Future Generation Computer Systems 96 (2019) 490–501

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Quality of Service (QoS)-driven resource provisioning for large-scale graph processing in cloud computing environments: Graph Processing-as-a-Service (GPaaS)

Safiollah Heidari [*], Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

HIGHLIGHTS

- A novel service-based architecture for processing large-scale graphs on clouds.
- A new multi-handling mechanism for multi-graph processing requests.
- A new auto-scaling algorithm driven by workloads and SLA agreements.
- A new dynamic repartitioning approach to improve usability and performance.

ARTICLE INFO

Article history:
Received 9 June 2018
Received in revised form 31 December 2018
Accepted 22 February 2019
Available online 26 February 2019

Keywords:
Graph processing
Cloud computing
Quality of service
Resource provisioning

ABSTRACT

Large-scale graph data is being generated every day through applications and services such as social networks, Internet of Things (IoT) and mobile applications. Traditional processing approaches such as MapReduce are inefficient for processing graph datasets. To overcome this limitation, several exclusive graph processing frameworks have been developed since 2010. However, despite broad accessibility of cloud computing paradigm and its useful features namely as elasticity and pay-as-you-go pricing model, most frameworks are designed for high performance computing infrastructure (HPC). There are few graph processing systems that are developed for cloud environments but similar to their other counterparts, they also try to improve the performance by implementing new computation or communication techniques. In this paper, for the first time, we introduce the large-scale graph processing-as-a-service (GPaaS). GPaaS considers service level agreement (SLA) requirements and quality of service (QoS) for provisioning appropriate combination of resources in order to minimize the monetary cost of the operation. It also reduces the execution time compared to other graph processing frameworks such as Giraph up to 10%–15%. We show that our service significantly reduces the monetary cost by more than 40% compared to Giraph or other frameworks such as PowerGraph.

1. Introduction

Today data is an asset and being able to collect, store, analyse, protect and use this big data provides companies with critical advantages. Every second huge amount of data is being created by various applications such as social networks, Internet of things (IoT), mobile Apps, bloggers, and even smart web robots that are using artificial intelligent (AI) to produce news. According to [1], during each minute at 2017, 3.3 million posts were put on Facebook, 3.8 million queries were searched on Google search engine, 500 h of new videos were uploaded on YouTube and 448.800 tweets were shared on Twitter. These numbers are almost doubled compared to the amount of content was made per minute in 2014. Moreover, a big fraction of generated data is in the form of graphs. Graph-shape data encompasses a set of vertices that are connected to each other via a set of edges. In a typical social network website, users are vertices and friendship relationships between users form the edges of the graph while in an IoT environment, sensors are considered as vertices and the connections between sensors shape the edges.

Increasing amount of graph data on one side and proven inefficiency of traditional processing approaches such as MapReduce for graphs on the other side [2] resulted in the appearance

[*] Corresponding author.
E-mail address: sheidari@student.unimelb.edu.au (S. Heidari).

https://doi.org/10.1016/j.future.2019.02.048
0167-739X/© 2019 Elsevier B.V. All rights reserved.

of exclusive large-scale graph processing frameworks. Pregel [3] was the first graph processing framework that was introduced by Google in 2010. After that, extensive efforts have been conducted in the research community to develop new processing frameworks or optimize previous ones [4]. However, most existing works have implemented on high performance computing (HPC) environments where the number of resources are considered to be unlimited. So, users do not have to deal with other complicated scenarios such as lack of sufficient computing resources, limited storage space, competitions in order to obtain resources, time limitations, cost limitations, etc. that are possible on distributed environments such as clouds. Based on these assumptions, most current works are concentrating on improving different components of the system namely as partitioning, computing, communication, and I/O.

Unlike HPC, a cloud environment is much more complex in terms of resource provisioning and scheduling [5]. Nevertheless, HPC is not available for everyone and many small/medium companies do not have the resources (budget, professionals, etc.) to own and preserve such infrastructure. Hence, researchers have started investigating cloud-based deployments recently. Cloud computing is a paradigm of computing that has changed software, hardware and datacentres design and implementation. It overcomes restrictions of traditional problems in computing by enabling some novel technological and economical solutions namely as scalability, elasticity and pay-as-you-go models which make service providers free from previous challenges to deliver services to their customers. Cloud computing presents computing as a utility that users access various services based on their requirements without paying attention to how the service is delivered or where it is hosted. It brings many advantages for both service providers and service consumers. For example, providers can virtually locate their services at the shortest distance to their users and decrease latency of delivering their services, which was a problem in traditional computing methods [6]. Because of these benefits, cloud computing has got attracted many attentions in recent years. Among the limitations that make many current graph processing frameworks not to be suitable for deployment in a cloud environment are: (1) they are not able to utilize scalability and elasticity capability of cloud environments, (2) they do not consider monetary cost (processing cost) as a crucial element in cloud computing, (3) they are not designed to take advantage of the heterogeneity of cloud resources which can affect the performance of the system, (4) they cannot work efficiently in a dynamic environment as clouds where for example network metrics are changing constantly.

To choose an appropriate service in a cloud environment, the client investigates some factors that can affect his/her processing requirements. Factors such as processing deadlines, available budget and costs, resource accessibility, etc. are usually taken into consideration for service selection. From there, both the service provider and the customer negotiate on a service level agreement (SLA) [7] by which the quality of service (QoS) will be guaranteed. SLA also determines the conditions of service violation, whose responsibility is to respond and how they can be avoided. An important step is to constantly monitor and evaluate the quality of service against pre-defined factors to ensure that the expected level of quality is provided.

On one hand, according to DB-Engines [8], a database industry observer, graph databases' utilization has been increased dramatically since 2013 and it has surpassed other database models in all popularity rankings ever since. On the other hand, increasing growth in graph data which in turn results in raising processing demands, and the popularity of cloud computing, led to cloud-based design of graph processing frameworks in recent years. ***However, although few graph processing frameworks such as iGiraph*** [9] ***are developed specifically to take advantage of cloud computing features, they do not support quality of service that is provided by these systems on cloud***. Another issue is that current frameworks typically receive "one" large-scale graph dataset as input and return the output after completing the processing. Nevertheless, different users have different priorities while using a system, and when it comes to cloud environments, a framework should be able to handle multiple requests. Several research gaps and open challenges including lack of a comprehensive cloud-based graph processing systems are discussed in [4,10,11]. *Therefore, in this paper we consider large-scale graph processing, "as a service" on cloud*. We used iGiraph to deploy the architecture of our graph processing service on it. The new approach provides a service that like any other services on the cloud, monitors and maintains the quality of service based on the users' requirements and the submitted service level agreement (SLA) while the user does not need to know the details of service implementation to be able to work with it. Our service also makes sure that at any given time during execution, an optimized amount of resources are provisioned to minimize the monetary cost of processing [12]. To the best of our knowledge, this work is the first implementation of a large-scale graph processing framework in which we go beyond simply processing a graph to considering it as a service that can be used by multiple customers on the cloud.

The key ***contributions*** of this work are:

- A novel service-based architecture for processing large-scale graphs on cloud to monitor and maintain the quality of service
- A new multi-handling mechanism for multi-graph processing requests
- A new dynamic auto-scaling algorithm that enables scale up and down according to the characteristics of different arriving workloads and agreements
- A new dynamic repartitioning approach combined with a new mapping strategy to improve the resource usability and performance

The system that we have developed in this work can be used in providing many services such as: (1) finding shortest paths between two or more positions in a geographical positioning system (GPS) where places are the vertices of a large-scale graph and roads are the edges of the graph, (2) finding relevant products by a recommendation algorithm to suggest to customers (products and customers are the vertices of the graph and relationships are the edges), and (3) discovering various patterns in graphs and extracting knowledge using pattern matching algorithms, and so on.

