Cost-efficient Resource Provisioning for Large-scale Graph Processing Systems in Cloud Computing Environments

Safiollah Heidari

Submitted in total fulfillment of the requirements of the degree of
Doctor of Philosophy

May 2018

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

Cost-efficient Resource Provisioning for Graph Processing Systems in Cloud Computing Environments

Safiollah Heidari
Principal Supervisor: Prof. Rajkumar Buyya

Abstract

A large amount of data that is being generated on Internet every day is in the form of graphs. Many services and applications namely as social networks, Internet of Things (IoT), mobile applications, business applications, etc. in which every data entity can be considered as a vertex and the relationships between entities shape the edges of a graph, are in this category. Since 2010, exclusive large-scale graph processing frameworks are being developed to overcome the inefficiency of traditional processing solutions such as MapReduce. However, most frameworks are designed to be employed on high performance computing (HPC) clusters which are only available to whom can afford such infrastructure.

Cloud computing is a new computing paradigm that offers unprecedented features such as scalability, elasticity and pay-as-you-go billing model and is accessible to everyone. Nevertheless, the advantages that cloud computing can bring to the architecture of large-scale graph processing systems are less studied.

Resource provisioning and management is a critical part of any processing system in cloud environments. To provide the optimized amount of resources for a particular operation, several factors such as monetary cost, throughput, scalability, network performance, etc. can be taken into consideration.

In this thesis, we investigate and propose novel solutions and algorithms for cost-efficient resource provisioning for large-scale graph processing systems. The outcome is a series of research works that increase the performance of such processing by making it aware of the operating environment while decreasing the dollar cost significantly. We have particularly made the following contributions:

1. We introduced iGiraph, a cost-efficient framework for processing large-scale graphs on public clouds. iGiraph also provides a new graph algorithm categorization and processes the graph accordingly.
2. To demonstrate the impact of network on the processing in cloud environment, we developed two network-aware algorithms that utilize network factors such as traffic, bandwidth and also the computation power.
3. We developed an auto-scaling technique to take advantage of resource heterogeneity on clouds.
4. We introduced a large-scale graph processing service for clouds where we consider the service level agreement (SLA) requirements in the operations. The service can handle multiple processing requests by its new prioritization and provisioning approach.

Declaration

This is to certify that

1. The thesis comprises only my original work towards the PhD,
2. Due acknowledgement has been made in the text to all other material used,
3. The thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Safiollah Heidari, 20 May 2018

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapter 2-6 and are based on the following publications:

- **Safiollah Heidari**, Yogesh Simmhan, Rodrigo N. Calheiros and Rajkumar Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges", *ACM Computing Surveys*, vol. 51, Issue. 3, No. 60, ACM Press, New York, USA, 2018.
- **Safiollah Heidari**, Rodrigo N. Calheiros and Rajkumar Buyya, "iGiraph: A Cost-efficient Framework for Processing Large-scale Graphs on Public Clouds", in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016)*, Cartagena, Colombia, Pages 301-310, 2016.
- **Safiollah Heidari** and Rajkummar Buyya, "Cost-efficient and Network-aware Dynamic Repartitioning-based Algorithms for Scheduling Large-scale Graphs in Cloud Computing Environments", *Software: Practice and Experience (SPE)*, vol. 48, Issue 12, Wiley & Sons, 2018.
- **Safiollah heidri** and Rajkumar Buyya, "A cost-efficient Auto-scaling Algorithm for Large-scale graph processing in Cloud Environments with Heterogeneous Resources", *IEEE Transactions on Software Engineering (TSE)*, 2018 (Under review).
- **Safiollah Heidari** and Rajkumar Buyya, "Quality of service (QoS)-driven Resource Provisioning for Large-scale Graph Processing in Cloud Computing Environments: Graph Processing-as-a-Service (GPaaS)", *Future Generation Computer Systems (FGCS)*, 2018 (Second Revision: Minor Revision).

Acknowledgements

PhD is an exciting journey by which I obtained many experiences in my academic and personal life. It would not have happened without endless help from people around me. First and foremost, I would like to thank my supervisor, Professor Rajkumar Buyya, who has offered me the opportunity to undertake a PhD, and provided with insightful guidance, continuous support, and invaluable advice throughout my PhD journey.

I would like to express my deepest appreciation and gratitude to my co-supervisor, Dr. Benjamin Rubinstein, whose knowledge and professional advices helped me to enjoy this journey more. I would like to sincerely thank Dr. Rodrigo N. Calheiros for being my co-supervisor while working at the University of Melbourne and also my external supervisor Dr. Yogesh Simmhan for their guidance on my research. Beside my supervisors, I woild like to thank the chair of my committee, Prof. Rui Zhang for his kindness and advice to help me progress along my PhD candidature.

