A Cost-Efficient Auto-Scaling Algorithm for Large-Scale Graph Processing in Cloud Environments with Heterogeneous Resources

Safiollah Heidari, *Member, IEEE* and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Graph processing model is being adopted extensively in various domains such as online gaming, social media, scientific computing and Internet of Things (IoT). Since general purpose data processing tools such as MapReduce are shown to be inefficient for iterative graph processing, many frameworks have been developed in recent years to facilitate analytics and computing of large-scale graphs. However, regardless of distributed or single machine based architecture of such frameworks, dynamic scalability is always a major concern. It becomes even more important when there is a correlation between scalability and monetary cost - similar to what public clouds provide. The pay-as-you-go model that is used by public cloud providers enables users to pay only for the number of resources they utilize. Nevertheless, processing large-scale graphs in such environments has been less studied and most frameworks are implemented for commodity clusters where they will not be charged for the resources that they consume. In this paper, we have developed algorithms to take advantage of resource heterogeneity in cloud environments. Using these algorithms, the system can automatically adjust the number and types of virtual machines according to the computation requirements for *convergent graph applications* to improve the performance and reduce the monetary cost of the entire operation. Also, a smart profiling mechanism along with a novel dynamic repartitioning approach helps to distribute graph partitions expeditiously. It is shown that this method outperforms popular frameworks such as Giraph and decreases more than 50 percent of the dollar cost compared to Giraph.

Index Terms—Cloud computing, large-scale graph processing, auto-scaling, cost saving, heterogeneous resources

1 Introduction

Graph-Oriented data has grown to a very large-scale and it is becoming massively critical with the emergence of social networks and Internet of Things (IoT) [1]. However, traditional data processing approaches such as MapReduce [2] are not suitable for processing graphs due to the inherent iterative characteristic of graph algorithms [3]. This has led to the development of graph processing frameworks such as Pregel [4], GraphLab [5], and others [6], [7] that can perform efficiently on various iterative graph algorithms [8], [9].

Although it has been shown that graph processing systems offer a good level of scalability on fast interconnected high-performance computing machines, their behavior on "virtualized commodity hardware" – known as cloud computing - is less studied [10]. Public cloud has become more popular by offering cost scalability through provisioning on-demand computing resources based on its *pay-as-you-go model* where resource access is democratized. However, there are issues that affect the advantages of using such systems including: 1) the overhead for virtualizing infrastructure on a commodity cluster, 2) performing in controlled situations and environments, 3) lack of complete control on communication bandwidth and latency due to imperfect virtual machine (VM) placement. On one side, users may value the monetary cost more than reliability or performance while selecting a public cloud service. On the other side, while many scientific computing need to utilize more than thousands of cores on a high-performance cluster, the dollar cost of public cloud resources restricts the number to tens/hundreds. Therefore, scientific applications that require resources beyond a single large server and less than a huge cluster of high-performance nodes can fit the elasticity of public clouds.

Cloud providers usually provide a wide range of resources including various types of virtual machines so that customers can find the best combination to fulfill their requirements with different priorities. In fact, the adoption of *heterogeneous computing resources* (i.e., VMs with different configurations) by cloud users will enable improving the efficiency of resources utilization.

Despite the significant impacts of *elasticity* and *cost* in cloud environments, investigating these features for graph processing systems' performance on such platforms is still a major gap in the literature. Few graph processing frameworks such as Pregel.Net [10] (with its Bulk Synchronous Parallel (BSP) model [11]) and Surfer [12] are developed to be used on public clouds in order to process large graphs but they are investigating only particular characteristics other than scalability and monetary cost.

• *The authors are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, the University of Melbourne, Parkville, VIC 3010, Australia. E-mail: sheidari@student.unimelb.edu.au, rbuyya@unimelb.edu.au.*

Manuscript received 4 Apr. 2018; revised 18 June 2019; accepted 30 July 2019. Date of publication 14 Aug. 2019; date of current version 13 Aug. 2021. (Corresponding author: S. Heidari.) Recommended for acceptance by A. Zisman. Digital Object Identifier no. 10.1109/TSE.2019.2934849

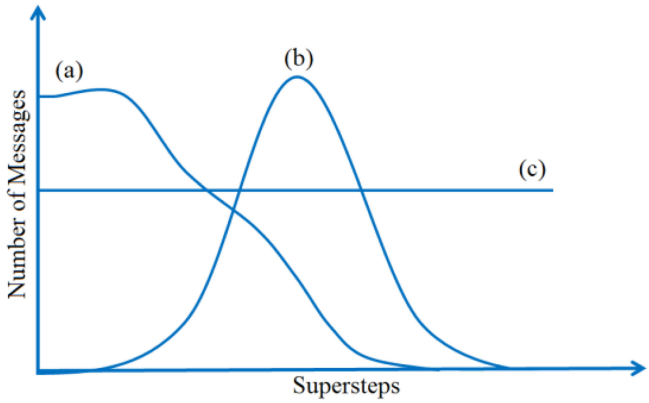


Fig. 1. General patterns of the number of messages passing through the network during a typical processing show convergence by the end of the operation for (a) CC and (b) SSSP, but (c) PageRank is not converged.

Scalability is another important feature that can help cloud applications to gain optimal performance and minimizes the cost. iGiraph [13] (a Pregel-like graph processing framework based on Apache Giraph [14]) deploys a scalable processing approach on clouds. It proposes a dynamic repartitioning method to decrease the number of VMs during the operation. This method uses network message traffic patterns to merge or move partitions across workers which eventually reduces the monetary cost of resource utilization. However, iGiraph works only with homogeneous resources instead of heterogeneous VMs.

Distributed graph processing contains a set of iterations in which graph partitions will be placed on different machines (workers). The operation continues until the expected result is achieved or there are no more vertices to be processed. *An effective approach to minimize the cost in such a system is to provide the best combination of resources (i.e., appropriate number of resources with the right type) out of the available resource pool at any iteration*. To utilize the aforementioned capacity of public clouds in providing heterogeneous computing resources in the context of large-scale graph processing, we equipped iGiraph with an auto-scaling algorithm to minimize the cost of processing. Our approach significantly reduces the financial cost of utilizing cloud resources compared to other popular graph processing frameworks such as Giraph [14] and ensures faster execution. To the best of our knowledge, this work is the first implementation of a graph processing framework for scalable use of heterogeneous resources in a cloud environment. This approach is very effective when the monetary cost is important for the user.

The key *contributions* of this work are:

- A new cost-efficient provisioning of heterogeneous resources for convergent graph applications.
- A new resource-based auto-scaling algorithm.
- A new characteristic-based dynamic repartitioning method combined with a smart process monitoring that allows efficient partitioning of the graph across available VMs according to VM types.
- A new implementation of operation management on the master machine.

The rest of the paper is organized as follows: Section 2 describes graph applications and the proposed auto-scaling method that we use in this paper. Section 3 explains the scaling policy and how we are going to apply it to the heterogeneous resources in a cloud environment for processing large-scale graphs. Section 4 explains the proposed approaches and algorithms for repartitioning and processing graphs in such a heterogeneous environment. The implementation and evaluation of the proposed mechanisms are discussed in Section 5, while related works are studied in Section 6. Finally, we conclude the paper along with the directions for future work in Section 7.

2 Graph Applications and Auto-Scaling Architecture

In this section, we discuss in details the applications that we used along with our proposed auto-scaling architecture.

2.1 Applications

According to iGiraph [13], when it comes to processing, there are two types of graph algorithms: 1) non-convergent algorithms, and 2) convergent algorithms. During processing a large-scale graph by a non-convergent algorithm such as PageRank [8], the number of messages that are being created in every iteration (superstep) is the same and will not change until the end of the operation (Fig. 1c).