The rest of the paper is organized as follows: Section 2 is providing the related work study by investigating existing research works about large-scale graph processing frameworks and the opportunities for them on cloud environments. Section 3 explains in detail the architecture and workflow of our proposed solution for enabling a service-based graph processing. Section 4 describes the novel dynamic scalable resource provisioning algorithm by which appropriate amount of resources will be provided for every operation based on their requirements. Section 5 provides performance evaluation and Section 6 concludes the paper and identifies directions for future work.

2. Related work

This section discusses various graph processing frameworks and attempts to provide compatibility with cloud environments and challenges.

Table 1
Comparison of the most related works in the literature.

System	Architecture	Implemented environment	Partitioning method	Resource-aware	Scalability	QoS-aware
Pregel [3]	Distributed	HPC	Static	No	No	No
Giraph	Distributed	HPC	Static	No	No	No
PowerGraph [13]	Distributed	HPC	Static	No	No	No
GPS [14]	Distributed	HPC	Dynamic	No	No	No
Pregel.Net [15]	Distributed	Cloud	Dynamic	No	No	No
Surfer [16]	Distributed	Cloud	Dynamic	No	No	No
iGiraph [9]	Distributed	Cloud	Dynamic	Yes	Only scale-in	No
Our work — GPaaS	Distributed	Cloud	Dynamic	Yes	Scale-in/out	Yes

2.1. Different graph processing frameworks

Since 2010, when Google introduced its graph processing framework called Pregel [3], many research works have been conducted to exclusively improve processing of graph data structures. Some graph processing systems such as GraphChi [17], TurboGraph [18], X-Stream [19] and Grace [20] were developed to enable processing based on single-server architecture to operate in-memory. Although, these systems are fast and they do not need to be worried about the communication difficulties between different nodes as their distributed counterparts, they have other restrictions such as limited amount of memory and computing capacity that make them inefficient for more complicated scenarios when the graph is larger than their capacity. On the other side, distributed graph processing frameworks such as Mizan [21], PowerGraph [13], GiraphX [22], Trinity [23], etc. are designed to overcome these issues. However, there are other challenges in distributed environments such as distributed memory, communication, distributed processing and so on that make developing such systems more complex [4]. Many of these challenges have been investigated in various research works and different solutions have been proposed to address them. A summary of most related works along with their notable features are provided in Table 1 and explained in detail in this section.

2.2. Challenges with cloud-based frameworks

One of the less studied areas for graph processing frameworks is cloud environments. Although cloud computing is providing interesting features namely as scalability, elasticity and pay-as-you-go billing model by which large-scale processing can be accessible for everyone, the majority of research works are conducted on high-performance computing (HPC) clusters where they assume that the number of resources are unlimited, resources are always available and there is no need to pay to use the them. The problem is that owning HPC infrastructure to deploy such computations is very costly and many small and medium companies or individuals cannot afford it [12]. Another issue is that because HPC-based frameworks do not need to consider the aforementioned cloud features, they cannot take advantages of their benefits. Even few graph processing frameworks such as Surfer [16] and Pregel.Net [15] that are developed to be used on clouds are not investigating scalability or pricing models. Instead, these systems are trying to reduce the cost of processing by providing faster execution so that they can release the resources quicker. For example, Surfer is offering a bandwidth-aware graph partitioning algorithm that places partitions on VMs according to the VMs' bandwidth and Pregel.Net is evaluating the impact of Bulk Synchronous Parallel (BSP) model [24] on graph processing using Microsoft Azure public clod.

In addition to attempts to improve the performance of processing by ameliorating the computing operation, a system such as iGiraph [9] is also proposing strategies to take advantage of scalability feature of clouds in order to decrease the dollar cost. iGiraph is a Pregel-like graph processing framework that is developed based on popular Giraph.[1] iGiraph is also employing BSP model while it is implemented on top of Hadoop[2] and is using its distributed file system (HDFS). Since cost is a main element for utilizing cloud infrastructure, iGiraph came up with the idea of reducing the number of resources dynamically during the processing rather than using the same amount of resources for the entire operation. It introduced a dynamic repartitioning algorithm that is being applied to the computation at the end of each iteration according to the type of application that is being used. iGiraph categorizes graph applications into two major categories including (1) non-convergent, (2) convergent. When graph data is being processed by a convergent application, the vertices that their status has changed to *inactive* will be eliminated from the memory at the end of every superstep. Therefore, the rest of the graph with active vertices might be fitted into less number of VMs and spare VMs can be terminated. For non-convergent applications in which the status of vertices is always *active* during the operation, utilizing high-degree vertices concept assists the computation to be completed quicker while reducing the communication cost.

2.3. Specific cloud features

Scalability and monetary costs have been investigated separately in few other research works. For example, Pundir et al. [25] have developed a dynamic repartitioning technique based on LFGraph framework [26] in which, similar to iGiraph, they aimed to enable scale out/in by minimizing the network overhead and migrating vertices between machines. In another work, Li et al. [27] have investigated monetary cost of large-scale distributed graph processing on Amazon cloud. Graphic processing units (GPUs) have been also utilized in some works such as [28], where authors are improving the performance of the system by distributing the computation among GPUs to boost the computation speed while others such as [29] are evaluating the performance of single-node frameworks on cloud environments.

Despite the specific development of cloud-based graph processing frameworks, they have never been considered to provide processing as a service on cloud infrastructure. This even make the implementation of graph processing systems harder because there will be new parameters that need to be taken into consideration for delivering an acceptable service [30]. Parameters namely as response time, throughput, cost, etc. are usually negotiated in SLA between the customer and cloud provider to ensure the quality of the provided service. According to Ardagna et al. [31], "Quality of service (QoS) is the problem of allocating resources to the application to guarantee a service level along dimensions such as performance, availability and reliability". QoS in cloud computing has been investigated well in many research works and various techniques have been proposed to monitor and maintain the quality of the service in different platforms [32–34]. However, in order to addressing QoS challenges in the context

[1] https://giraph.apache.org/.
[2] https://hadoop.apache.org/.

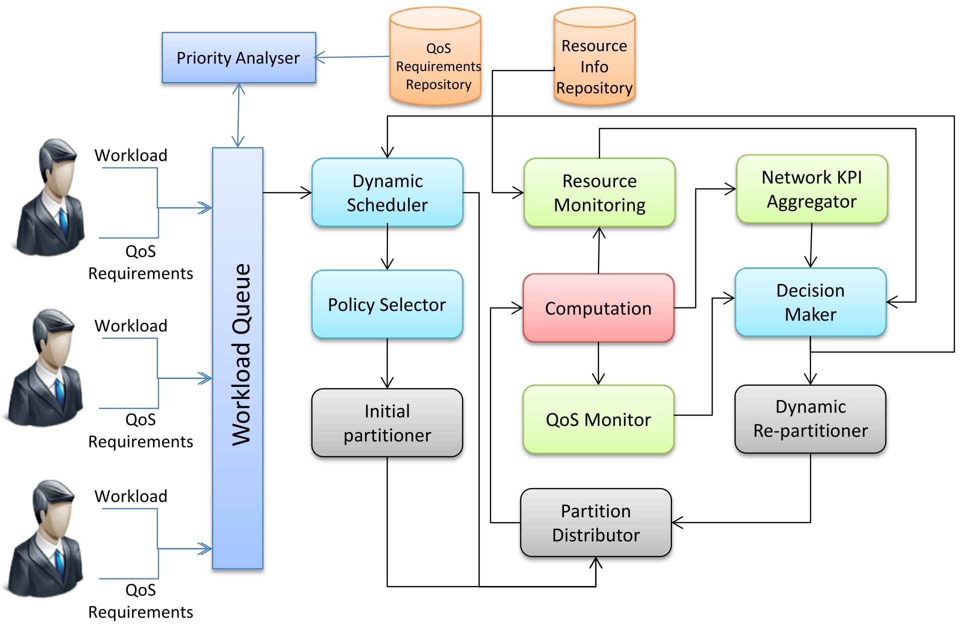


Fig. 1. The workflow of the proposed solution.

of large-scale graph processing, every solution needs to meet specific requirements due to the inherent characteristics of highly connected graph data. In this paper, we are providing a graph processing as a service framework based on our latest version of iGiraph that appeared in [35]. This service enables multiple users to submit their graph processing requests to the system, while the system considers their preferred QoS parameters and provides the best combination of resources to meet the pre-defined requirements. Table 1 shows the comparison of the most related works.

