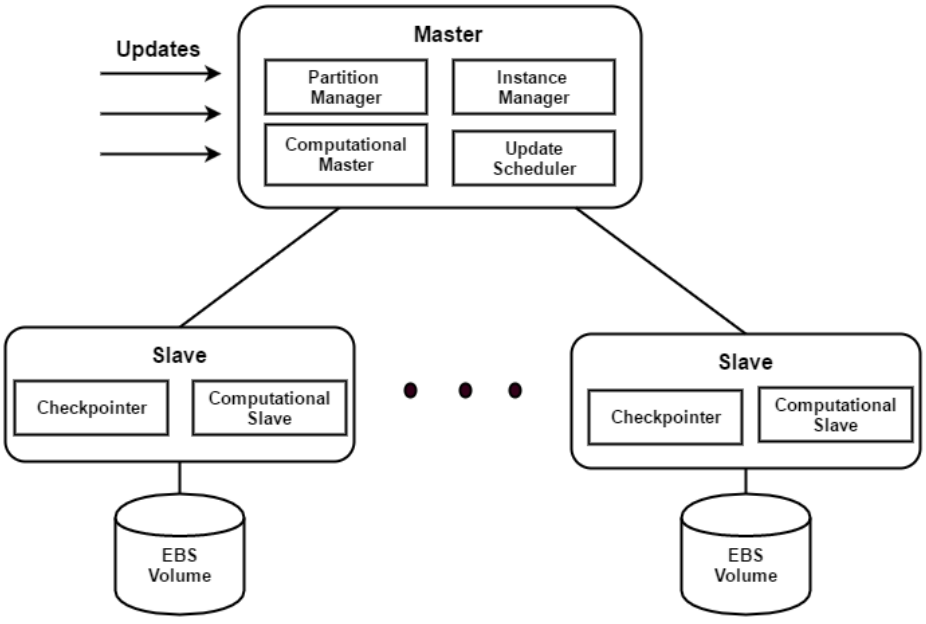

Fig. 4. Hourglass Architecture's Overview

8.1 Overall Strategy

AWS provides different machine renting mechanisms with three different types of instances: *On-Demand Instances*, *Reserved Instances* and *Spot Instances*. On-Demand instances are paid by the hour and the user is free to increase or decrease the number of machines she uses. Reserved instances, as the name suggests, allow the user to reserve instances for a predefined period of time with discounts off On-Demand, around 40% for a period of one year, and the user is charged even if she do not use them. Spot instances allow the user to bid on spare instances. Their price is based on the current usage of On-Demand instances and when the user's bid price is equal or higher than the current calculated price the platform launches the number of machines requested when the bid was placed. When the price for those spare instances is higher than the user's bid those instances are immediately terminated, even if they were performing any computation. These instances allow to pay up to 90% less than for the On-Demand ones, making them good candidates to hold the partitions that are rarely updated. In this case, the machine heterogeneity is also observable in terms of their reliability and rental price. Having cheap Spot instances that are likely to fail and can not be started in a deterministic way and On-Demand instances that are more expensive but have a much lower probability of failure and can be turned on and off as needed.

During the initial phase, while update patterns are being analyzed, we will use the same type of On-Demand instances for all slaves. The system will then reassign vertices in order to allocate those with higher update rates to the same set of partitions, leaving other partitions with vertices that are rarely updated. This will allow the system to keep partitions with high update rates in the On-Demand instances and move the remaining partitions to Spot instances that can be turned off most of the time. Based on the size and computational requirements of each of the obtained partitions, On-Demand and Spot instances can be of different types, providing different levels of resources and enabling the system to better adapt to the computational needs of each partition. After some time of usage, it is easier for the system to predict the number of machines necessary for those partitions with high update rates, enabling the usage of Reserved instances to hold them, further reducing the deployment costs.

In this report, vertices that belong to partitions assigned to instances that are not active are called *offline vertices*. Creating those types of vertices also creates some challenges from the graph computational point of view, given that it was never studied before. For example, what happens if offline vertices receive updates or are scheduled for execution. In the next sections we describe how we deal with such situations.