On the other side, while processing a graph by a convergent algorithm, the number of messages in the network will start getting decreased at some point during the operation and will continue reducing until the end of processing (Figs. 1a and 1b). This is because the more iterations are completing, the more vertices become deactivated (i.e., processed), so they do not need to exchange messages with their neighbors anymore. As a result, deactivated vertices (i.e., processed vertices) can be kept outside the memory in an operation that is using a convergent algorithm. It means that the remaining active vertices can be processed by using less amount (or smaller type) of resources. This continues until there are no more vertices left to be processed or the desirable result has been achieved. Therefore, the processing can be dynamically scalable. Two important algorithms in this category are single source shortest path [15] and connected components [9] that have been used in many studies.

Single source shortest path (SSSP): single source shortest path has derived from general shortest path problem. In a directed graph, the aim of SSSP is to find the shortest path from a given source vertex r to every other vertex $v \in V - \{r\}$. The weight (length) of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges: $w(p) = \sum w(v_{i-1}, v_i)$. At the beginning of the algorithm, the distance (value) for all nodes will be set to INF (∞) except the source node that will be set to zero (0) (because it has zero distance from itself). During the first iteration in a graph processing operation, all neighbors of the source node will be updated by receiving its value and update their distance values. In the second iteration, the updated neighbors will send their values to their own adjacent vertices and this will continue until all vertices in the graph update their value and there is no more active node in the graph. Changing the status of

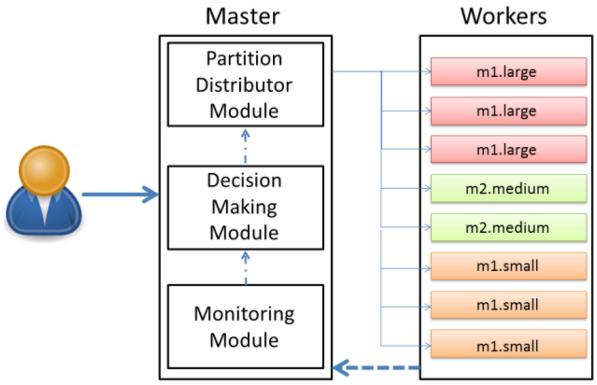


Fig. 2. Proposed auto-scaling architecture.

processed vertices during the operation often means we do not need them for the rest of processing. Therefore, SSSP is a convergent algorithm.

Connected components (CC): Connected components algorithm is for detecting various sub-graphs in a large graph where there is a route between any two nodes of the sub-graph but it may not be connected to all nodes in the large graph. A highly connected component algorithm starts by setting all graph nodes' status to active. At the start of the computation, each node's ID will be considered as its initial component ID. The component ID can be updated if a smaller component ID is sent to the node. Then, the node will send its new value to its neighbors. In this operation, the number of messages required to be passed between vertices will reduce as the processing progresses. It is because the states of vertices change to inactive during the operation. Similar to SSSP, CC is also a convergent algorithm that can be considered for our auto-scaling approach.

We have targeted convergent algorithms in this paper as they are more suitable for scaling scenarios. We will show how they can benefit from resource heterogeneity in a public cloud by using our proposed auto-scaling and repartitioning algorithms and framework.

2.2 Proposed Auto-Scaling System Architecture

Most distributed graph processing frameworks only rely on homogenous implementation while trying to reduce the cost by speeding up the computation and decreasing the execution time [16], [17], [18]. These frameworks consider dedicated clusters in various sizes, whereas in real world it is not possible for all users to provide such infrastructure. Instead, from a user point of view, it is beneficial to use public clouds for processing large-scale graphs [19]. There are many important issues that influence the final performance and cost of the processing. They include: 1) what is the best scaling policy (i.e., horizontally or vertically) to reduce the cost?, 2) what is the best partitioning method to take advantage of more cost-efficient VMs?, 3) how these policies can be applied to a graph processing framework?, and 4) how to improve the system performance on public cloud? To enhance the performance of large-scale graph processing on public clouds, first, we implement an auto-scaling approach within our framework to utilize the heterogeneity of resources in this environment.

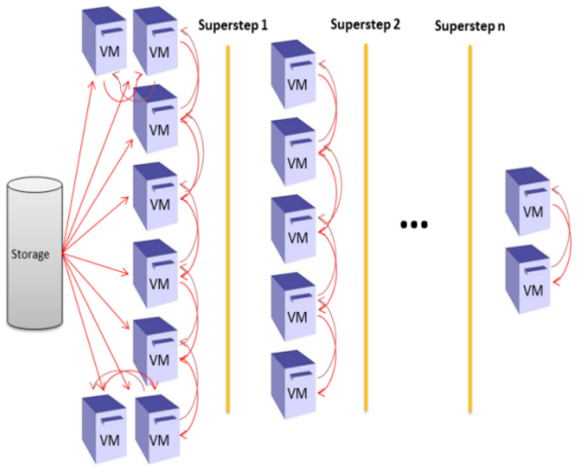
As shown in Fig. 2, our proposed auto-scaling system is aware of the states of available machines at any moment. The system consists of a *monitoring module* by which it tracks different states of each machine and network metrics such as the number of generated messages, memory utilization, CPU utilization, VM info, etc. There is also a *decision-making module* that decides how to apply the right scaling policy based on the information gathered about current situations of VMs, network and the graph itself. Finally, the *partition distributor module* distributes the partitions across the available VMs according to the computing strategy. All these modules are implemented on the master machine that controls the entire processing and partition assignments.

At the end of each superstep, the monitoring module collects various information from all workers about the current state of the system, network and the graph. Then it passes the information to decision making module. Decision-making module compares new information with information from the previous superstep and investigates different scenarios to replace VMs in order to reduce the cost. For each calculation, the cost of iteration $i+1$ should be *equal to* or *less than* the cost of iteration i. If migrating vertices and merged partitions to smaller/less costly VMs decreases the cost of iteration compared to the previous iteration, then current VMs will be replaced by new ones. Otherwise, the current configuration will be remained untouched. This module also determines the number of VMs that can be replaced along with the types of new VMs based on the information from monitoring module. Partition distributor module will be notified about the new configuration and eventually distributes new partitions accordingly.

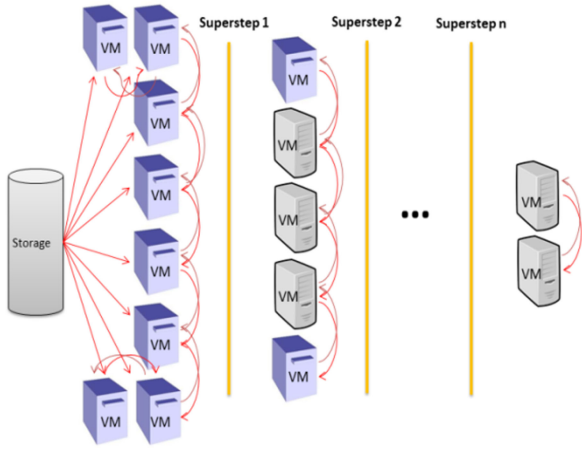
3 Horizontal Scaling (Step Scaling)

Horizontal scaling is simply adding more machines to the existing configuration of resources. Although scaling happens based on additional needs to new resources, adding new machines does not necessarily mean provisioning more powerful machines. Sometimes a large resource needs to be broken down into smaller types and share the burden to minimize the cost. When the machines that are appended to or removed from a pool of resources in a particular configuration are from the same type, the scaling is called homogeneous whereas it is called heterogeneous when machines are from different types. Scaling also can be upward when new resources (machines) are being added to the system or downward when some machines are being removed from it.

iGiraph [13] is an extension of Giraph [14] and the only Pregel-like framework that scales down homogeneously across public clouds while processing convergent algorithms (Fig. 3a). Basic iGiraph does not monitor network factors (except network traffic) or VM availability. Its decisions are made only based on the number of generated messages in the network, size of the partitions and memory. The idea is to merge small partitions from two different machines to make a bigger partition that fits into one machine; or migrate border vertices of a partition to another partition to reduce message passing ratio between VMs. Although the number of VMs will be reduced in this approach, the processing starts and finishes by using only one type of machines (e.g., large machines) during the entire operation.