3. Overview of the proposed solution

Figs. 1 and 2 show the workflow and architecture of our proposed solution respectively. The system contains seven different modules that are depicted by seven different colours. These modules include: (1) Users, (2) Repositories, (3) Priority queue, (4) Monitoring, (5) Management, (6) Partitioning, and (7) Computation. Each module comprises a couple of components and is responsible for accomplishing different function while it has input from/output to other parts of the system. Our proposed solution: (1) enables multiple users to apply their jobs at the same time for processing (unlike all other existing frameworks that only accept one job at a time), (2) enables users to submit their QoS requirement for each job (none of existing systems can do so), (3) introduces a new complex workflow to handle intertwined requests, (4) utilizes the heterogeneity of cloud resources with graph algorithm characteristics to reduce the monetary cost of processing, (5) considers various important metrics to adjust dynamic repartitioning in order to meet QoS requirements, (6) can handle multiple scenarios of different job requirements. Here, we explain each module and its components in detail.

3.1. Users

Users provide the input to the system. Each user has to enter two objects into the framework: (1) a large-scale workload or dataset that contains the graph data, and (2) a list of QoS requirements that are derived from the negotiated SLA between customer and service provider. In this paper, we discuss two

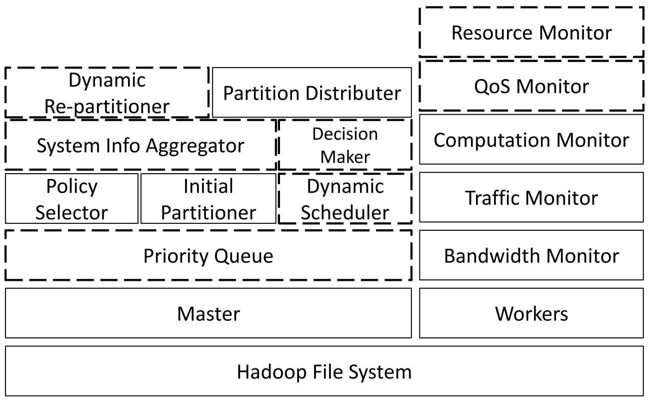


Fig. 2. The components that we added to [36] are shown in dotted rectangles.

factors for QoS and develop algorithms to manage these factors: (a) budget and price, (b) processing time and deadline. Cloud computing features enable us to supply sufficient amount of resources to manage various situations. Cloud providers usually provide a broad range of resources with various characteristics that can be mixed to deal with more complicated requirements and scenarios. For example, if a user has low budget to spend, but he has no deadline for his processing request to be completed, cheaper virtual machines (VMs) can be assigned to his request. Instead, if a user has strict deadline but no budget restriction, more powerful VMs can be dedicated to his request for meeting the deadline properly. In order to provide the user with a prioritization mechanism which helps him to demonstrate his preferences over each QoS requirement, two *priority statuses* have been defined: *(a) Urgent*, *(b) Normal*. *Urgent* refers to the immediacy of a request execution which in turn mentions the execution time. Meanwhile, requests with *Normal* priority compete over low price. Therefore, the user defines the priority of his job by providing his preferred priority status while submitting his request to the system.

3.2. Repositories

There are two main repositories in the system. *QoS requirements repository* includes a set of pre-defined quality conditions and constrains namely as execution time, execution cost, availability, throughput, energy, reliability, etc. In this paper, we consider two important QoS factors including execution time and execution cost. *Resource information repository* contains the information about all the available resources in the resource pool. For instance, for a typical VM, information such as number of cores, memory capacity, usage cost, networks speed, etc. are stored in the repository. Having this information helps the system to make decision about *which* resources and *how* they can be mixed to meet the quality of service (time and cost) properly for a specific request.

3.3. Priority queue

This module comprises two components. As mentioned above, each workload will be submitted with a set of QoS requirements and a priority status. The whole submission is called a *Job* in this system. All jobs will be stored in the *workload queue* where *priority analyser* analyses the priority of each job and reorders them to be processed according to their priority compared to other jobs. Jobs with urgent priority are time constrained with deadline and usually need to be processed before other jobs. So, the first step is to prioritize urgent jobs over normal ones. Next step is to find the execution priority among urgent jobs since there might be more than one urgent job in the queue. In order to do so, a simple version of *Knapsack algorithm* is employed by which urgent jobs will be prioritized based on their required execution time and deadline. Moreover, jobs with normal priority will be processed based on a *first in first out (FIFO)* strategy. The prioritization procedure occurs every time a new job is submitted to the system. However, this might keep some jobs with normal priority in the queue forever because urgent jobs are being submitted constantly. To avoid this, we assign each normal job with a timestamp based on its required execution time (deadline). When the timestamp run out, the job will be considered and treated as an urgent job. This makes sure that no job will be trapped in the queue forever. Algorithm 1 demonstrates the described prioritization mechanism.

3.4. Monitoring module

This module is responsible to constantly monitor the system and measure various metrics that can be used in each processing based on its requirements. The input to this module is coming from the *computation* module where the actual graph processing operation happens. This is because it is very important to track every changes that might affect the processing and use the metrics to enhance the operation. Therefore, the output from monitoring module goes to *management* module where metrics will be used in the decision making and dynamic scheduling processes for the next step. Inputs and outputs of this module will be exchanged after each superstep i and before superstep $i + 1$. Moreover, this is the only module in our proposed solution that is partially implemented on *worker machines*. The reason is that its components need to gather information from workers during the execution. All other modules are implemented on the *master machine*. Monitoring module contains the following components:

– *Resource monitoring*: It is very critical to know about the amount of resources that are available in the resource pool at any moment along with their characteristics. So, this component is placed in the intersection of *resource information repository* and the *computation* module to be able to provide a holistic view of the resource usage situations. It is aware of the amounts and properties of all resources in the repository while it is monitoring the changes that occur to resources that are being used in the operation. The information that this component gathers from the computation part includes: the CPU capacity, memory capacity, monetary cost, VM type, etc.
– *Network Key Performance Indicator (KPI) aggregator*: This component monitors network factors such as network traffic, bandwidth, latency, topology, etc. In this paper, we are using two major factors including traffic and bandwidth in our dynamic repartitioning algorithm. We are using the method that is introduced in [36]. Network KPI aggregator component gathers information from the *computation* module and passes them to the *decision making* component.
– *QoS monitor*: As mentioned before, every job in the system is submitted with a list of SLA requirements which in this paper comprises the customer's preferred *time* and *dollar cost*. Using this information, the system tries to provision the best combination of resources for each job to maintain the quality of service. Like other components in this module, QoS monitor components also receives the input from computation module by watching the mixture of VMs and the execution time of each superstep. It then passes the information to decision making component where various provisioning possibilities will be assessed.