8.2 Master Node

The *master* node is the entity responsible for the system management and is the entry point to the update requests. It has several modules implementing different responsibilities, namely:

Partition Manager: The master node implements in this module an entity similar to the *hotspot rebalancer* on iGraph [47]. In the original system it is responsible for analyzing update patterns and reassign vertices in order to obtain partitions that receive approximately the same number of updates. In our system, as described before, the goal is different and it will assign vertices with high update rates to the same partitions. Some vertices may never receive updates and still be constantly scheduled for execution due to received messages from other vertices. To prevent them from being moved to partitions with vertices that are never executed we also consider how often vertices are scheduled for execution.

Update Scheduler: The master node is responsible for receiving the update requests and forward them to the slave that holds the partition that will receive the update. Following the typical approach of other incremental graph processing systems [35, 46, 47], our system will receive the requests and group them at batches of updates that are delivered to the slaves at fixed time intervals to initiate the incremental computation.

Instance Manager: This module manages the instances in the system. It is able to check if it is necessary to create requests to launch Spot instances based on the updates that came to offline vertices. It is also responsible for checking if any of the On-Demand instances failed and replace it if necessary.

Computational Master: The master node is still responsible for coordinating the supersteps of the incremental execution. This module is responsible for coordinating the slaves and check stop conditions.

8.3 Slave Node

The slave nodes are responsible for performing the incremental computation on the in-memory graph partitions that they hold and checkpointing the obtained results to persistent storage. Although EC2 instances have secondary storage, it is not reliable. On machine failures or shutdown, the instance loses all the data including the data stored in the secondary storage. Thus, it is necessary to use another mechanism to store the data that needs to be persistently stored. The use of Spot instances, that are likely to fail, further motivates the need for a persistent way of storing information. We intend to use the Elastic Block Store (EBS) Amazon service, this service enables us to attach to each instance a persistent block storage. Even if the attached instance is terminated or fails the volume persists and can be later on assigned to a new instance. This storage service provides low latency and is automatically replicated to prevent data loss from component failures.

Regarding the computational execution, given that some graph partitions are assigned to Spot instances it is possible, but unlikely given the characteristics of the vertices assigned to those instances, that in the incremental computation some messages are directed for vertices that are not at any active instance,

therefore scheduling offline vertices to execution. When this happens, the slaves sending the messages proceed the computation normally and hold the messages for a later phase when the destination vertices are not offline. This enables the computation to continue even in the presence of offline vertices that receive messages. When this happens, it is possible that some graph regions become inconsistent and their computational values do not reflect the latest received updates. This will eventually converge when the scheduled offline vertices (that have pending updates or messages from other vertices) become active and are executed.

8.4 Fault Tolerance

The discussed static graph processing systems use checkpointing mechanisms to handle machine failures. The graph state is checkpointed at the end of certain supersteps, when the system is at a steady state. In case of slave failure, the system rolls back to the last checkpointed superstep and continues computation from there. Incremental graph processing systems have an additional aspect to take into consideration on fault tolerance mechanisms. These systems receive update requests that cannot be lost until their effects are reflected in the persistent state.

A naive solution would be to also checkpoint every update request that comes into the system. Although effective this solution is far from efficient, a high number of update requests would degrade the system performance due to frequent checkpoints. Other systems, like Kineograph [35], use a quorum-based replication mechanism. Every vertex in the system is replicated to k different partitions and the system is able to tolerate up to f failures, where $k \geq 2f + 1$ holds. An update is then replicated to every replica and considered reliably stored as long as at least $f + 1$ have confirmed that the update was stored. This approach removes the need for checkpointing at all, given that partitions are replicated, but requires more in-memory space (about k times the original space) to hold every graph vertex and its associated replicas. When a failure occurs, the replicas of the vertices on the failed partitions take their places and the system is able to continue execution. On large graphs this replication mechanism could require a larger number of slave machines, thus defeating the main system goal of reducing deployment costs.

To avoid the replication drawbacks, we intend to use a log-based checkpointing mechanism. In this approach, slave nodes still perform the checkpoints at specific time periods and updates are sent to k machines, being one of them the machine holding the vertex to which the update is intended. When a new checkpoint occurs, the updates whose effects are reflected on the current checkpoint can be discarded. When a slave machine failure occurs, the master node is able to detect the failure and start a new machine to take its place. The new machine loads the last checkpoint and requests other slaves for the updates intended for its assigned partitions that were not yet processed on the last checkpoint state. This approach allows the system to handle frequent Spot instances terminations as any other machine failure. Updates directed to offline vertices are sent to the

other $k-1$ machines, when the Spot instances come back up they follow the procedure described above.