(a) iGiraph homogeneous scaling policy



(b) Our proposed heterogeneous scaling policy

Fig. 3. Scaling policies for large-scale graph processing using convergent algorithms (a) basic iGiraph uses the same VM type during the entire processing, (b) iGiraph-heterogeneity-aware replaces VMs with smaller/less costly types as the processing progresses.

A more efficient alternative to this method is using a combination of different VM types. We observed that in many experiments, during final supersteps, the last VM (which is usually a large or medium size VM) is much larger than what is needed for the operation to be completed. It means that the user is paying for a large machine to accomplish a small task.

To address this issue, we propose a heterogeneous scaling in VM level which is specifically appropriate for processing large-scale graphs using convergent algorithms. In this method, as the processing continues, the system chooses suitable VM type based on the required capacity to host/process the rest of the graph, and partitions it accordingly. The new framework can be used easily as a cost-efficient graph processing service. Fig. 3 compares the original iGiraph homogeneous scaling policy versus our proposed policy.

Unlike default iGiraph, iGiraph-heterogeneity-aware scheduling algorithm measures and monitors more network factors such as bandwidth and CPU utilization in addition to the network traffic. Combining this with other information such as memory utilization, VM states, partitioning changes, vertices migration, and available VMs, provides a holistic view of the entire environment that the system is operating in it. This is required to estimate and optimize the cost of processing. Since every processing operation consists of several iterations (supersteps), in order to reduce the overall cost of the processing, the summation of the costs of iterations must be decreased. To achieve this, the cost of every iteration should be either equal to or less than the cost of its previous iteration.

$$C(S_{i+1}) \leq C(S_i). \tag{1}$$

According to Equation (1), new VMs can be added and replaced only if the cost of new configuration will be less than the cost of the current configuration. C(S) is the cost of the superstep. This decision will be made by the decision-making module (Section 2.2). This approach successfully deals with price heterogeneity of the cloud resources too as price is one of the variables in the equation. It ensures that not only smaller VMs are being used, but the monetary cost is being considered as well.

4 Dynamic Characteristic-Based Repartitioning

We discuss the smart VM monitoring and our proposed characteristic-based dynamic repartitioning approach.

4.1 Smart VM Monitoring

The first step towards a smart partitioning is smart monitoring. A large number of existing graph processing frameworks do not measure important environmental factors such as network metrics and VM properties. These factors have huge impacts on the system's performance in various manners. For example, monitoring network traffic can help to direct communication messages to the channels with less traffic to reduce latency, or monitoring available memory and price of VMs enables to select the right machines for hosting partitions in order to increase the performance and reduce the cost of processing. In addition, the knowledge that is achieved from monitoring these factors can be utilized in helping to design and develop a more efficient framework. Therefore, to better take advantage of aforementioned metrics, we have designed a smart monitoring center in the heart of our proposed system, the master machine. The reason for centralizing this information on the master is that all decisions can be made in one place which leads to more accurate decision-making process.

There are two types of information that are gathered by our proposed system: 1). The information that is generated during the processing which is very dynamic and can change over time (such as network traffic, remaining VM memory, etc.), and 2). The information that remains unchanged during the entire operation (such as VM price, VM total memory capacity, etc). At this stage, the information that is listed in the proposed monitoring system includes the network traffic, VM bandwidth, CPU utilization, available VM memory and partition sizes. All information will be stored on the master machine and updated at the end of each superstep after the synchronization barrier

occurrence. Having these, the algorithm is able to choose the best approach to repartition the graph continuously, scale up by using available heterogeneous resources on the cloud and distribute the new partitions accordingly. Meanwhile, selecting the appropriate set of information to be used at each step depends on the strategy that is defined in the repartitioning algorithm. This is usually dependent on the application itself. For example, if the application is communication-bound (which will be determined by the user at the beginning of the processing in the input command), the algorithm aims to reduce the network traffic by repartitioning the graph in a way that high-degree vertices will be migrated and placed near their neighbors. This way, a large number of messages will be passed in-memory and do not need to travel across the network. The communication will speed up as well by mapping new partitions and VMs based on their bandwidth. The strategy would change when the application is computation-bound. These situations have been investigated in [20]. In this paper, we use two convergent communication-bound algorithms: single source shortest path and connected components.

Different mechanisms have been implemented to measure various factors in the environment. As discussed in [13], to measure the network traffic, we calculate the number of messages that are passed between partitions in each iteration. This measurement also shows us which partitions contain more high-degree border vertices which will affect our decision-making strategy. Bandwidth and CPU utilization are two factors which were not measured in basic iGiraph. For measuring the bandwidth between each pair of machines, we use an end-to-end mechanism that is utilized in [21]. This factor is important because the bandwidth constantly changes in a cloud environment. Moreover, since we store one partition on each worker, this evaluation gives us the bandwidth between two partitions in the network which in turn can be used in the mapping operation. On the other side, we use Ganglia[1] monitoring tool to obtain CPU utilization and other network metrics. To have a more accurate measurement, the percentages of both CPU utilization and CPU idle time are measured. CPU idle time is for cases where a small piece of a job consumes a large part of the computation resources for a very short amount of time while they are free for the rest of the time [20]. However, in some cases, one small continuous task will be running on CPU for a long time. In this situation, the idle time is small while CPU utilization is also small. So, only if idle time is small and the CPU utilization is big, the VM *will not* be considered for migration or replacement. The system will consider a default threshold of 50 percent for both CPU utilization and idle time and selects the policy based on that. Nevertheless, the user can define the threshold for both variables manually too. We also calculate the available capacity of each machine by considering the correlation of the sizes of partitions and VMs. Additionally, to avoid making monitoring a bottleneck for the performance of the system, changeable information will be stored on workers until the synchronous barrier happens and final values will be sent to the master only once after every barrier signal.

Besides factors that are being persistently modified during the processing, there are constant factors such as VM properties, prices and types of machines that will not change. In fact, they are inherently part of the cloud environment. So, they will be stored at the beginning of the operation. The system also will know how many machines are available on the network and how much resources they can provide for the execution. The resource pool will be considered based on the maximum amount of resource requirements by users at the beginning of the processing which later will be optimized during the operation. Another important difference between iGiraph-heterogeneity-aware and basic iGiraph is that the latter is environment-agnostic and did not use any of this information for a better computation. All these information alongside the changeable metrics' information will be stored in a separate file on the master machine to be used in the partitioning algorithm.

4.2 Dynamic Repartitioning

To enable and improve the usage of heterogeneous resources, we have proposed a characteristic-based repartitioning method. "Characteristic-based repartitioning" here means that the system knows the characteristics of the resources and is aware of specific statistics (such as network metrics) by which new decisions can be made about partitioning the graph again in a dynamic manner. To achieve this, the algorithm contains two major steps: 1) prioritization step, and 2) mapping step. In the prioritization step, the algorithm prioritizes partitions and resources based on the application requirements before distributing partitions across the network. The mapping step is where the algorithm decides how to utilize the available resources.

As mentioned in Section 4.1, the static information about the available VMs and their types will be stored on the master. This information includes the price, the number of cores and memory capacity of each VM along with labeling VMs based on their size e.g., small, medium and large. The labeling mechanism increases the speed of algorithm when it is making decisions about where to place the new partitions (Without a labeling mechanism, the algorithm had to compare VM capacities to find out which type they are). In this paper, the system knows how many machines are available in the network (resource pool) and it will be given by a list of information before the processing starts. However, to start the processing, the initial number and the type of VMs will be given by the user. So, at this step, out of the resource pool, only a specific number of VMs will be used to start the processing with. This can be considered as a pre-processing operation. Also, in all our experiments in this paper, at the start of the processing, the graph will be partitioned randomly (based on vertices identifiers). The first iteration of processing (superstep 0) ends when the global synchronization barrier happens. At this point, the VM monitoring module collects the information (changeable information-Section 4.1) before the next superstep to use them for repartitioning purpose.