3.5. Management module

Management module is the heart of the system in our proposed architecture. This module is responsible for scheduling the tasks and provisioning the best combination of resources in a way that each job can meet its SLA requirements while ensuring the QoS. It is also responsible to minimize the occurrence of service violation as much as possible. This module collects information from all other modules in the architecture directly or indirectly which enables it to have a comprehensive view on what is happening in the system and the status of other parts. Having such a comprehensive view is a critical pre-requisite for making optimized decisions. All the outputs from this module also directly affect the *partitioning* module. Management module includes three main components as follow:

– *Dynamic scheduler*: Since a cloud provider has to provide services for many users in a cloud computing environment, resources need to be scheduled efficiently to achieve maximum profit. Dynamic scheduler component first becomes active as soon as a job is coming out of the queue to schedule the primary amount of resources for the processing. The number of initial resources will be determined by the user. However, to better utilize the resources, dynamic scheduler takes the size of the submitted dataset and QoS requirements into consideration to select best VM type to start with (Algorithm 2 — Line 1–4). At the beginning of the processing, all VMs will be from the same type. Later during the processing, dynamic scheduler receives the information about the changes in the system from another component in the management module called *decision maker*. This information will be obtained during the intervals between supersteps and will be used to dynamically re-schedule the resources.

Algorithm 1: Prioritization algorithm

```
1:  Queue = receiveInput (Job)
2:  For the entire Queue do
3:      If (getPriority(Job i) == NORMAL) and (getPriority(Job i+1) == URGENT) then
4:          swap(Job i, Job i+1)
5:      If (getPriority(Job i) == URGENT) and (getPriority(Job i+1) == URGENT) then
6:          knapsackJob(Job i, Job i+1)
7:  For any suspendedJob(Job i) in the Queue do
8:      If (priorityTime(Job i) == (Job i).Deadline) then
9:          setPriority(Job i) = URGENT
```

Algorithm 2: Dynamic Scheduler

```
1:  InitialVMs = userInitialVMs(UserVMs)
2:  VMMemory = DatasetSize/InitialVMs
3:  VMType = bringVMWithMemory(VMMemory)
4:  startVM(VMType, InitialVM)
5:  For Superstep1 to the end of computation do
6:      NewInfo = receiveInfo(DecisionMakerVMList)
7:      matchVMWith(NewInfo)
```

– *Policy selector*: Original iGiraph [9] and its extended network-aware version [36] provided a general categorization for various processing environments on clouds and different graph algorithms. This is shown in Fig. 3. Depends on what algorithm is being used for the processing, the user will choose the proper policy for his application while submitting his job. Policy selector component selects the appropriate approach for re-partitioning the graph and informs the system. For example, if the algorithm is convergent and the environment is communicational-intensive, policy selector will pick up a traffic-and-bandwidth-aware [36] strategy for repartitioning.
– *Decision maker*: To help dynamic scheduler with the provisioning of appropriate resources, decision maker component provides a holistic view of the system's state at any given moment. It collects data from *monitoring* module which in turn includes three components. According to the collected data, the system will learn about the available resources and their characteristics, network situation, possible service violations, etc. by which it can intelligently make decision about the amount of resources that is needed for the rest of the operation. Information will be sent to decision maker during the intervals between supersteps. The output of this component will be sent to *partitioning module* and *dynamic scheduler*.

3.6. Partitioning module

This module is responsible for partitioning the graph into smaller jobs and distributes them across the allocated machines. Proper partitioning is the key to improve the performance and speed up the execution of a graph system. Similarly, when graph processing is being provided as a service, suitable partitioning can help to meet the quality of service. However, in the literature, several mechanisms have been proposed for graph partitioning and each tries to increase the efficiency [4]. The inputs for this module are all coming from the *management module* which shows that the resources have been provisioned for computation and partitioning should consider the limitations. Partitioning module comprises three components:

– *Initial partitioner*: When a user submits a job, it will be waiting in the priority queue until its priority is higher than other jobs. Then, it will be passed to dynamic scheduler and policy selector, respectively. At this stage, initial resources have been allocated to the processing and the large graph needs to be partitioned and distributed across the machines. Initial partitioning will be applied to the graph only before the first superstep. The approach for initial partitioning in this paper is a simple random partitioning which is a hash function on vertex IDs. However, the user can replace the simple initial partitioning with more complicated one such as METIS [37] to improve the performance even more.
– *Dynamic re-partitioner*: Unlike initial partitioning that is statistic and happens only at the start of the processing, dynamic re-partitioning changes the partitioning of the graph multiple times during the operation. The aim of dynamic re-partitioning is to match the size and number of partitions with the allocated resources based on graph modification. The core of our dynamic repartitioning algorithm in this work is coming from our other work in which we employed a characteristic-based repartitioning to take advantage of heterogeneous resources on cloud environments [35]. This allows us to achieve better performance with less monetary cost compared to other frameworks such as Giraph.
– *Partition distributor*: When partitions are ready, they need to be distributed across the machines. Entry data to this component might come from the initial partitioner if it is before the first superstep or they can come from dynamic re-partitioning component after the first iteration. The output from this component goes to *computation module* which means that the computation function will be executed on all allocated worker nodes.

3.7. Computation module

Computation module is the computation function that will be executed on graph vertices. This module does not have additional components like other modules. It receives the partitions from the *partitioning module* and applies the *compute()* function on them. So, this function is being implemented on each worker machine. The output of this module is metric measurements that will be passed to the *monitoring module*. Depending on the graph algorithm, status of vertices might change to *inactive* or may remain intact.

4. Dynamic scalable resource provisioning

To ensure that a service is responding properly to SLA requirements for each request, it should be able to employ flexibility for resource provisioning and processing. In this section, we discuss the new multi-handling resource provisioning algorithm for a graph service. In our framework, "dynamic resource provisioning" belongs to the management module and receives inputs from various modules. Our experiments show that using this approach, adequate amount of resources will be assigned to processing jobs and enables them to meet their pre-defined QoS.

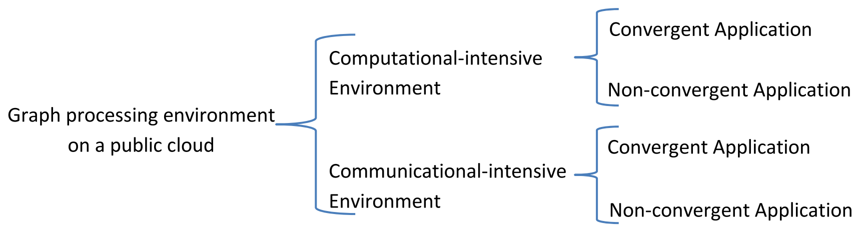

Graph processing environment on a public cloud
Computational-intensive Environment
Communicational-intensive Environment
Convergent Application
Non-convergent Application
Convergent Application
Non-convergent Application

Fig. 3. Graph applications and processing environment categorization [36].

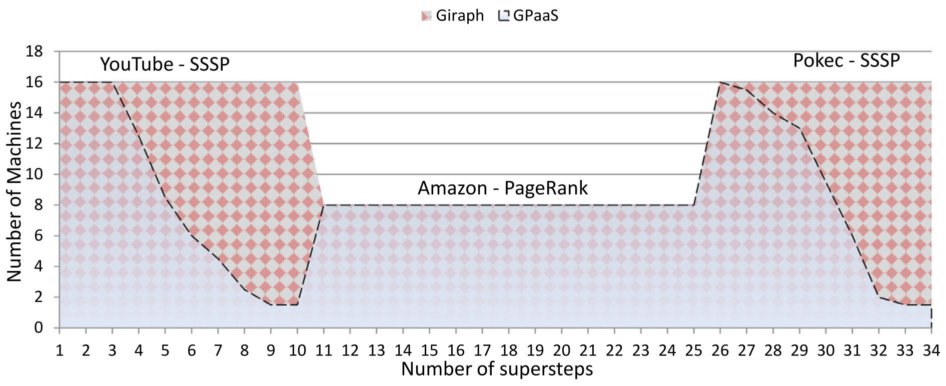


Fig. 4. Scenario1: all jobs with “NORMAL” priority.