I want to take the opportunity to thank all the past and present members of CLOUDS laboratory at the University of Melbourne: Dr. Mohsen Amini Salehi, Dr. Amir Vahid Dastjerdi, Dr. Adel Nadjaran Toosi, Dr. Deepak Poola, Dr. Nikolay Grozev, Dr. Atefeh Khosravi, Dr. Sareh Fotuhi, Dr. Maria Rodriguez, Dr. Chenhao Qu, Dr. Yaser Mansuri, Dr. Bowen Zhou, Dr. Jungmin Son, Dr. Xunyun Liu, Dr. Sukhpal Singh Gill, Dr. Hasanul Ferdaus, Diana Barreto, Minxian Xu, Sara Kardani Moghaddam, Caesar Wu, Muhammad H. Hilman, Redowan Mahmud, Muhammed Tawfiqul Islam, Shashikant Ilager, TianZhang He, Artur Pilimon and Arash Shaghaghi for their help and friendships.

I thank the University of Melbourne for the scholarship, research training support and facilities to let me pursue my PhD degree. My sincere thank goes to the staff at the School including Rhonda Smithies, Julie Ireland and Madalain Dolic for their support. I am also grateful to NECTAR Cloud for providing me with appropriate infrastructure and facilities for my research.

Last but not least, I would like to thank my parents, my sister and my brothers for their unconditional and loving support and encouragement. I am very fortunate that I have them in my life and nothing is greater than having an amazing family.

Safiollah Heidari
Melbourne, Australia
May 2018

Contents

1. Introduction 1
1.1 Challenges in large-scale graph processing on cloud environments 3
1.1.1 Cost Limitations and Models 4
1.1.2 Scalability and Communication Models 4
1.1.3 Standard Entry and I/O Issues 5
1.1.4 Network Issues and Limitations 5
1.1.5 Resource Provisioning 5
1.2 Research Problems and Objectives 6
1.3 Evaluation Methodology 8
1.4 Thesis Contribution 9
1.5 Thesis Organization 10
2 Taxonomy and Survey of Graph Processing Systems 14
2.1 Introduction 15
2.2 Background 16
2.2.1 Overall Scheme of Graph Processing 19
2.2.2 Large Graph-Oriented Applications 21
2.2.3 Algorithms in Graph Processing Studies and Experiments 23
2.3 Graph Programming Model 26
2.3.1 Graph Processing System Architectures 26
2.3.2 Graph Processing Frameworks 31
2.3.3 Distributed Coordination 38
2.3.4 Computational Models 43

2.4 Runtime Aspects of Graph Frameworks....47
2.4.1 Partitioning....47
2.4.2 Communication Models....52
2.4.3 Storage View....57
2.4.4 Fault Tolerance....60
2.4.5 Scheduling....62
2.5 Graph Databases....64
2.6 System Classification And Gap Analysys....67
2.7 Different Viewpoints On Categorization Of Graph Processing Systems....69
2.8 Summary....71
3 iGiraph: A Cost-efficient Graph Processing Framework....73
3.1 Introduction....74
3.2 Background....77
3.2.1 Giraph....77
3.2.2 Bulk Synchronous Parallel Model....79
3.2.3 Internal Vertices and Border Vertices....80
3.2.4 Graph Algorithms....80
3.2.5 Graph Processing Challenges on Clouds....82
3.3 iGiraph....83
3.3.1 Motivation....83
3.3.2 iGiraph's Dynamic Re-partitioning Approach....84
3.4 iGiraph Implementation....87
3.5 Performance Evaluation....88
3.5.1 Experimental Setup....88
3.5.2 Evaluation and Results....89
3.6 Related Work....96
3.7 Summary....98
4 Network-aware Dynamic Repartitioning for Scheduling Large-scale Graphs....101
4.1 Introduction....102
4.2 Related Work....105
4.3 Processing Environment Categorization and Graph Applications....108