1. http://ganglia.sourceforge.net/

Algorithm 1. Characteristic-based Dynamic Re-partitioning

```
Get the information about available VMs in the network
Partition the graph randomly
Set PP = 0 for each partition and WP = 0 for each worker
For the rest of the computation do
  Calculate PP for each partition based on the number of messages that each partition receives
  Calculate WP for each worker using end-to-end mechanism
  If global synchronization happened then
    If Size(PartitionP1+PartitionP2)< = Size(MemoryOfSmall VM) then
      mergeIntoSmallVM(P1,P2)
      removeCurrentVM(P1,P2)
    elif Size(P1+P2)>Size(MemoryOfSmallVM) and Size(P1+P2)< = Size(MemoryOfCurrentVM) then
      mergeIntoCurrentVM(P1,P2)
      removeCurrentVM()
    elif Size(PartitionP1+Adjacentpartitions)< = Size(MemoryOf AdjacentVMs) then
      migrateIntoAdjacentVMs(P1)
      removeCurrentVM()
  Merge the partitions or migrate vertices if needed
  Set the priorities based on PP and WP
  Add/Remove VMs if needed
  Map partitions(based on PP) and workers(based on WP)
  If VoteToHalt() then
    Break
```

After superstep 0, the algorithm starts prioritizing partitions and VMs according to the changes that occurred in the first iteration. It also investigates any scaling possibility at the resource level for the next iteration. At this phase, each partition will be given a new label value called Partition Priority (PP) based on the number of messages they have received. The PP for the partition that has received the largest number of messages will be set to $0\,(\mathrm{PP}=0)$, the PP for the partition that has received the second largest number of messages through the network will be set to $1\,(\mathrm{PP}=1)$ and so on. When a partition receives more messages in comparison with other partitions, it means that it contains more high-degree border vertices. Therefore, since the aim is to move high-degree vertices closer to their adjacent vertices, this can be considered as a candidate for partition merge or vertex migration. With a similar mechanism, all worker machines that were used in the operation will be labeled by a Worker Priority (WP) label. Because *we are using communication-bound application in this paper* (computation-bound algorithms will be investigated in our future works), the prioritization of workers is based on their bandwidth (not CPU utilization) and available memory. So, the WP for the VM with the highest bandwidth will be set to $0\,(\mathrm{WP}=0)$, WP for the VM with the second highest bandwidth will be set to $1\,(\mathrm{WP}=1)$ and so on. If two partitions or two workers have the same value for prioritization, one of them will be given the higher priority randomly. After this phase, because we put one partition per VM, the partitions and VMs with the same priority number will be mapped to each other. This calculation is fast as all information is gathered during the iteration.

Although assigning priorities seems very straightforward, there are situations that need to be taken into consideration during the assignment process. Any partition merging or vertex migration could affect the prioritization. According to [20], the partition that has been given more migrated vertices gets the highest priority. In case that vertices are migrated across multiple partitions, the partition that has received more vertices will be set to the highest priority and so on. Meanwhile, the priorities of the new partitions that are formed by merging other partitions will be set to the highest priority among partitions before merging. For example, if partition P1 has higher priority than partition P2, after merging these two partitions $(\mathrm{P3}=\mathrm{P1}+\mathrm{P2})$, P3's priority will be set to the former priority of P1. Also in cases a large VM is splitting into smaller VMs or moving its entire assigned partition to a smaller type of VM, the priority of the new VM will be calculated the same way as a large VM (described in the previous paragraph). The priorities of all partitions and VMs are set to 0 at the beginning of the operation (before superstep 0 starts).

As mentioned above, the system selects VMs of the same type from the resource pool based on the user's requirement. For example, if the user sets the number of VMs to 16, and chooses the type medium, then 16 medium VMs will be allocated to the processing. Nevertheless, both the number and the type (size) of VMs will change during the execution due to partition merges or vertices' migrations. The aim of merging partitions or migrating vertices is to take high-degree vertices closer to their adjacent vertices. This, results in a significant reduction in cross-edges between machines which leads to less message transmission throughout the network. When the total number of messages that are transferred during the superstep $i+1$ is less than the total amount of the messages that were transferred during superstep i, there is a possibility for partition merge. So, as long as the number of messages is increasing, partitions cannot merge (e.g., SSSP). After each superstep, if the sum of the size of typical partition P1 and partition P2 is less than the memory capacity of a smaller VM type, then they will merge and partition $(\mathrm{P1}+\mathrm{P2})$ will be moved to the new small VM. If $(\mathrm{P1}+\mathrm{P2})$ is larger than the memory of small VM, but it can be fit into the memory of one of current VM types, they will merge into one VM and the other VM will be removed. If some partitions, that are neighbors of a very small partition (a partition that has occupied a tiny fraction of a VM memory), have enough space to host the vertices of the small partition without needing to employ a new VM, then all vertices of the small partition will be distributed among its adjacent partitions. So, there is no need to add a new machine. Algorithm 1 shows the characteristic-based dynamic repartitioning. In this algorithm, *CurrentVM* and *SmallVM* are two representatives of the current utilized VM and the smaller VM that partitions and vertices will be migrated or merged to, respectively. For example, if the *CurrentVM* is "Large" type, then *SmallVM* can be a "Medium" type and so on. As a result, this algorithm works for all other types of VMs.

5 Performance Evaluation

5.1 Experimental Setup

To evaluate the effectiveness of our framework and proposed algorithms, we utilized resources from Australian national Cloud Infrastructure (NECTAR) [22]. We utilize three different

TABLE 1
Some Typical Amazon VM Types

VM Type	#Cores	RAM	Disk (root/ephemeral)	Price/hour
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

VM types for our experiments based on NECTAR VM standard categorization: m2.large, m1.medium, and m1.small. Detailed characteristics of utilized VMs are shown in Table 1. 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 the 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 observed that regardless of which region the user chooses the VMs from, our approach reduces the monetary cost by the order of magnitude compared to other existing frameworks. We use iGiraph [13] (the extended version of Giraph [14]) 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 shortest path and connected components algorithms as two convergent graph algorithms for our experiments. They are good representatives of many other algorithms regarding their behavior. We also use three real-world datasets of different sizes: YouTube, Amazon, Pokec and Twitter [23] as shown in Table 2.

5.2 Evaluation Results

We have compared our system 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 [24]. We also compared the performance of basic iGiraph that scales out homogeneously with our proposed heterogeneous extension of iGiraph (i.e., iGiraph-Heterogeneity-aware). In this paper, the size of the messages in all experiments is equal, hence the relative cost of communication is independent of message size. Instead, the total number of messages that are transferring through the network has been calculated for cost. All experiments start with medium VMs as their workers.