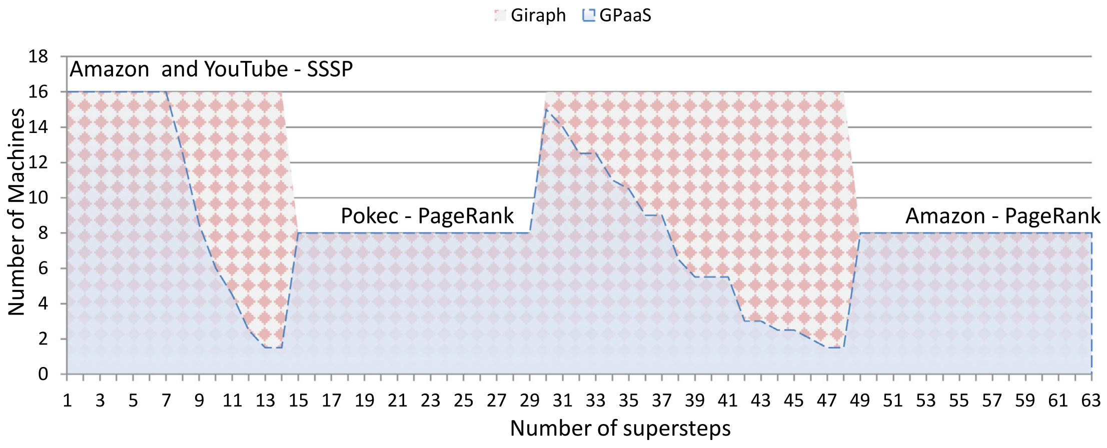


Fig. 5. Scenario 2 – Number of VMs Comparison.

Different jobs with different priorities and requirements will be sent to the graph processing service and they will be processed based on their priorities one after the other. However, there are situations in which while a job is being processed in the system, another job with a strict deadline or higher priority arrives and need to be processed as soon as possible. In a typical scenario, imagine job *A* with *Normal* priority is being assigned a number of resources and it is being processed in the system. Suddenly, job *B* with *Urgent* priority arrives and makes a request for the service. One solution for dealing with this situation is to make the later request to wait until the ongoing processing is finished. In this approach, the urgent request will miss the deadline whereas a possible SLA violation might happen and the service will not be efficient at all.

Another solution, which we implemented in this paper for our service, is to stop the processing, take the less urgent job out of the system and start processing the more urgent job. After completion of the urgent job, the previous job will be brought back to the system to continue its processing from where it was stopped. However, there are some questions that need to be answered here: (1) what will happen to the resources that were being used by the former processing?, (2) how the new processing will receive enough resources to ensure that the requirements will be met?, (3) can we utilize the already existing resources from the previous operation for the new processing?, and (4) do we need to restore the same resources for the less urgent job as the ones it was assigned before being stopped?

Algorithm 3 demonstrates our proposed dynamic scalable resource provisioning mechanism. According to this algorithm, if the priority of the ongoing job in the system is more than the priority of the arriving job, it continues processing. But, if the priority of the arriving job is more than the priority of the ongoing job, then system exchanges the jobs. In this situation, if the applied graph algorithm to the current ongoing job is *convergent* type, in which the status of processed vertices will change to *inactive* and vertices will be removed from the memory, remaining *active*

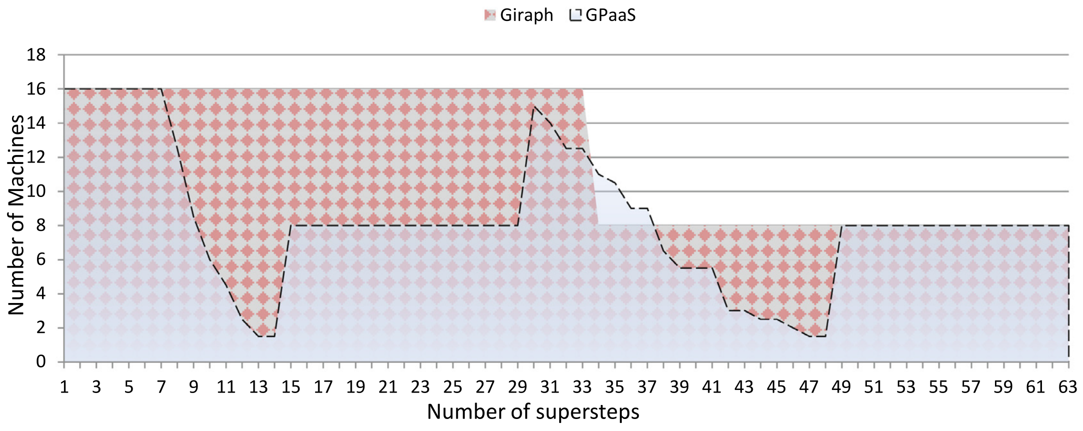


Fig. 6. Scenario 2 – If Giraph follows the job order.

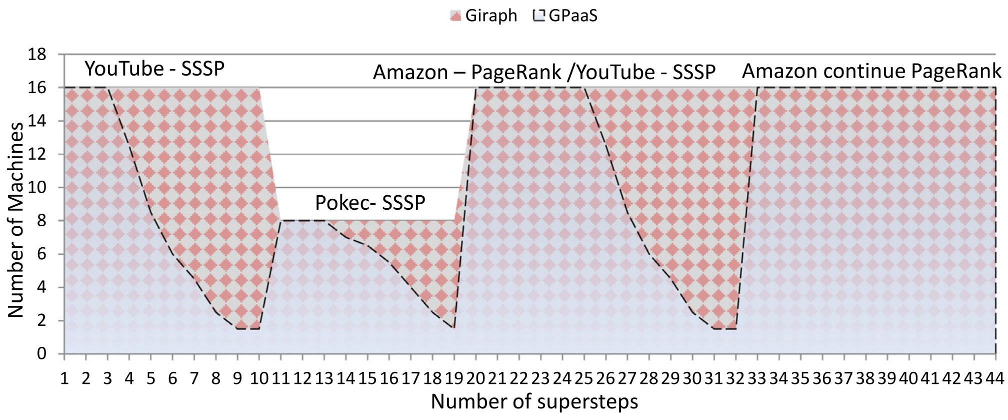


Fig. 7. Scenario 3: jobs with different QoS/deadline requirements.

vertices in the processing will be moved back to the queue. If the applied graph algorithm is *non-convergent* type which does not change the status of vertices, the whole dataset will be moved back to the queue. Then, the new urgent job will be taken from the queue to be loaded for processing. At this phase, instead of terminating the resources from the previous processing, the *dynamic scheduler* calculates the capacity of existing resources in terms of VM types, available memory, available computation power, etc. Meanwhile, it knows the size of arriving job, its QoS criteria, and the number of resources that is ordered by the user at the job submission stage. Following situations are considered in order to provision resources for the new processing job.