Considering that Spot instances are very likely to fail, they are not considered to hold the replicated update logs and should have much shorter checkpoiting periods than the On-Demand instances. Elnozahy et al. [53] do a very detailed analysis of log-based checkpointing fault tolerance mechanisms on message passing systems.

8.5 Final Considerations

Due to the lack of publicly available incremental graph processing systems our implementation will have to be made from scratch. Therefore, and due to the short period of time to implement the system, we intend to leave the implementation of the initial execution phase of the system, where the update patterns are analyzed and vertices assigned to different partitions based on their update patterns, to future work. This part of the system could be created based on the *hotspot detector* from iGraph [47]. This entity is responsible for analyzing the number of updates that each vertex receives and, at user defined intervals, rebalance partitions in order to balance the work. In our system, as explained before, we would also check the number of times a vertex was executed in that period so that vertices that do not receive high number of updates but are frequently scheduled for execution do not get moved to partitions that are likely to go offline.

We also intend to implement a simplified version of the master node, using a single machine. In order to cope with master failures a paxos-based solution [54], such as Apache Zookeeper [55], should be used to implement the centralized functionality instead. Using a single machine to receive all the update requests will possibly become a scalability bottleneck in the system, causing performance degradation. If such situation arises, a solution similar to the Kineograph [35] approach should be used. The system uses multiple *ingest nodes* that receive the updates and have different sequence numbers to timestamp different operations.

This way we intend to use our time to implement the aspects of the system that are new. Namely, the usage of heterogeneous resources in different partitions and the aspects that need to be adapted in order to cope with such modification (fault tolerance mechanism, vertex scheduling policies and resource allocation).

9 Evaluation

Regarding the system evaluation we intend to deploy it on the target AWS environment and analyze different dimensions of the problem on different usage conditions.

9.1 Deployment Costs and Performance Analyses

First, being that the main system goal is to reduce the deployment costs, we intend to compare under the same conditions (same number and type of

machines, input graph and set of updates) the costs of deploying our system on the AWS environment against another incremental graph processing system that does not take the cluster heterogeneity into consideration. Due to the lack of publicly available incremental graph processing systems, we intend to use a modified version of Hourglass, that only uses On-Demand instances, as the baseline comparison system. Considering that the potential cost reductions achieved by Hourglass are highly dependent of the generated update patterns, we intend to use different update sets. We shall use at least three different types of update sets, one with a typical use case where most of the updates are directed to the partitions that belong to On-Demand instances and a very small part of updates are for those on Spot instances. Another update set that is evenly distributed across partitions and a third update set that is highly directed towards the partitions that belong to Spot instances. We shall compare the costs and processing time necessary to complete the three update sets in Hourglass and in the baseline implementation. We expect to achieve cost reductions in the three update sets without performance degradation at least in the typical use case update set.

In order to understand the real performance implications of using Spot instances, the Hourglass execution under the three types of datasets should be evaluated for different bid values on Spot instances. The goal of such variance is to find the sweet spot between the bid value (and consequently the price paid) and the performance degradation for the three datasets. We expect to be able to use the lowest possible bid value on the typical use case without performance degradation and increase bid values when atypical situations (the other two datasets) arise to mitigate the performance problems that may occur.

9.2 Fault Tolerance Mechanism Analysis

To the best of our knowledge, this system is the first incremental graph processing system to use a log-based checkpointing mechanism, therefore we intend to compare it against other fault tolerance mechanisms, namely the replication approach followed by Kineograph [35] and the naive solution that performs a checkpoint every time a new update request comes into the system. The evaluation should compare the system performance under the three alternative mechanisms for a fixed update set. We also intend to analyze the necessary time to process a batch of updates directed for a specific machine that will fail during the incremental execution under the three mechanisms. For this experiment it is expected that the replication mechanism outperforms the others because it is not necessary to start new machines in order to proceed. However, this approach requires a larger number of machines that we shall also consider defining the existing trade-off between deployment cost and execution time under machine failure.