TABLE 2
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
Twitter (WWW)	41,652,230	1,468,365,182

The first group of experiments is conducted for processing various datasets using shortest path algorithm. As shown in the above figures, the *blue* area demonstrates the number of VMs that are being used by Giraph which is correlated with the cost of the operation. So, Giraph is the most costly solution among all the systems because it uses the same number of machines during the entire operation. Many existing distributed graph processing frameworks never reduce the number of resources during the processing. On the other hand, as the operation is being progressed, more vertices become processed. So, iGiraph removes unnecessary VMs and distributes the rest of partitions on the remaining machines. The *red* area shows that iGiraph is reducing the number of utilized VMs. This declines the cost significantly on a public cloud compared to Giraph. However, basic iGiraph only utilizes homogeneous machines. It means that if the processing has been started with medium size VMs, it will be ended with medium size VMs as well despite the VM reduction. In this case, although smaller partitions tend to be merged to create a bigger partition to optimize VM utilization, there are always situations where a tiny partition (that has occupied a large VM and all its capacity) cannot be merged or migrated. To address this issue, iGiraph-Heterogeneity-aware replaces current VMs by smaller ones. It has been shown that iGiraph-Heterogeneity-aware provides more than 20 percent cost reduction compared to original iGiraph (the *green* area). The majority of this cost saving is due to removing unnecessary VMs from the list of active VMs or replacing them with smaller types. We consider a VM as a package of resources including computation and storage resources. Hence, removing or downsizing VMs leads to significant cost savings. All VM types are correlated with particular prices as shown in Table 1. Therefore, as it can be seen in the diagrams (Figs. 4, 5, 6, 7) for both basic iGiraph and

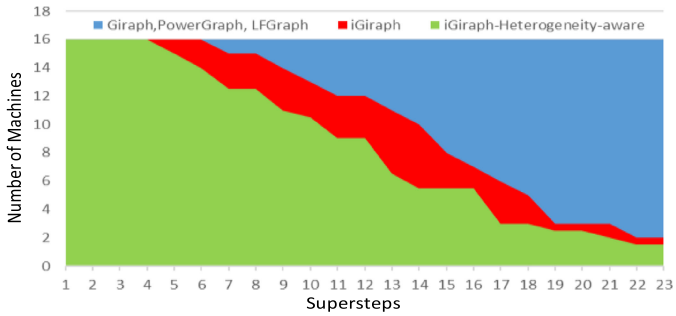


Fig. 4. Number of machines during processing shortest path on Amazon.

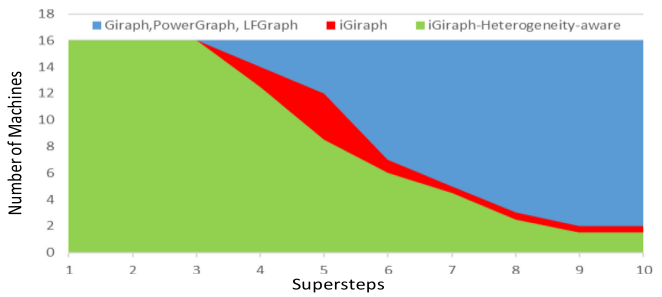


Fig. 5. Number of machines during processing shortest path on YouTube.

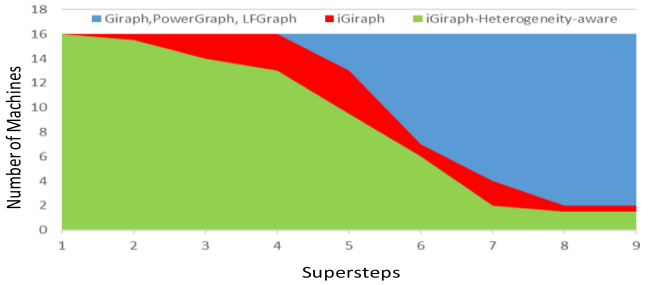


Fig. 6. Number of machines during processing shortest path on Pokec.

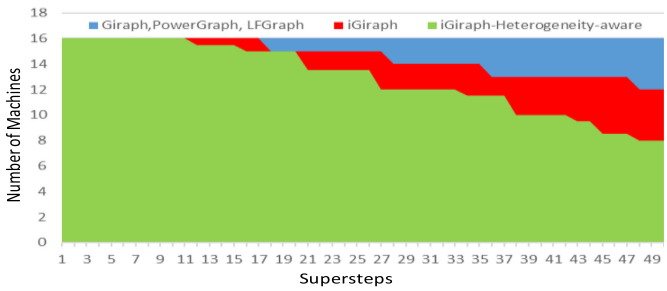


Fig. 7. Number of machines during processing shortest path on Twitter for the first 50 supersteps.

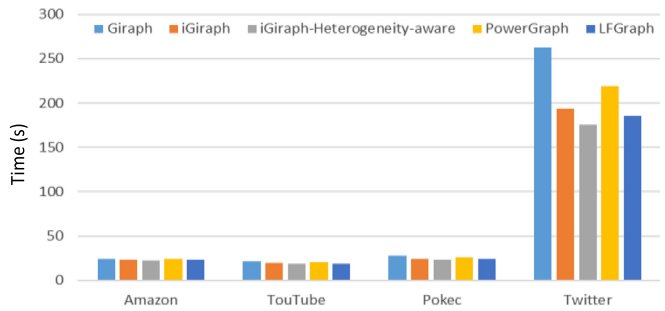


Fig. 8. Total execution time for processing shortest path algorithm on various datasets.

iGiraph-heterogeneity-aware, the cost of each superstep is either *equal to* or *less than* the cost of its previous superstep due to VM elimination or replacement (Section 3). However, iGiraph-heterogeneity-aware achieves better results by taking advantage of resource heterogeneity. As shown in Figs. 8 and 16, iGiraph-Heterogeneity-aware even completes the processing faster than other frameworks due to its new partitioning approach which distributes partitions based on their characteristics and the properties of available machines. Figs. 9, 10, and 11 show the number and types of machines in each iteration. These results that have been generated by putting together the average outcomes of 45 runs demonstrate the behavior of our proposed solution and how it removes or replaces VMs during the processing.

As a result, the total cost of the processing is dependent on *the number* and *the time* (duration) that a particular type of VM is being utilized during the operation. This is shown in Equation (2), where $\mathrm{C(VM_i)}$ is the price of the VM and $\mathrm{T(VM_i)}$ is the time that within the VM is used. The equation calculates the cost for all VMs (n) during the entire processing iterations (m).

$$\mathrm{Cost}_{final} = \sum_{j=0}^{m} \sum_{i=1}^{n} (C(VM_i) \times T(VM_i)). \tag{2}$$

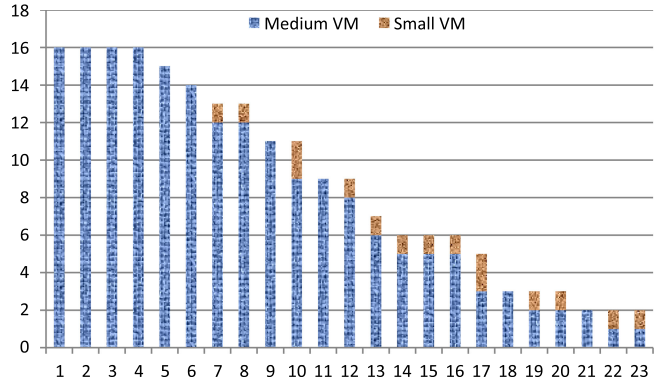


Fig. 9. Resource modification during processing shortest path on Amazon.

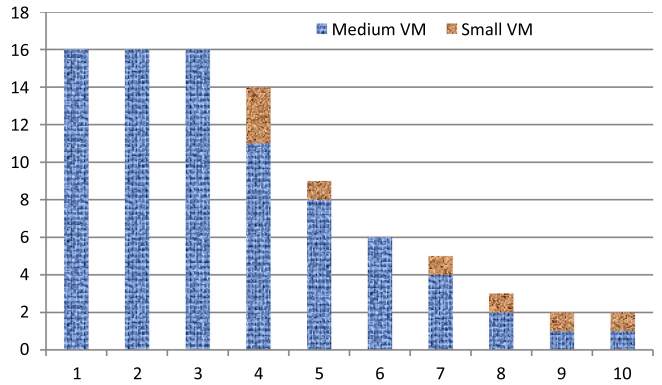


Fig. 10. Resource modification during processing shortest path on YouTube.

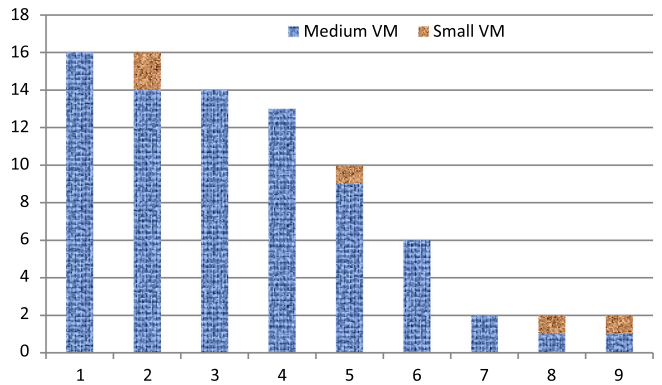


Fig. 11. Resource modification during processing shortest path on Pokec.