(1) If the new dataset is small and current resources can handle the SLA requirements, then there is no need for employing new resources.
(2) If the size of the dataset is big, and the type of current resources is appropriate, then more machines will be employed to reach the resource needs. So, we have a combination of old and new resources that are assigned to the new operation. For example, if there are 3 *medium* VMs left from the previous processing and system learns that 7 medium VMs are needed for the new operation, it only needs to employ 4 more medium VMs ($3mediumold + 4mediumnew = 7mediumrequired$).
(3) If only parts of the existing resources are useable for the new operation, system will keep those VMs and removes the inappropriate ones. Afterwards, it repeats the previous step (step 2). For example, if 4 *medium* and 2 *small* VMs are left from the previous operation and the system learns that the new operation needs 10 *medium* VMs to meet the SLA requirements, it terminates 2 small VMs and employs 6 new medium VMs ($(4mediumold-2smallold) + 6mediumnew = 10mediumrequired$).
(4) If any of the remaining VMs from the previous operation are not suitable for the needs of the new operation, then all of them will be terminated and new appropriate resources will be employed for the new operation.

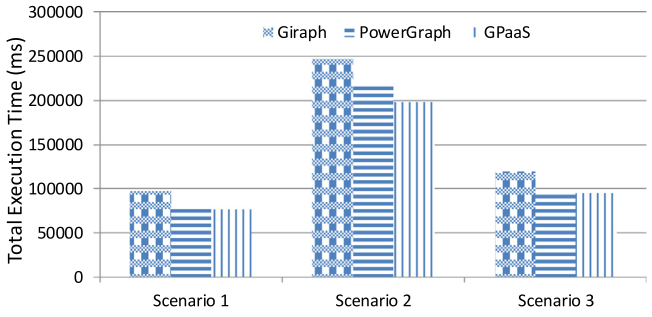


Fig. 8. Total execution time per scenario.

As noted in Algorithm 3 and the described scenarios, our algorithm can both scale up and scale down for provisioning resources. It should be considered that all the operations in this paper will be started with the same VM type. So, if the system learns that for example *large* VM type is suitable for processing, then all VMs at the beginning of the processing will be *large* type whereas if system learns that *medium* VM type is better, then all VMs at the start of the processing will be *medium* type. We will investigate more complicated scenarios such as starting the operation using a combination of different VM types (for example combination of *large* and *medium* VMs) in our future works.

The impact of our proposed mechanism on resource usability is demonstrated in the evaluation section (Figs. 4–8). We show how resources are being provisioned or released based on the SLA requirements (priority, deadline, number of machines, etc.) at each moment in the system. We also show that this approach improves the performance of the system by utilizing resources more intelligently while reducing the execution time (Fig. 8) and monetary costs of the processing operation (Table 6).

5. Performance evaluation

In this section we explain the environment that we conducted our experiments on, and discuss the evaluation results.

5.1. Experimental setup

To evaluate our framework and effectiveness of the proposed algorithms, we utilized resources from Australian national cloud infrastructure (NECTAR) [38]. We utilize three different VM types for our experiments based on NECTAR VM standard categorization: m2.large, m1.medium, and m1.small. Detailed characteristics of NECTAR standard VMs are shown in Table 2. Table 3 describes the utilized VMs in our work with their prices which are determined proportionally based on their closest AWS counterparts. The reason for using m-type VM is because the algorithms that we are using are memory-intensive and using m-type machines provides better performance. Since NECTAR does not correlate any price to its infrastructure for research use cases, the prices for VMs are put proportionally based on Amazon Web Service (AWS) on-demand instance costs in Sydney region according to closest VM configurations as an assumption for this work. According to this, NECTAR m2.large price is put based on AWS m5.xlarge Linux instance, NECTAR m1.medium price is put based on AWS m5.large Linux instance and NECTAR m1.small price is put based on AWS t2.small Linux instance. All VMs have NECTAR Ubuntu 14.04 (Trusty) amd64 installed on them, being placed in the same zone and using the same security policies. We use iGiraph [9] (the extended version of Giraph [39]) with its checkpointing characteristics turned off along with Apache Hadoop version 0.20.203.0 and modify that to contain heterogeneous auto-scaling policies and architecture. All experiments are run using 17 machines where one large machine is always the master and workers are a combination of medium and small instances.

We use single source shortest path (SSSP) [40] and PageRank (PR) [41] algorithms as representatives of convergent and non-convergent graph algorithms respectively for our experiments. They are good representatives of many other algorithms regarding their behaviour. SSSP is solving a particular case of a bigger problem called shortest path (SP) which aims to discover a path with minimum weights of edges between two vertices in a graph. SSSP will find the shortest path between a typical source node and all other vertices in the graph. First, the source node sends its value (which is set to 0 at the beginning) to its adjacent vertices. Those vertices update their value and send their new value to their neighbours. This operation continues until there are no more vertex left to be updated. Whenever a vertex updates its value, its status changes to inactive. So process completes

Table 2
NECTAR standard VM characteristics [38].

VM type	VCPUS	RAM	Total disk
m2.tiny	1	768 MB	5 GB
m2.xsmall	1	2 GB	10 GB
m1.small	1	4 GB	40 GB
m2.small	1	4 GB	30 GB
m2.medium	2	6 GB	30 GB
m1.medium	2	8 GB	70 GB
m2.large	4	12 GB	110 GB
m1.large	4	16 GB	130 GB
m1.xlarge	8	32 GB	250 GB
m2.xlarge	12	48 GB	390 GB
m1.xxlarge	16	64 GB	490 GB

Table 3
Utilized VM characteristics and their proportional cost based on their closest AWS counterparts.

VM type	#Cores	RAM	Disk (root/ephemeral)	Price/h
m2.large	4	12 GB	110 GB (30/80)	$0.24
m1.medium	2	8 GB	70 GB (10/60)	$0.12
m1.small	1	4 GB	40 GB (10/30)	$0.0292

Table 4
Databases' properties.

Graph	Vertices	Edges
YouTube links	1,138,499	4,942,297
Amazon (TWEB)	403,394	3,387,388
Pokec	1,632,803	30,622,564

when all vertices' status change to inactive. This is why SSSP is a convergent algorithm. On the other hand, a vertex status remains intact in PageRank algorithm which makes it to be categorized as a non-convergent algorithm. PageRank weighs the significance of websites and web pages by calculating the number of links that are connected to them (hyperlinks). The more connected links a page has, the more important the page is. This algorithm values each page solely and does not value the entire website as a unit.

We also use three real-world datasets of different sizes: YouTube, Amazon, and Pokec [42] as shown in Table 4.

5.2. Experiments and results

We have compared our systems and algorithms with Giraph because it is a popular open-source Pregel-like graph processing framework and is broadly adopted by many companies such as Facebook [43]. To evaluate different scenarios by our service, we have provided various workloads and jobs by combining the datasets from Table 3 with different characteristics. Table 5 demonstrates input jobs and the order of inputs along with their properties.

Scenario 1: This is the simplest situation in which all jobs in the queue have the same priority as "normal". In this situation, deadline is not very important for the processing, so all jobs will be executed by a first-in-first-out (FIFO) approach and it is fine if any deadline was missed. However, as can be seen in Fig. 4, the cost of processing in our service is much less than conducting it on a popular framework as Giraph. The reason is that our service scales up and down to provision the best combination of resources for the processing while Giraph uses the same amount of resources for the entire operation. Note that in processing graphs by PageRank algorithm, the number of VMs for both Giraph and our service is the same because PageRank is a *non-convergent* algorithm. We also consider up to 20 supersteps for PageRank algorithm in all our experiments. In our future research work,

Algorithm 3: Dynamic scalable resource provisioning

```
1:  If ((getPriority(CurrentJob)==URGENT) and (getPriority(ArrivingJob)==NORMAL)) then
2:      continueWithNoChange()
3:  If ((getPriority(CurrentJob)==NORMAL) and (getPriority(ArrivingJob)==URGENT)) then
4:      backToQueue(CurentJob.ActiveVertex)
5:      If (currentVMMemory(AvailableVMs) ==DatasetSize) and (AvailableVMs<InitialVM) then
6:          continueWithCurrentConfig()
7:      If (currentVMMemory(AvailableVMs)<DatasetSize) and (AvailableVMs<InitialVM) then
8:          onlyKeepVM(VMType)
9:          update(AvailableVMs)
10:         NeededVMs = InitialVM – AvailableVMs
11:         Start(VMType , NeededVMs)
12:         executeWithNewConfig()
13:     If (currentVMMemory(AvailableVMs)>DatasetSize) and (AvailableVMs>InitialVM) then
14:         onlyKeepVM(VMType)
15:         update(AvailableVMs)
```

Table 5
Input scenarios for evaluation.