4.4 Bandwidth-and-Traffic-aware Graph Scheduling Algorithm with Dynamic Re-partitioning 110
4.5 Computation-aware Graph Scheduling Algorithm with Dynamic Re-partitioning 114
4.5.1 Complexity Analysis 117
4.6 System Design and Implementation 118
4.6.1 Bandwidth Measurement 118
4.6.2 Traffic Measurement 119
4.6.3 CPU Measurement 119
4.6.4 Policy Selector 119
4.6.5 Network KPI Aggregator 119
4.6.6 Re-partitioner 120
4.7 Performance Evaluation 120
4.7.1 Experimental Setup 120
4.7.2 Results 121
4.7.3 Discussion 127
4.8 Summary 128
5 Auto-scaling Algorithm for Graph Processing with Heterogeneous Resources 131
5.1 Introduction 132
5.2 Graph Applications And Auto-Scaling Architecture 135
5.2.1 Applications 135
5.2.2 Proposed Auto-Scaling System Architecture 137
5.3 Horizontal Scaling (Step Scaling) 139
5.4 Dynamic Characteristic-Based Repartitioning 142
5.4.1 Smart VM Monitoring 142
5.4.2 Dynamic Repartitioning 145
5.5 Performance Evaluation 148
5.5.1 Experimental Setup 148
5.5.2 Evaluation and Results 149
5.6 Related Work 157
5.7 Summary 160

6 Graph Processing-as-a-Service 162
6.1 Introduction 163
6.2 Related Work 166
6.3 Overview of the Proposed Solution 170
6.3.1 Users 171
6.3.2 Repositories 172
6.3.3 Priority Queue 172
6.3.4 Monitoring Module 173
6.3.5 Management Module 174
6.3.6 Partitioning Module 176
6.3.7 Computation Module 177
6.4 Dynamic Scalable Resource Provisioning 178
6.5 Performance Evaluation 181
6.5.1 Experimental Setup 181
6.5.2 Evaluation and Results 181
6.6 Sumary 186
7 Conclusions and Future Directions 189
7.1 Conclusions and Discussion 189
7.2 Future Directions 193
7.2.1 Incremental Processing Models 194
7.2.2 Complex Workflows 195
7.2.3 Graph Databases 196
7.2.4 Cloud Features and Cost Models 197
7.2.5 Network Optimizations 199
7.2.6 Graph Compression 200
7.2.7 Energy-efficient Resource Allocation 201
7.2.8 Other Improvements 202
7.3 Final Remarks 203
BIBLIOGRAPHY 205

List of Figures

Figure 1-1Graphs are everywhere! 2
Figure 1-2 The thesis organization 11
Figure 2-1: Graph processing phases. 19
Figure 2-2 Graph processing architectures. 26
Figure 2-3 Master-workers architecture. 26
Figure 2-4 Taxonomy of programming models used by graph processing frameworks 31
Figure 2-5 Graph element-based approaches for graph processing frameworks 34
Figure 2-6 Distributed coordination 38
Figure 2-7 Classification of computational models in graph processing systems 43
Figure 2-8 Partitioning views in graph processing systems 49
Figure 2-9 Communication models in graph processing systems 52
Figure 2-10 Shared memory model with ghost (mirror) vertices 56
Figure 2-11 Storage view 57
Figure 2-12 Fault-tolerance in graph processing systems 60
Figure 2-13 Graph processing scheduling methods 63
Figure 2-14 Popularity changes in using databases 65
Figure 2-15 Proposed graph processing features' categorization in this chapter 70
Figure 2-16 Graph processing features' categorization according to application characteristics and computing platforms 70
Figure 3-1 Giraph's Architecture 77
Figure 3-2 Internal vertices and border vertices 79
Figure 3-3The role of high degree border vertices in reducing network traffic 85

Figure 3-4 Worker W2 has sent more messages to W1 than other workers 87
Figure 3-5 System architecture and components 87
Figure 3-6 Number of network messages transferred between partitions across supersteps for the Amazon graph using connected components algorithm 90
Figure 3-7 Number of network messages transferred between partitions across supersteps for the Pokec graph using connected components algorithm 90
Figure 3-8 Number of machines varying during supesteps while running connected component algorithms on different datasets on iGiraph 90
Figure 3-9 Total time taken to perform connected components algorithm 91
Figure 3-10 Number of network messages transferred between partitions across supersteps for the Amazon graph using shortest path algorithm 93
Figure 3-11 Number of network messages transferred between partitions across supersteps for the Pokec graph using shortest path algorithm 93
Figure 3-12 Number of network messages transferred between partitions across supersteps for the YouTube graph using shortest path algorithm 93
Figure 3-13 Number of machines varying during supesteps while running connected component algorithms on different datasets on iGiraph 94
Figure 3-14 Total time taken to perform shortest path algorithm 94
Figure 3-15 The average number of network messages in each experiment 95
Figure 3-16 Total time taken to perform PageRank algorithm 95
Figure 4-1 Graph applications and processing environment categorization 108
Figure 4-2 Network bandwidth unevenness in Amazon EC2 small instances with (a) 64 instances and (b) 128 instances [46] 112
Figure 4-3 Mapping strategy for 5 partitions and 5 workers. Partitions with higher priorities are assigned to the machines with higher bandwidth 113
Figure 4-4 Percentage of CPU idle time in a system with 15 workers 116
Figure 4-5 The components that we added to original iGiraph are shown in dotted rectangles 118
Figure 4-6 System architecture 120
Figure 4-7 Number of network messages transferred between partitions across supersteps for Amazon graph using shortest path algorithm 122