Although data transfer affects the ultimate cost calculation, we did not consider that in this equation, but we will take it into consideration for our future works. Table 3 shows the cost comparison for different datasets for shortest path algorithm on each framework.

We carried out similar experiments on connected component algorithm using the same datasets. Final results are showing significant improvements and cost saving compared to Giraph (Figs. 12, 13, 14, and 15). Also, our proposed partitioning method for iGiraph-Heterogeneity-aware makes it outperform basic iGiraph up to 20 percent. Table 4 shows the cost comparison for different datasets for connected components algorithm on each framework.

TABLE 3
Processing Cost for SSSP on Different Frameworks

Dataset	Giraph	PowerGraph	LFGraph	iGiraph	iGiraph-heterogeneity-aware
Amazon	$0.0133	$0.0118	$0.0107	$0.0082	$0.0064
YouTube	$0.0117	$0.0114	$0.0098	$0.0070	$0.0045
Pokec	$0.0149	$0.0143	$0.0121	$0.0095	$0.0056
Twitter	$8.84	$7.48	$5.61	$4.92	$3.303

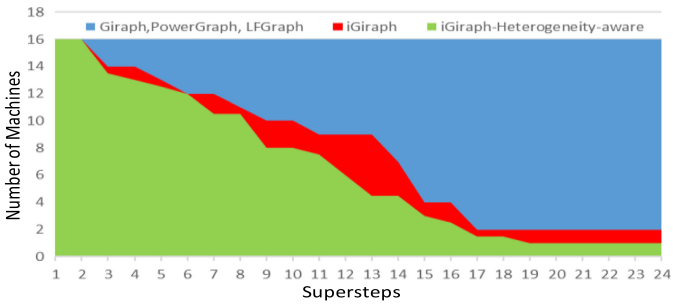


Fig. 12. Number of machines during processing connected components on Amazon.

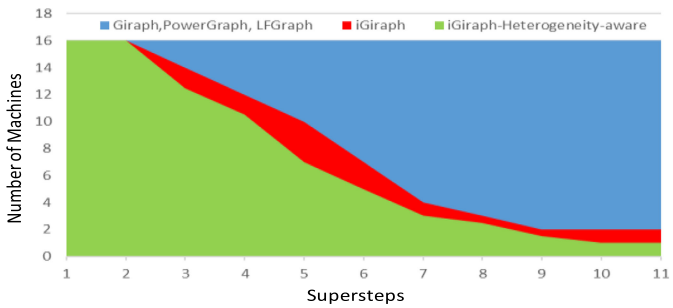


Fig. 13. Number of machines during processing connected components on YouTube.

Table 6 (Section 6) demonstrates different characteristics of these systems and the newly implemented features.

5.3 Discussion and Analysis

The results demonstrate that our iGiraph-Heterogeneity-aware approach significantly reduces the monetary cost of the operation compared to other frameworks while improving the performance. These improvements are obtained through a series of interconnected/interoperable operations. In our system, each component has a critical role and each one of them contributes to overall system performance. For example, dynamic repartitioning heuristics have higher overhead than a simple random partitioning approach. This is due to the time and resources that are required for making smarter decisions or merging/migrating partitions across the system. However, this reduces drastically for later itterations (as observed in Fig. 17).

According to Fig. 17, the average time overhead for processing connected component algorithm on Amazon dataset using Giraph framework is almost intact during the entire operation due to its simple random partitioning strategy. On the other hand, the average time overhead varies remarkably while processing by iGiraph-Heterogeneity-aware. At the start of the operation, the number of partitions that need to be

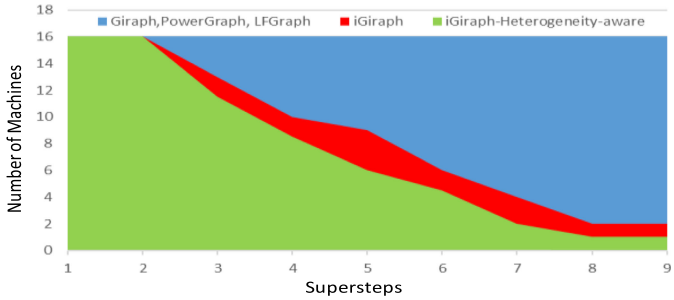


Fig. 14. Number of machines during processing connected components on Pokec.

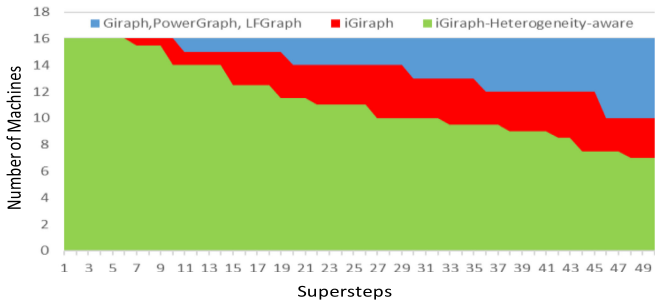


Fig. 15. Number of machines during processing connected components on Twitter for the first 50 supersteps.

TABLE 4
Processing Cost for CC on DIfferent Frameworks

Dataset	Giraph	PowerGraph	LFGraph	iGiraph	iGiraph-heterogeneity-aware
Amazon	$0.0138	$0.0116	$0.0110	$0.0062	$0.0051
YouTube	$0.0128	$0.0109	$0.0087	$0.0065	$0.0053
Pokec	$0.0160	$0.0146	$0.0135	$0.0086	$0.0072
Twitter	$8.5	$7.99	$5.78	$4.07	$3.43

merged or vertices that need to be migrated is the most. Therefore, it takes more time to move partitions and vertices across the workers. However, as the operation progresses, processed vertices and edges will be removed from the memory which results in quicker repartitioning among the remaining nodes.

Our observations showed that although dynamic repartitioning in iGiraph-Heterogeneity-aware creates larger time overhead compared to random partitioning in Giraph, it contains only *5-8* percent of the entire runtime of the operation. This overhead does not noticeably affect the system performance because of other optimizations that increase in-memory computation and reduce other overheads such as network and memory overheads (as follows).

Another important overhead in a graph processing system is caused due to the network traffic. Since iGiraph-Heterogeneity-aware is using iGiraph as its core framework, its dynamic repartitioning distinguishes between internal and border vertices [14]. In this method high degree border vertices will be placed with their adjacent neighbors in the same partition. Therefore, it minimizes the number of cross-edges between partitions (workers), which in turn minimizes message transmission across the network as shown in Fig. 18. Table 5 compares the maximum number of messages that needed to be passed through the network in the peak superstep in both Giraph and iGiraph-Heterogeneity-aware

TABLE 5
Maximum Number of Messages that Needed to Be Transferred Through the Network in the Peak Superstep While Processing SSSP

	Amazon	YouTube	Pokec
iGiraph	643,237	215,754	1,259,613
iGiraph -Heterogeneity-aware	500,964	84,382	524,496

while processing SSSP. Our iGiraph-Heterogeneity-aware approach significantly reduces the network overhead by utilizing the new dynamic repartitioning method proposed in this paper. It also speeds up the process by mapping partitions and traffic based on the available bandwidth.

Complexity Analysis. We analyzed the time complexity of our dynamic repartitioning algorithm. This algorithm relies on the number of supersteps (N) when N can fluctuate based on the application and the number of nodes in the graph. Moreover, partition prioritization (PP) and worker prioritization (WP) that are required to be calculated in every superstep are affecting the algorithm too. Hence, the worst case complexity can be $O(N(PP + WP))$ where both PP and WP are dependent to the number of machines (m). Whereas, the best case complexity can be $O(N(log\, m))$. Giraph complexity is $O(N(n))$ where n is the number of vertices. It shows that our approach is better than default Giraph in terms of complexity.