10 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

11 Conclusions

Graph processing systems are designed to improve the execution of graph algorithms that leverage on the graph structure and perform long operations that traverse the entire graph. In this report we analyzed the architectures of state-of-the-art systems and discussed the main advantages and disadvantages of each one of them.

Incremental graph processing systems, that allow modifications to the underlying graph structure during computation, have been proposed. They aim at keeping the values extracted from computation in pace with the fast changing structure of graphs, originated from the dynamic nature of the domains that they model. These systems have shown to the best solution for long term deployments.

Graphs have different computational requirements that could be leveraged to reduce the system deployment costs by matching computational needs of each graph region with the right resources. However, existing systems are oblivious to the heterogeneity of resources that exist in their deployment environment thus losing such opportunities.

We proposed Hourglass, an incremental graph processing system highly integrated with its deployment environment. It is able to reduce the associated deployment costs by leveraging the existing resource heterogeneity and computational imbalances across graph regions. We have proposed the usage of a log-based checkpointing fault tolerance mechanism to reduce the memory requirements of other approaches [35]. We also presented the evaluation plan and future work schedule.

Acknowledgments We are grateful to Rodrigo Rodrigues and Manuel Bravo for the fruitful discussions and comments during the preparation of this report. 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.

References

1. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. Proc. VLDB Endow. **8**(12) (August 2015) 1804–1815

2. Kleinberg, J.M., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.S.: The web as a graph: Measurements, models, and methods. In: Proceedings of the 5th Annual International Conference on Computing and Combinatorics. COCOON'99, Berlin, Heidelberg, Springer-Verlag (1999) 1–17
3. Zhang, S., Chen, H., Liu, K., Sun, Z.: Inferring protein function by domain context similarities in protein-protein interaction networks. BMC bioinformatics **10**(1) (2009) 395
4. Gori, M., Pucci, A.: ItemRank: A random-walk based scoring algorithm for recommender engines. In: IJCAI International Joint Conference on Artificial Intelligence. (2007) 2766–2771
5. Reddy, P., Kitsuregawa, M.: A graph based approach to extract a neighborhood customer community for collaborative filtering. Databases in Networked Information Systems (2002) 188–200
6. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10, New York, NY, USA, ACM (2010) 135–146
7. Newman, M.E.J.: The structure and function of complex networks. Siam Review **45**(2) (2003) 167–256
8. Rapoport, A., Horvath, W.J.: A study of a large sociogram. Behavioral Science **6**(4) (1961) 279–291
9. De Solla Price, D.J.: Networks of scientific papers. Science **149**(3683) (1965) 510
10. Cohen, J., Briand, F., Newman, C.: Community food webs: data and theory. Biomathematics. Springer Verlag (1990) Includes bibliographical references.
11. White, J.G., Southgate, E., Thomson, J.N., Brenner, S.: The structure of the nervous system of the nematode Caenorhabditis elegans. Philosophical Transactions of the Royal Society of London **314**(1165) (1986) 1–340
12. Gartner: The Competitive Dynamics of the Consumer Web: Five Graphs Deliver a Sustainable Advantage (2012)
13. Inc, F.: Facebook reports second quarter 2016 results (2016)
14. : Common crawl. `http://commoncrawl.org/` Last Accessed: November 2016.
15. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media, Inc. (2013)
16. : Neo4j. `https://neo4j.com/` Last Accessed: November 2016.
17. Dubey, A., Hill, G.D., Escriva, R., Sirer, E.G.: Weaver: A High-Performance, Transactional Graph Store Based on Refinable Timestamps. In: VLDB. (2015) 852–863
18. : Orientdb. `http://orientdb.com/orientdb/` Last Accessed: November 2016.
19. Brin, S., Page, L.: The anatomy of a large scale hypertextual Web search engine. Computer Networks and ISDN Systems **30**(1/7) (1998) 107–17
20. Gleich, D.F.: PageRank beyond the Web. SIAM Review **57**(3) (2014) 321–363
21. Winter, C., Kristiansen, G., Kersting, S., Roy, J., Aust, D., Knösel, T., Rümmele, P., Jahnke, B., Hentrich, V., Rückert, F., Niedergethmann, M., Weichert, W., Bahra, M., Schlitt, H.J., Settmacher, U., Friess, H., Büchler, M., Saeger, H.D., Schroeder, M., Pilarsky, C., Grützmann, R.: Google goes cancer: Improving outcome prediction for cancer patients by network-based ranking of marker genes. PLOS Computational Biology **8**(5) (05 2012) 1–16
22. Weng, J., Lim, E.P., Jiang, J., He, Q.: Twitterrank: Finding topic-sensitive influential twitterers. Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010) (2010) 261–270