Figure 4-8 Number of network messages transferred between partitions across supersteps for YouTube graph using shortest path algorithm 122
Figure 4-9Number of network messages transferred between partitions across supersteps for Pokec graph using shortest path algorithm 123
Figure 4-10 Number of machines varying during supersteps while running shortest path algorithm on different datasets 124
Figure 4-11 Total time taken to perform shortest path algorithm 124
Figure 4-12 The average number of network messages in each superstep 124
Figure 4-13 Total time taken to perform PageRank algorithm 124
Figure 4-14 Total time taken to perform shortest path algorithm 125
Figure 4-15 Total time taken to perform PageRank algorithm 125
Figure 4-16 Number of machines varying during supesteps while running shortest path algorithms on different datasets 126
Figure 4-17 The average number of network messages in each experiment 126
Figure 5-1 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 136
Figure 5-2 Proposed auto-scaling architecture 139
Figure 5-3 General horizontal scaling policies 140
Figure 5-4Scaling 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 141
Figure 5-5 Number of machines during processing shortest path on Amazon 150
Figure 5-6 Number of machines during processing shortest path on YouTube 150
Figure 5-7 Number of machines during processing shortest path on Pokec 151
Figure 5-8 Number of machines during processing shortest path on Twitter for the first 50 supersteps 151
Figure 5-9 Total execution time for processing connected components algorithm on various datasets 151
Figure 5-11 Resource modification during processing shortest path on Amazon 153

Figure 5-12 Resource modification during processing shortest path on YouTube 153
Figure 5-13 Resource modification during processing shortest path on Pokec.....153
Figure 5-14 Number of machines during processing connected components on Amazon 154
Figure 5-15 Number of machines during processing connected components on YouTube.....155
Figure 5-16 Number of machines during processing connected components on Pokec 155
Figure 5-16 Number of machines during processing connected components on Twitter 155
Figure 5-17 Total execution time for processing connected components algorithm on various datasets.....156
Figure 6-1 The workflow of our proposed solution (GPaaS) 170
Figure 6-2 The components that we added to our work in Chapter 5 are shown in dotted rectangles.....171
Figure 6-3 Scenario1 183
Figure 6-4 Scenario 2 – Price(#VMs) Comparison.....184
Figure 6-5 Scenario 2 – If Giraph follows the job order.....184
Figure 6-6 Scenario 3 185
Figure 6-7 Total execution time per scenario.....186
Figure 7-1 Future directions.....193
Figure 7-2 Data processing approaches.....194

List of Tables

Table 2-1 Graph-like application and environments 17
Table 2-2 Graph algorithms categorization 23
Table 2-3 Overview of existing graph processing frameworks 68
Table 3-1 Evaluation datasets and their priorities 88
Table 3-2 Processing cost on different frameworks 96
Table 4-1 Comparison of the most related works in the literature 107
Table 4-2 Processing cost for SSSP on different frameworks 124
Table 4-3 Processing cost for PageRank on different frameworks 125
Table 4-4 Processing cost for SSSP on different frameworks 126
Table 4-5 Processing cost of PageRank on different frameworks 126
Table 5-1 VM characteristics 149
Table 5-2 Processing cost for SSSP on different frameworks 154
Table 5-3 Processing cost for CC on different frameworks 156
Table 5-4 Comparison of scheduling and resource provisioning algorithms 157
Table 6-1 Comparison of the most related works in the literature 169
Table 6-2 Input scenarios for evaluation 182
Table 6-3 Processing cost for each scenario in different systems 186

Chapter 1
Introduction

BIG Data era is the result of Information and Communication Technology (ICT) fast development in recent years where several Internet-scale applications and connected devices have created enormous data explosion. Big Data continues to find new application areas with representatives of data in a various formats and characteristics. A big fraction of generated data these days by applications ranging from social networks to Internet of Things (IoT) to search engines and mobile computing is in the form of graphs. A graph is made up of a series of nodes (vertices) that are in some way connected (via edges) and can have different properties.