The benefits/advantages of using our proposed approach for processing large-scale graphs include: 1) reducing memory overhead by removing *inactive* vertices from the memory, 2) enhancing computation speed by increasing in-memory computations, 3) decreasing network traffic by reducing cross-edge connections between partitions, 4), improving partitioning by using a smart characteristic-based dynamic repartitioning method, and 5) significant monetary cost reduction by improving elasticity and utilizing heterogeneous resources. All these benefits will be effective even if homogeneous resources are used. We have investigated this case in our previous works [13], [20].

6 Related Work

We investigated various factors such as scalability, dynamic partitioning, which are studied individually by other works. Scalability is a major concern in many systems. Each work has addressed scalability issue in a different way. Pregel-like frameworks such as Giraph [14], GPS [25] and GiraphX [26] along with some non-Pregel-like frameworks such as Trinity [27], Presto [28] and PowerSwitch [29] argue that a distributed architecture can provide better scalability. The system can access as many resources as needed to operate and increase the performance.

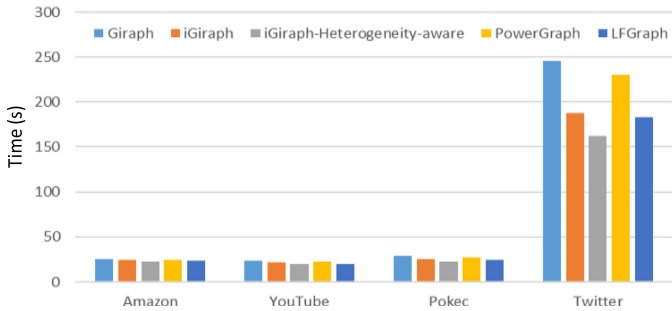


Fig. 16. Total execution time for processing connected components algorithm on various datasets.

However, these systems use other optimizations to deliver better performance as well. GPS [25], for instance, introduced a dynamic repartitioning approach by which the partitions will be distributed again among faster workers who complete their jobs before other workers inside each iteration. This keeps all workers busy all the time during the processing and faster workers do not need to wait until slower workers finish their jobs. This approach has become possible by utilizing an asynchronous implementation of partition distribution inside supersteps. Another system called PowerSwitch [29], improves the performance of the system by effectively predicting the proper heuristic for each step. It then switches between synchronous and asynchronous execution states if required. Not only iGiraph-heterogeneity-aware provides scalability over heterogeneous resources, but also provides elasticity as it provisions in an autonomic way. Hence, the current demand and resource consumption matches at any given time, according to [30].

Uta et al. [31] have studied elasticity in graph processing and proposed a benchmarking framework called JoyGraph for elastic graph processing. They found that graph workloads are sensitive to data migration when employing or releasing resources. JoyGraph is using more metrics to provide accurate elasticity. Although the paper is using different application and datasets, it shows that increasing or

TABLE 6
A Comparison with the Most Relevant Works

System	Implemented Environment	Partitioning Method	Network-aware	Resource-aware	Resource Configuration
Pregel	HPC	Static	No	No	Homogeneous
Giraph	HPC	Static	No	No	Homogeneous
PowerGraph	HPC	Static	No	No	Homogeneous
GPS	HPC	Dynamic	No	No	Homogeneous
Pregel.Net	Cloud	Dynamic	No	No	Homogeneous
Surfer	Cloud	Dynamic	No	No	Homogeneous
iGiraph	Cloud	Dynamic	No	No	Homogeneous
iGiraph- Network-aware	Cloud	Dynamic	Yes	No	Homogeneous
iGiraph -Heterogeneity-aware (Our Work)	Cloud	Dynamic	Yes	Yes	Heterogeneous

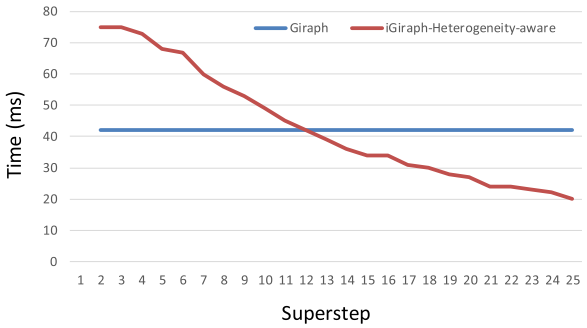


Fig. 17. Average time overhead for random partitioning in Giraph versus smart dynamic repartitioning in iGiraph-Heterogeneity-aware during processing connected components on Amazon.

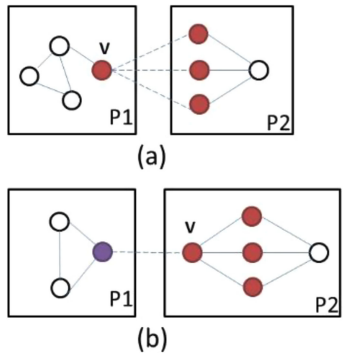


Fig. 18. The role of high degree border vertices in reducing network traffic by reducing the number of cross-edges between partitions (a) before repartitioning, (b) after repartitioning [14].

decreasing the number of machines creates significant overhead on CPU and memory. Unlike JoyGraph, we have proposed a novel processing and dynamic repartitioning strategy in our work (iGiraph-Heterogeneity-aware) by which all processed vertices are being removed from the memory. This reduces overheads drastically. Another difference is that JoyGraph is not investigating the monetary cost of processing large-scale graphs on public cloud environments. Financial cost is a factor that has critical impact on user's decision while selecting a particular service [32].

Although distributed graph processing frameworks are developed based on commodity cluster environment properties, the situation is different on cloud environments, particularly public clouds. There are different pricing models available on clouds and users have to pay for the resources that they use. They can also use a pay-as-you-go billing model. In such environment, it is important to reduce the cost of utilizing resources as much as possible. Many graph frameworks tried to decrease the processing cost by providing faster execution to reduce the total runtime so that they can release the resources quicker to pay less. For example, Surfer [12] develops a bandwidth-aware repartitioning mechanism by which partitions are being placed on workers based on their bandwidth. While only few graph processing frameworks are developed to specifically operate in cloud environments, iGiraph [13], which we used in this paper as one of the benchmarks, is using a different strategy. Using its novel dynamic repartitioning approach, iGiraph eliminates unnecessary resources during the processing period while operating on convergent algorithms which leads to significant cost saving compared to other frameworks. Although systems such as GPS [25] and Mizan [33] implement dynamic repartitioning and vertex migration, they do not scale across VMs. It has been shown that iGiraph outperforms frameworks such as popular Giraph while operating on non-convergent algorithms such as PageRank. It declines the number of messages that are passing through the network and executes faster.

In addition to distributed systems, many graph processing frameworks are developed at the scale of a single machine [34], [35], [36], [37], [38], [39], [40]. Since high capacity memories and disks have become unprecedentedly available on single PCs, these frameworks implement mechanisms for processing large-scale graphs without the hassle of distributing computations on several machines. Solid state drives (SSD) that provide higher speed data access compared to hard disk drives (HDD) have made the idea of processing on a single server even more promising. GraphChi [41] is a pioneer in this category. It is a vertex-centric framework that offers a parallel sliding window (PSW). PSW is an asynchronous computing method for leveraging external memory (disk) and is suitable for sparse graphs. Using this technique, GraphChi only requires to transmit a small number of disk-blocks sequentially. PSW's input is a subset of the graph that is loaded from the disk. It then updates the values on vertices and edges and finally writes the new values back on the disk. Systems such as FlashGraph [42] are specifically designed to perform on SSDs. In FlashGraph, I/O requests will be merged cautiously to improve the throughput and decrease CPU overhead.