Scenarios	Dataset	Input order	Priority	Submission time (s)	Deadline (s)	Number of initial VMs	Algorithm
Scenario 1	YouTube	1	Normal	0	30	16	SSSP
	Amazon	2	Normal	5	80	8	PR
	Pokec	3	Normal	7	110	16	SSSP
Scenario 2	Amazon	1	Normal	0	50	16	SSSP
	YouTube	2	Urgent	6	30	16	SSSP
	Pokec	3	Urgent	8	80	8	PR
	Amazon	4	Normal	15	110	8	PR
Scenario 3	Pokec	1	Urgent	0	60	8	SSSP
	YouTube	2	Urgent	1	30	16	SSSP
	Amazon	3	Normal	12	130	16	PR
	YouTube	4	Urgent	15	90	16	SSSP

we will find the best combination to reorder the queue in case if deadlines are different so jobs will be processed to meet their deadline as well.

Scenario 2: In this situation a combination of "normal" and "urgent" jobs are arriving to the service for processing. According to Algorithm 1 and Algorithm 3, when a normal job is getting processed, it should be replaced by the urgent job as soon as such job is arrived to the system. Nevertheless, the normal job cannot wait in the queue forever only because urgent jobs are being submitted constantly. To resolve this situation, when the normal job goes back to the queue to be replaced by an urgent job, a deadline will be set for it so that its priority will change to urgent when the deadline arrives. Fig. 5 shows how this scenario works and Fig. 6 demonstrates the scenario in which Giraph follows the job order and depicts what is happening in reality.

Scenario 3: In this scenario, jobs are different in terms of their deadline. So, when two jobs with the same urgent priority arrive, the one with closer deadline will be processed first. Fig. 7 shows the processing order in this scenario and compares that with Giraph.

We conducted the same experiments on PowerGraph [13], an edge-centric distributed graph processing framework. PoweGraph outperforms Giraph due to its vertex-cut strategy and implemented optimizations to speed up the execution on natural graphs with "highly skewed power-law degree distributions" [27]. However, **PowerGraph's processing pattern is the same as Giraph as shown in Figs. 4–7 while performing under various scenarios**. *The reason is that, like Giraph, PowerGraph does not have any priority recognition or other mechanisms to distinguish between the priorities of different jobs. So, it executes jobs based on first-in-first-out (FIFO) approach*. Similarly, it does not distinguish between different graph algorithms' behaviour (convergent, non-convergent, etc.), hence it cannot utilize the resources efficiently.

Fig. 8 demonstrates the execution time in our service against Giraph and PowerGraph for each scenario. It shows that our proposed service completes faster than both Giraph and PowerGraph due to its dynamic resource provisioning and scheduling. GPaaS also eliminates overheads for manual job submissions after each process completion. It reduces the cost even more because resources will be released quicker. In Table 6, monetary cost of each scenario in three different systems are being compared. It shows that using GPaaS, the user has to pay much less (more than 40% less in some cases) for performing the same job when compared to Giraph and PowerGraph. Whereas, using PowerGraph can save more money than Giraph due to its faster execution. The cost here is calculated based on the amount of time that various resources have been utilized in each system. In both Giraph and PowerGraph, the number of provisioned machines remains the same during the entire processing which is a very expensive approach while there is no need to keep all machines in use if the behaviour of the algorithm and operation characteristics are considered. The number and configurations of utilized resources (machines) in GPaaS are being updated regularly to obtain the efficient combination of VMs in order to minimize the cost.

6. Conclusions and future work

Many applications such as social networks, mobile applications, IoT devices and applications, etc. are generating huge

Table 6
Processing cost for each scenario in different systems.

	Giraph	PowerGraph	GPaaS
Scenario 1	$0.0399	$0.0302	$0.0185
Scenario 2	$0.0532	$0.0483	$0.0342
Scenario 3	$0.0516	$0.0428	$0.0294

amount of data which a considerable fraction of it is graph data. Due to the inefficiency of traditional processing solutions such as MapReduce, several unprecedented frameworks are developed to address the challenges of large-scale graph processing. Many of these frameworks are designed to operate on HPC environments rather than clouds. Since HPC infrastructure is not available to everyone, cloud computing with its unprecedented features such as elasticity and pay-as-you-go billing model is a suitable candidate for implementing the frameworks on as it can be accessible easier too. However, the few existing frameworks that are developed exclusively to be used on cloud environments have many limitations and cannot guarantee the quality of services as it is expected in negotiated SLA between cloud provider and clients. In this paper, we have proposed the first large-scale graph processing service on cloud (graph processing-as-a-service). Unlike graph processing frameworks, our service can handle multiple processing requests while it considers each request's priorities and requirements to avoid SLA violations. Our proposed architecture and algorithms such as dynamic scheduling and dynamic resource provisioning make it possible to utilize the heterogeneous cloud resources efficiently in order to respond the requests. This service can be used for many real-world applications such as finding shortest path in GPS systems, recommendation systems, pattern recognition, knowledge extraction and data analytics systems that require processing large-scale graph data. Our evaluation results presented that our service can handle graph processing requests successfully to a high extent. To achieve this, three real-world datasets (YouTube, Amazon and Pokec) were used in three different scenarios. We observed that GPaaS can minimize the monetary cost more than 40% by utilizing resources intelligently and executes faster when compared with Giraph and PowerGraph — two popular distributed graph processing frameworks. It also reduces the execution time up to 20%. This means that customers can save a lot of money and time while the quality of service is being maintained.

As part of the future work, we plan to improve our proposed system by enabling it to utilize various combinations of resources to start a processing with, instead of starting with the same VM types for all resources. We will also consider other network factors such as network latency and topology to investigate their impact on the computation and if they can improve it.

References

[1] R. Allen, What happens online in 60 seconds?, Smart Insights, 06 02 2017. [Online]. Available: https://www.smartinsights.com/internet-marketing-statistics/happens-online-60-seconds/ [Accessed 22 03 2018].

[2] F.N. Afrati, A. Das Sarma, S. Salihoglu, J.D. Ullman, Vision paper: towards an understanding of the limits of map-reduce computation, in: Proceedings of the Cloud Futures 2012 Workshop, Berkeley, California, USA, 2012.

[3] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 2010.

[4] S. Heidari, Y. Simmhan, R.N. Calheiros, R. Buyya, Scalable graph processing frameworks: a taxonomy and open challenges, ACM Comput. Surv. 51 (3) (2018) 1–53.

[5] B. Varghese, R. Buyya, Next generation cloud computing: new trends and research directions, Future Gener. Comput. Syst. 79 (2) (2018) 849–861.

[6] G. Pallis, Cloud computing: the new frontier of internet computing, IEEE Internet Comput. 14 (5) (2010) 70–73.

[7] P. Patel, A. Ranabahu, A. Sheth, Service Level Agreement in Cloud Computing, CORE Scholar, Dayton, OH, USA, 2009.

[8] DBMS Popularity Broken Down by Database Model, DB-Engines, [Online]. Available: https://db-engines.com/en/ranking_categories [Accessed 30 08 2018].