Everything is connected in today's world and graphs are everywhere (Fig. 1.1). Social networks such as Facebook, Twitter and YouTube are significantly generating large amount of data every day while a majority is stored as graph data. In a typical social network, everything can be mapped to a graph. Graph of users is shaped by placing each member as vertices of the graph while the connections between them form the edges of the graph. Likewise, other graphs can be formed for posted comments, shared photos, common interests, video recommendation, friendship recommendation, etc. During each minute at 2017, 3.3 million posts were put on Facebook, 3.8 million queries were searched on Google search engine, 500 hours of new

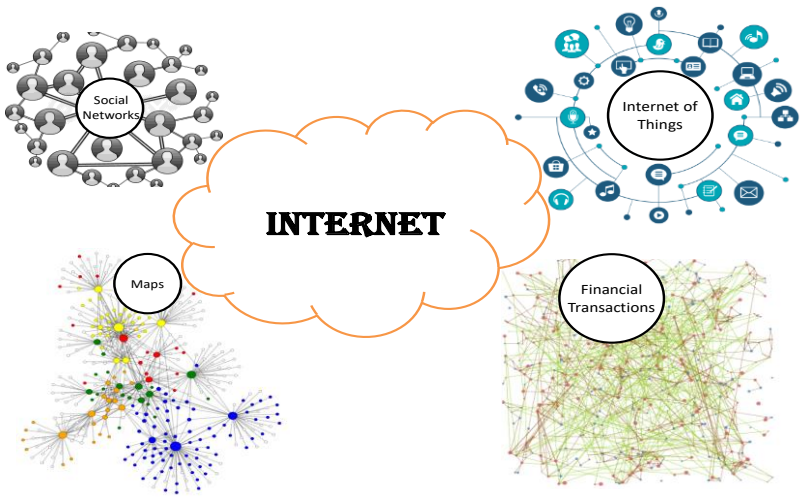


Figure 1-1Graphs are everywhere!

videos were uploaded on YouTube and 448.800 tweets were shared on Twitter[1]. These numbers are almost doubled compared to the amount of content was made per minute in 2014.

IoT is another significant source of exponentially large data generation. IoT includes billions of sensors and devices around the world that are collecting, measuring, detecting or enabling various activities and factors. These devices are used for smart homes, driverless cars, medical equipment, business supply chains, drone delivery services, mining, retails, etc. It has been predicted that by 2025, IoT will have an economic impact of $11 trillion per year while users will deploy one trillion IoT devices[2].

There are many other origins of graph data such as astronomy, telecommunication, mobile computing, machine learning, smart utilities, etc. However, traditional distributed data processing approaches such as MapReduce do not work efficiently on graph data because of a number of reasons. First, MapReduce is a two-step computational model which is not compatible with iterative inherit of graph algorithms. Second, a lot of disk I/O is required in MapReduce computation because middle states of the computation cannot be retained in the main memory. Third,

[1] https://www.smartinsights.com/internet-marketing-statistics/happens-online-60-seconds/
[2] https://www.gartner.com/newsroom/id/3598917

MapReduce programs are usually agnostic about the relationships and connected nature of a graph.

On the other side, graph processing brings intrinsic challenges due to the nature of graph characteristics. These characteristics include data driven computation requirements, irregular graph problems, poor locality during computation and high data access to computation ratio. To overcome these challenges and shortcomings, specific-designed systems are introduced for processing large-scale graphs. To overcome these challenges and the issues with traditional processing solutions, exclusive graph processing frameworks are started to be developed since 2010.

Cloud computing paradigm offers on-demand and scalable distributed storage and processing services like never before. It has brought new solutions such as elasticity, distributed computing and pay-as-you-go model by which it overcomes challenges and restrictions of traditional computing. Cloud computing treats computing as a utility where users have access to different services they need without knowing where the service is hosted or how it is being delivered. So, it is profitable for both service providers and consumers.

Despite all the advantages that cloud computing provides, most existing graph processing frameworks are developed on high performance computing (HPC) clusters. Unlike cloud environments, HPC infrastructure cannot be afforded by everyone hence these solutions are no available widely. There are few research works in the literature that propose cloud-based graph processing frameworks, but the problem with these systems is that similar to their HPC counterparts, cloud-based systems also try to improve the performance through utilizing novel computation or communication techniques. Other cloud features such as pricing and scalability are neglected in most research studies. In this thesis, we discuss the challenges and problems in processing large-scale graphs and provide new solutions to address them.

1.1 Challenges in large-scale graph processing on cloud environments

The trends of utilizing unprecedented computing capabilities that is provided in cloud environments namely as distributed computing, elasticity and pay-as-you-go pricing models along with the rising interests towards graph processing applications bring many challenges and research questions. We discuss the most important challenges regarding the requirements of new approaches in order to employ the potential of new provided computing paradigm to improve processing of large-scale graphs.