23. Nie, Z., Zhang, Y., Wen, J.R., Ma, W.Y.: Object-level ranking (PopRank). Proceedings of the 14th international conference on World Wide Web - WWW '05 (2005) 567
24. Walker, D., Xie, H., Yan, K.K., Maslov, S.: Ranking scientific publications using a model of network traffic (2007)
25. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113
26. : Apache hadoop. `http://hadoop.apache.org/` Last Accessed: November 2016.
27. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: A peta-scale graph mining system implementation and observations. In: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining. ICDM '09, Washington, DC, USA, IEEE Computer Society (2009) 229–238
28. Cohen, J.: Graph twiddling in a mapreduce world. Computing in Science and Engg. **11**(4) (July 2009) 29–41
29. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. CoRR **abs/1006.4990** (2010)
30. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. PPoPP (2013) 135–146
31. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8) (1990) 103–111
32. : Apache giraph. `http://giraph.apache.org/` Last Accessed: November 2016.
33. Salihoglu, S., Widom, J.: GPS : A Graph Processing System. SSDBM Proceedings of the 25th International Conference on Scientific and Statistical Database Management (2013) 1–31
34. Han, M., Daudjee, K.: Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. Proc. VLDB Endow. **8**(9) (May 2015) 950–961
35. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: Taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems. EuroSys '12, New York, NY, USA, ACM (2012) 85–98
36. Gonzalez, J., Low, Y., Gu, H.: Powergraph: Distributed graph-parallel computation on natural graphs. OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (2012) 17–30
37. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing **20**(1) (1998) 359–392
38. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: Proceedings of the 20th International Conference on World Wide Web. WWW '11, New York, NY, USA, ACM (2011) 607–614
39. Xie, C., Chen, R., Guan, H., Zang, B., Chen, H.: Sync or async: Time to fuse for distributed graph-parallel computation. SIGPLAN Not. **50**(8) (January 2015) 194–204
40. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8) (April 2012) 716–727
41. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12, Berkeley, CA, USA, USENIX Association (2012) 31–46

42. Wang, G., Xie, W., Demers, A.J., Gehrke, J.: 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)
43. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: A distributed framework for prioritized iterative computations. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing. SOCC '11, New York, NY, USA, ACM (2011) 13:1–13:14
44. Zhang, K., Chen, R., Chen, H.: Numa-aware graph-structured analytics. SIGPLAN Not. **50**(8) (January 2015) 183–193
45. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. SOSP '13 (2013) 456–471
46. Sengupta, D., Sundaram, N., Zhu, X., Willke, T.L., Young, J., Wolf, M., Schwan, K. In: GraphIn: An Online High Performance Incremental Graph Processing Framework. Springer International Publishing, Cham (2016) 319–333
47. Ju, W., Li, J., Yu, W., Zhang, R.: igraph: An incremental data processing system for dynamic graph. Front. Comput. Sci. **10**(3) (June 2016) 462–476
48. Iyer, A.P., Li, L.E., Das, T., Stoica, I.: Time-evolving graph processing at scale. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems. GRADES '16, New York, NY, USA, ACM (2016) 5:1–5:6
49. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(October) (1999) 509–512
50. Easley, D., Kleinberg, J.: Networks , Crowds , and Markets : Reasoning about a Highly Connected World. Science **81** (2010) 744
51. Akoglu, L., Tong, H., Koutra, D.: Graph based anomaly detection and description: A survey. Data Min. Knowl. Discov. **29**(3) (May 2015) 626–688
52. Pundir, M., Kumar, M., Leslie, L., Gupta, I., Campbell, R. In: Supporting on-demand elasticity in distributed graph processing. Institute of Electrical and Electronics Engineers Inc., United States (6 2016) 12–21
53. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3) (September 2002) 375–408
54. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (May 1998) 133–169
55. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. USENIXATC'10, Berkeley, CA, USA, USENIX Association (2010) 11–11