[9] S. Heidari, R.N. Calheiros, R. Buyya, Igiraph: a cost-efficient framework for processing large-scale graphs on public clouds, in: Proceedings of 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '16), Cartagena, Colombia, 2016.

[10] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, T. Özsu, The ubiquity of large graphs and surprising challenges of graph processing, VLDB Endow. 11 (4) (2017) 420–431.

[11] J. Wang, Q. Wu, H. Dai, Y. Tan, Challenges in large-graph processing: a vision, in: Proceedings of the 5th International Conference on Computer Science and Network Technology (ICCSNT), Changchun, China, 2016.

[12] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.

[13] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, 2012.

[14] S. Salihoglu, J. Widom, GPS: a graph processing system, in: Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM), Baltimore, Maryland, 2013.

[15] M. Redekopp, Y. Simmhan, V.K. Prasan, Optimizations and analysis of BSP graph processing models on public clouds, in: Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13), Boston, MA, 2013.

[16] R. Chen, X. Weng, B. He, M. Yang, Large graph processing in the cloud, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '10), Indianapolis, IN, USA, 2010.

[17] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: large-scale graph computation on just a PC, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12), Hallywood, USA, 2012.

[18] W.S. Han, S. Lee, K. Park, J.H. Lee, M.S. Kim, J. Kim, H. Yu, Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13), Chicago, IL, 2013.

[19] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: edge-centric graph processing using streaming partitions, in: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13), Farmington, USA, 2013.

[20] G. Wang, W. Xie, A. Demers, J. Gehrke, Asynchronous largescale graph processing made easy, in: Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR;13), Asilomar, USA, 2013.

[21] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoo, D. Williams, P. Kalnis, Mizan: a system for dynamic load balancing in large-scale graph processing, in: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13), Prague, Czech Republic, 2013.

[22] S. Tasci, M. Demirbas, Giraphx: parallel yet serializable large-scale graph processing, in: Proceedings of the 19th international Conference on Parallel Processing (Euro-Par'13), Aachen, Germany, 2013.

[23] B. Shao, H. Wang, Y. Li, Trinity: a distributed graph engine on a memory cloud, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '13), New York, USA, 2013.

[24] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.

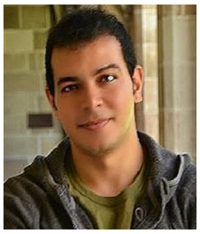
[25] M. Pundir, M. Kumar, L.M. Leslie, I. Gupta, R.H. Campbell, Supporting on-demand elasticity in distributed graph processing, in: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E'16), Berlin, Germany, 2016.

[26] I. Hoque, I. Gupta, Lfgraph: simple and fast distributed graph analytics, in: Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13), Farmington, Pennsylvania, USA, 2013.

[27] Z. Li, T.N. Hung, S. Lu, R.S.M. Goh, Performance and monetary cost of large-scale distributed graph processing on amazon cloud, in: Proceedings of the International Conference on Cloud Computing Research and Innovations (ICCCRI'16), Singapore, Singapore, 2016.

[28] T. Zhang, W. Tong, W. Shen, J. Peng, Z. Niu, Efficient graph mining on heterogeneous platforms in the cloud, in: Proceedings of the Lecture Notes of the Institute for Computer Sciences, Cham, Switzerland, 2017.

[29] C. Xu, J. Zhou, Y. Lu, F. Sun, L. Gong, C. Wang, X. Li, X. Zhou, Evaluation and trade-offs of graph processing for cloud services, in: Proceedings of the 24th International Conference on Web Services (ICWS'17), Honolulu, HI, USA, 2017.
[30] K. Xiong, H. Perros, Service performance and analysis in cloud computing, in: Proceedings of the World Conference on Services, Los Angeles, CA, USA, 2009.
[31] D. Ardagna, G. Casale, M. Ciavotta, J.F. Pérez, W. Wang, Quality-of-service in cloud computing: modeling techniques and their applications, J. Internet Serv. Appl. 5 (11) (2014) 1–17.
[32] P. Manuel, A trust model of cloud computing based on quality of service, Ann. Oper. Res. 233 (1) (2013) 281–292.
[33] J. Mei, A. Ouyang, K. Li, A profit maximization scheme with guaranteed quality of service in cloud computing, IEEE Trans. Comput. 64 (11) (2015) 3064–3078.
[34] J.Y. Lee, J.W. Lee, D.W. Cheun, S.D. Kim, A quality model for evaluating software-as-a-service in cloud computing, in: Proceedings of the 7th ACIS International Conference on Software Engineering Research, Management and Applications, Haikou, China, 2009.
[35] S. Heidari, R. Buyya, A cost-efficient auto-scaling algorithm for large-scale graph processing in cloud environments with heterogeneous resources, IEEE Trans. Softw. Eng. (2018) (Second Revision).
[36] S. Heidari, R. Buyya, Cost-efficient and network-aware dynamic repartitioning-based algorithms for scheduling large-scale graphs in cloud computing environments, Softw.: Pract. Exper. 48 (12) (2018) 2174–2192.
[37] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, in: Proceedings of the International Conference on Parallel Processing(ICPP'95), Raleigh, NC, US, 1995.
[38] NECTAR Cloud, [Online]. Available: http://nectar.org.au/research-cloud/ [Accessed 10 09 2018].
[39] Apache Giraph!, Apache, [Online]. Available: https://giraph.apache.org/ [Accessed 31 10 2017].
[40] P. Roy, A new memetic algorithm with GA crossover technique to solve single source shortest path (SSSP) problem, in: Proceedings of the Annual IEEE India Conference (INDICON), Pune, India, 2014.
[41] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web, Stanford InfoLab, 1998.
[42] J. Kunegis, KONECT - The Koblenz network collection, in: Proceedings of International. Web Observatory Workshop, 2013.
[43] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: Graph processing at facebook-scale, VLDB Endow. 8 (12) (2015) 1804–1815.



Safiollah Heidari is a PhD student at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems (CIS), The University of Melbourne, Australia. He is doing his PhD, supported by International Research Scholarship (MIRS) and Melbourne International Fee Remission Scholarship (MIFRS), at the CIS department of the University of Melbourne. He has a BSc of software engineering and a MSc in information management systems. He has been given a number of awards including Google PhD Travel prize and runner-up prize for one of his papers at the IEEE Victorian students best paper award. His research interests include scheduling and resource provisioning for distributed systems. Currently he is working on large-scale graph processing scheduling and resource provisioning approaches in cloud environment.

Rajkumar Buyya is a Fellow of IEEE, Professor of Computer Science and Software Engineering and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He served as a Future Fellow of the Australian Research Council during 2012–2016. He has authored over 525 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He also edited several books including "Cloud Computing: Principles and Paradigms" (Wiley Press, USA, Feb 2011). He is one of the highly cited authors in computer science and software engineering worldwide (h-index=118, g-index=225, 72,200+ citations). Recently, Dr. Buyya is recognized as "2016 Web of Science Highly Cited Researcher" by Thomson Reuters. Software technologies for Grid and Cloud computing developed under Dr. Buyya's leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. Manjrasoft's Aneka Cloud technology developed under his leadership has received "2010 Frost & Sullivan New Product Innovation Award". Recently, Dr. Buyya received "Bharath Nirman Award" and "Mahatma Gandhi Award" along with Gold Medals for his outstanding and extraordinary achievements in Information Technology field and services rendered to promote greater friendship and India–International cooperation. He served as the founding Editor-in-Chief of the IEEE Transactions on Cloud Computing. He is currently serving as Co-Editor-in-Chief of Journal of Software: Practice and Experience, which was established over 45 years ago. For further information on Dr.Buyya, please visit his cyberhome: www.buyya.com