1.1.1 Cost Limitations and Models

Cost is an important factor in cloud environments. Cloud providers provide different types of infrastructures and facilities for highly storage or computing intensive programs which customers should pay to be able to use them. Cloud providers usually provide three types of clouds: public, private and hybrid. A giant cloud provider such as Amazon even has three different cost models including: reserve, spot and on-demand models. However, most current graph processing frameworks have not considered cost factor in their computations. One reason is that many of the systems in this area have been developed using high performance computing (HPC) facilities and cluster computing environments and they have assumed that the resources are limitless. Even some works that are assuming cloud metrics to explain their techniques have used clusters to simulate a cloud environment while the communication and cost limitations are very different on clouds. So, to propose a suitable cost model for a graph processing system we should consider the following issues: 1) What computation model is used to process the graphs? This factor is important because the cloud requirements can be found in the computation model. 2) Which cost model is more suitable for the computation model? So, we need to map the cost model with the computation model to choose the best fit option. 3) Is that possible to change the policy during the computation so that we can reduce the cost as much as possible?

1.1.2 Scalability and Communication Models

Many current graph processing frameworks use message passing interface (MPI) as the communication platform between different machines. As long as the system works base on cluster environment or there is no need to apply any scalability during the

computation, then MPI works well for disk-based frameworks. But in a cloud environment, particularly when the number of virtual machines varies during the computation, MPI is not useful for communication. Moreover, despite introducing new communication models such as in-memory or pull/push communication by recent graph processing systems, these models are either suitable for single machine processing architectures or they are not quite fit into the cloud environment.

1.1.3 Standard Entry and I/O Issues

Another issue in graph processing systems is that they are very costly in terms of I/O because the number of readings and writings on the disk is very high. When it comes to cloud infrastructures it would still get worse since the graph needs to be transferred to the cloud at least once. The right policy should be taken to partition the graph correctly and distribute the proper number of partitions on virtual machines (VMs) to reduce the number of inputs and outputs. It also affects the computation algorithm to decrease the communication part burden. On the other side, although there are some standard datasets which are used in most of the experiments, each framework change the structure of the system data entry according to the processing model it proposes. So, there is no standard form of data structure to be used by graph algorithms.

1.1.4 Network Issues and Limitations

Although all the current research works have measured the execution time and I/O performance for different graph processing frameworks, only a few of them have considered network issues. In a network, many factors such as latency, response time, network bandwidth, number of packages transferring through the network, etc. can affect the performance of the system, particularly in distributed environments such as clouds. Most existing solutions do not investigate the effects of network factors. Instead, they try to improve the performance of the system by proposing new partitioning algorithms or using new computation models.

1.1.5 Resource Provisioning

In the literature, provisioning resources before starting the processing procedure is very common. Hence, the graph processing system and its resource provisioning

component must be optimized towards configuring the computing nodes prior to the deployment of the system in order to enhance the performance and resource utilization. However, this approach cannot guarantee that these frameworks are successfully achieving the desired cost and performance goals. Nevertheless, cloud computing offers features such as elasticity and scalability by which required number of resources can be determined at any stage during the life of an application. The ideal would be provisioning resources based on the graph application and the dataset that is being processed to reduce the monetary cost and improve the performance.

1.2 Research Problems and Objectives

This thesis aims to investigate and provide resource management mechanisms in a distributed graph processing environment while considering cost-efficiency and its impact on the system performance using various partitioning and scheduling techniques. To address the challenges that were discussed in the previous section, this thesis has identified and investigated the following research problems:

- **How to utilize the scalability and elasticity of cloud environments to provision the optimal number of resources for processing large-scale graphs?** As mentioned in the previous section, existing distributed graph processing frameworks are using a pre-defined and pre-configured number of resources which they keep up until the end of the operation. The problem with this method is that the resources are being provisioned in a static manner rather than dynamically. Therefore, in many situations during the operation, some machines have no tasks to operate on while they are still occupied and kept active by the system. This method is costly and will not necessarily provide the best performance. Instead, a dynamic approach by which resources on the cloud environment can be provisioned and scheduled according to the characteristics of the graph algorithms could give better results.

- **How to partition the graph efficiently to optimize the performance?** Partitioning plays a very important role in processing large-scale graphs in a

distributed environment. In many places in this thesis we discuss that a static partitioning approach is not as efficient as considering a dynamic repartitioning method. There are various factors that need to be considered while employing a partitioning mechanism that affect its usability. For example, in a distributed environment such as cloud: 1) what is the optimized number of partitions for a particular application?, 2) how many partitions should be placed on each machine?, 3) how to minimize the number of cross-edges between machines?, 4) how often should repartitioning happen?, etc.