Finally, dynamic partitioning and network factors are the last two aspects of our work in this paper. Many graph processing frameworks partition the graph at the start of operation and never change it again until the end of processing. Nonetheless, repartitioning the graph during the operation is becoming more common as the system/user can change the partitions' properties at any time to improve the performance. According to [25], a dynamic repartitioning function should be able to answer three main questions: 1) Which vertices must be reassigned?, 2) How and when to move the reassigned vertices to their new machine?, and 3) How to place the reassigned vertices? A framework like GPS [25] repartitions the graph based on using high-degree vertices while LogGP [43] does so based on analyzing and reusing "the historical statistical information" to rectify the partitioning outcomes. Other systems such as Mizan [33], XPregel [44] and xDGP [45] also have used various approaches to partition graphs dynamically. Network is another important factor that affects partitioning and the processing but it is studied less in the context of graph processing. Frank McSherry[2] has investigated the impact of fast networks on graph analytics after an NSDI paper [46] claimed that the network speed does not make a huge change in the processing performance. He showed that under some general conditions, a faster network can improve the operation's efficiency. Our earlier work [20] has used network factors such as bandwidth, traffic, and CPU utilization to partition the graph dynamically. It shows that using a suitable combination of

2. http://www.frankmcsherry.org/pagerank/distributed/performance/2015/07/08/pagerank.html

factors will make the system to outperform frameworks such as Giraph. A detailed comparison of many existing graph processing frameworks are discussed in [42], [47].

7 Conclusions and Future Work

In this paper, a new auto-scaling algorithm is proposed and is plugged into the iGiraph framework to reduce the monetary cost of graph processing on public clouds. To achieve this, heterogeneous resources have been considered alongside horizontal scaling policy. Also, a new characteristic-based dynamic repartitioning approach is introduced which distributes new partitions on heterogeneous resources. The experiments show that the new mechanism that is called iGiraph-heterogeneity-aware reduces the cost of processing significantly and outperforms frameworks such as original iGiraph and the famous Giraph. To the best of our knowledge, iGiraph-heterogeneity-aware method is the first implementation in a graph processing framework for supporting horizontally scalability by using heterogeneous resources in a real cloud environment.

As part of future work, we plan to make the system completely environment agnostic, which means it will dynamically identify characteristics and capabilities of the resources in the network. We plan to consider the large-scale graph processing, as a service on cloud platforms. Therefore, we will investigate the role of SLA (service level agreement) and how the quality of service should be applied in such environment. We also plan to investigate the impact of starting the operation with other partitioning approaches such as METIS and how they can improve the performance even more. In addition, we will investigate the possibility of starting the processing by utilizing heterogeneous combination of VMs instead of starting by VMs of the same type. We will also explore how to deal with computation-intensive graph applications and non-convergent applications such as PageRank for executing them on elastic Clouds.

Acknowledgments

This work is partially supported by ARC Future Fellowship and ARC Discovery Project grants. We thank NECTAR for providing access to Australian cloud infrastructure.

References

[1] L. Belli, S. Cirani, G. Ferrari, L. Melegari, and M. Picone, "A graph-based cloud architecture for big stream real-time applications in the internet of things," in *Proc. Adv. Serv.-Oriented Cloud Comput.*, 2014, pp. 91–105.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Des. Implementation*, 2004, pp. 137–150.

[3] F. N. Afrati, A. Das Sarma, S. Salihoglu, and J. D. Ullman, "Vision paper: Towards an understanding of the limits of map-reduce computation," in *Proc. Cloud Futures Workshop*, 2012, pp. 1–5.

[4] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *Proc. 26th Conf. Uncertainty Artif. Intell.*, 2010, pp. 1–11.

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.

[7] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 2.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," in *Proc. 7th Int. World Wide Web Conf.*, 1998, pp. 161–172.

[9] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, 1979.

[10] M. Redekopp, Y. Simmhan, and V. K. Prasan, "Optimizations and analysis of BSP graph processing models on public clouds," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 203–214.

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

[12] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1123–1126.

[13] S. Heidari, R. N. Calheiros, and R. Buyya, "iGiraph: A cost-efficient framework for processing large-scale graphs on public clouds," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 301–310.

[14] "Apache Giraph!," Apache, [Online]. Available: https://giraph.apache.org/. [Accessed on May 31, 2019].

[15] P. Roy, "A new memetic algorithm with GA crossover technique to solve single source shortest path (SSSP) problem," in *Proc. Annu. IEEE India Conf.*, 2014, pp. 1–5.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[17] Y. Tian, A. Balmin, S. Andreas Corsten, S. Tatikond, and J. McPherson, "From "Think like a vertex" to "Think like a graph"," *VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[18] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big (ger) graph analytics on a dataflow engine," *VLDB Endowment*, vol. 8, no. 2, pp. 161–172, 2014.

[19] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.

[20] S. Heidari and R. Buyya, "Algorithms for cost-efficient network-aware scheduling of large-scale graphs in cloud computing environments," *Softw.: Practice Experiences*, vol. 48, no. 12, pp. 2174–2192, 2018.

[21] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3079–3089, Nov. 2015.

[22] "NECTAR Cloud," [Online]. Available: http://nectar.org.au/research-cloud/. [Accessed on June 1, 2019].

[23] J. Kunegis, "KONECT - the koblenz network collection," in *Proc. Int. Web Observatory Workshop*, 2013, pp. 1343–1350.

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

[25] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 25th Int. Conf. Sci. Statist. Database Manage.*, 2013, Article no. 22.

[26] S. Tasci and M. Demirbas, "Giraphx: Parallel yet serializable large-scale graph processing," in *Proc. 19th Int. Conf. Parallel Process.*, 2013, pp. 458–469.

[27] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 505–516.

[28] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 197–210.

[29] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to fuse for distributed graph-parallel computation," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 194–204.

[30] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. 10th Int. Conf. Auton. Comput.*, 2013, pp. 23–27.

[31] A. Uta, S. Au, A. Ilyushkin, and A. Iosup, "Elasticity in graph analytics? A benchmarking framework for elastic graph processing," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 381–391.

[32] S. Heidari and R. Buyya, "Quality of Service (QoS)-driven resource provisioning for large-scalegraph processing in cloud computing environments: GraphProcessing-as-a-Service (GPaaS)," *Future Generation Comput. Syst.*, vol. 96, pp. 490–501, 2019.
[33] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoo, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.
[34] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 77–85.
[35] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 456–471.
[36] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 135–146.
[37] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proc. 32nd IEEE Int. Conf. Data Eng.*, 2016, pp. 409–420.
[38] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 195–207.
[39] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "MOSAIC: Processing a trillion-edge graph on a single machine," in *Proc. Eur. Conf. Comput. Syst.*, 2017, pp. 527–543.
[40] P. Sun, Y. Wen, T. Nguyen Binh Duong, and X. Xiao, "GraphMP: An efficient semi-external-memory big graph processing system on a single machine," in *Proc. IEEE 23rd Int. Conf. Parallel Distrib. Syst.*, 2017, pp. 276–283.
[41] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[42] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.
[43] N. Xu, L. Chen and B. Cui, "LogGP: A log-based dynamic graph partitioning method," *VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.
[44] N. Thien Bao and T. Suzumura, "Towards highly scalable pregel-based graph processing platform with x10," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 501–508.
[45] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xDGP: A dynamic graph processing system with adaptive partitioning," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–13.
[46] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 293–307.
[47] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Survey*, vol. 51, no. 3, 2018, Article no. 60.

Safiollah Heidari working toward the PhD degree in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems (CIS), the University of Melbourne, Australia. 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. He is a member of the IEEE.

Rajkumar Buyya is a Redmond Barry distinguished professor and the director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He has authored more than 625 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index = 127, g-index = 280, 84,900+ citations). He is recognized as a "Web of Science Highly Cited Researcher" in 2016 and 2017 by Thomson Reuters, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to Cloud Computing. He is a fellow of the IEEE.