- **How to consider and employ network factors to design an effective partitioning mechanism in order to reduce the monetary cost and improving the performance?** Without considering network factors such as bandwidth, latency, topology, etc. in a constantly changing area as clouds, any solution will be incomplete. These factors significantly affect the operation and its execution time. However, most existing distributed graph processing solutions are either completely agnostic about network metrics or they give only a small weight to them. Thus, a dynamic resource scheduling mechanism that takes critical network factors as variables in its solution is required to achieve optimized performance and reduce the cost.

- **How to utilize the heterogeneity of cloud resources to define appropriate provisioning policy at any time during the processing?** An important feature that is provided by cloud environments is the variety of resources that are accessible on their infrastructure. A typical client can utilize any size of computing machines, storage, communication bandwidth and other services according to his available budget, job priorities or even deadlines. This provides a remarkable capacity for optimizing any types of operations including graph processing. Although, heterogeneous resources have been used to manage generic kind of operations in many research works, this significant opportunity has been neglected in the studies related to large-scale graph processing. Therefore, combining exclusive graph algorithms' properties

with the appropriate partitioning techniques to utilize heterogeneous resources in a distributed manner can improve the efficiency of the system.

- **How to maintain and monitor the quality of service (QoS) in cloud-based graph processing systems?** Another important issue about using cloud facilities to operate a particular service or processing on them is to ensure that the service level agreement (SLA) will be preserved and guaranteed. If large-scale graph processing is going to be provided as a service to a broader range of users, then there should be some mechanisms to assure that the service will be delivered seamlessly and the reaction against any violation of the agreement has been predicted in advance. Hence, quality of the service needs to be managed and monitored comprehensively alongside the efforts for ameliorating the throughput and performance of the system.

1.3 Evaluation Methodology

All the proposed approaches in this thesis were designed and implemented in the form of real frameworks and evaluated using real-world graph datasets including Amazon (TWEB), YouTube links and Pokec along with various graph applications such as PageRank, single source shortest path and connected components. All algorithms and frameworks were implemented on Australian National Research Cloud Infrastructure (NECTAR) and compared against real-world benchmarks. Since NECTAR does not correlate any price to its infrastructure for research use cases, the prices for utilized virtual machines are put proportionally based on Amazon Web Service (AWS) on-demand instance prices in Sydney region according to closest VM configurations as an assumption for our works.

We are following an additive approach for implementing our solutions in this thesis. This means that we took a popular and widely used distributed graph processing framework called Apache Giraph as a base for our work while in each chapter we are plugging-in new features and algorithms to the system to modify and improve the solution in the previous chapter. However, understanding each chapter is necessary before moving to the next one. Major metrics such as execution time,

monetary cost and number of messages passing through the network along with other solution-specific metric that are presented in each chapter have used for evaluation and comparison of different benchmarks.

1.4 Thesis Contribution

To address the research problems mentioned in Section 1.2, this thesis made the following **key contributions**:

1. A survey and taxonomy of the state-of-the-art advances and improvements on large-scale graph processing frameworks
2. A system framework named as iGiraph which enables a cost-efficient graph processing in cloud environments
    - The design of the framework architecture and module interactions
    - A new dynamic repartitioning method that utilizes network traffic pattern to reduce the communication between compute nodes
    - Using a new behaviour-based classification for graph algorithms and operate accordingly
3. Two network-aware dynamic repartitioning-based algorithms for scheduling large-scale graphs deployed to consider some important network factors in the processing
    - Adding the second dimension (level) to the behaviour-based classification of graph algorithms to distinguish applications and select more accurate processing policy
    - A novel mapping strategy is designed to facilitate assigning partitions to the machines based on different features that each partition and machine has.
    - A new bandwidth-and-traffic-aware dynamic re-partitioning algorithm and a new computation-aware re-partitioning algorithm have been proposed by which the monetary cost of the operation declines remarkably

4. An auto-scaling algorithm which enables the system to take advantage of the heterogeneous resources providing in a cloud environment
    - A new cost-efficient provisioning of heterogeneous resources for convergent graph algorithms
    - A resource-based auto-scaling algorithm which significantly increases the capability of supplying the efficient number of required resources out of the available resource pool on the cloud by scaling horizontally
    - A characteristic-based dynamic repartitioning mechanism combined with a smart process monitoring that allows effective partitioning of the graph across available VMs according to VM types.
    - A new implementation of the operation management on the master machine
5. Monitoring and maintaining the quality of service (QoS) in a large-scale graph processing environment
    - A prioritization mechanism to identify and distinguish between different graph workloads and algorithms
    - Deploying a service level agreement (SLA) monitoring mechanism to fulfil the SLA requirements

1.5 Thesis Organization

The structure of the thesis is shown in Figure 1.2, which its chapters are derived from several research works that conducted or published during my PhD candidature.

- Chapter 2 presents a taxonomy and survey on the state-of-the-art advances and development on large-scale graph processing frameworks. This chapter is partially derived from:
    - **Safiollah Heidari**, Yogesh Simmhan, Rodrigo N. Calheiros and Rajkumar Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges", *ACM Computing Surveys*, vol. 51, Issue. 3, No. 60, ACM Press, New York, USA, 2018.

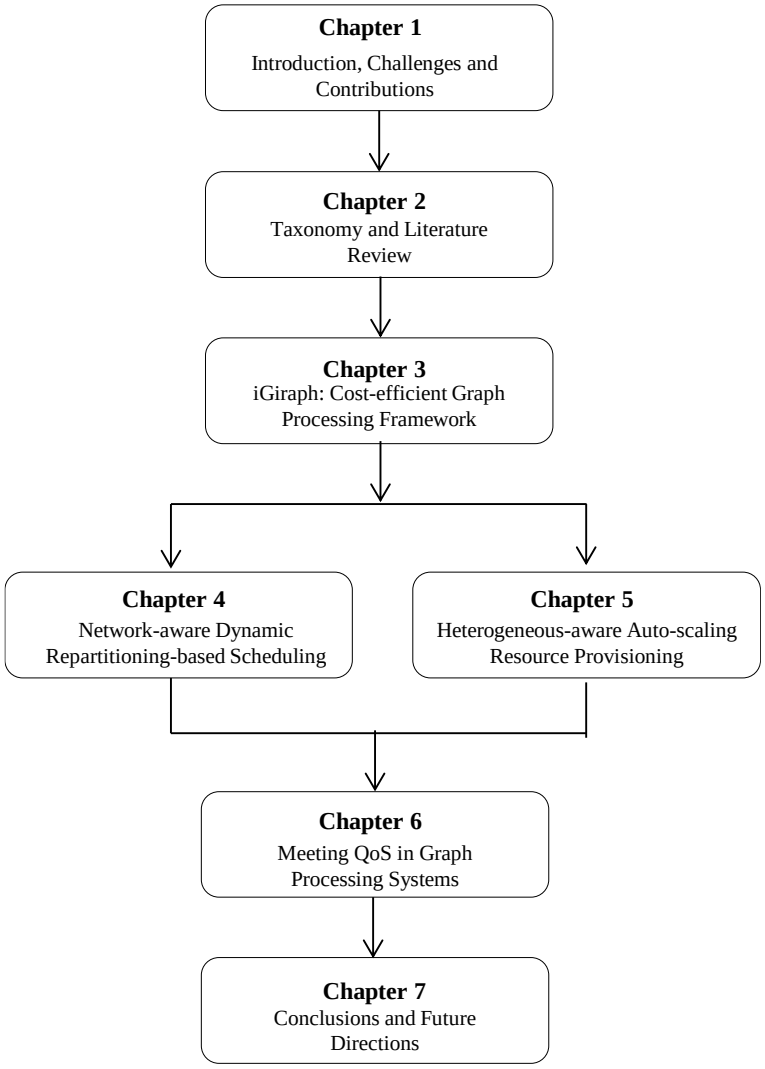


Figure 1-2 The thesis organization

- Chapter 3 proposes a cost-efficient graph processing framework called iGiraph by which the cost of the operation decreases dramatically. This chapter is derived from the following publication:
  - Safiollah Heidari, Rodrigo N. Calheiros and Rajkumar Buyya, "iGiraph: A Cost-efficient Framework for Processing Large-scale Graphs on Public Clouds", in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016)*, Cartagena, Colombia, Pages 301-310, 2016.
- Chapter 4 proposes two network-aware dynamic scheduling algorithms that are considering important network factors such as network traffic, bandwidth

and computation utilization for partitioning the graph and distributing the partitions across the system. This chapter is derived from:

  - **Safiollah Heidari** and Rajkummar Buyya, "Cost-efficient and Network-aware Dynamic Repartitioning-based Algorithms for Scheduling Large-scale Graphs in Cloud Computing Environments", *Software: Practice and Experience (SPE)*, vol, 48, Issue 12, Wiley & Sons, 2018.

- Chapter 5 proposes an approach to take advantage of heterogeneous resources that is provided in a cloud environment to minimize the cost of the operation. This work is derived from:

  - **Safiollah heidri** and Rajkumar Buyya, "A cost-efficient Auto-scaling Algorithm for Large-scale graph processing in Cloud Environments with Heterogeneous Resources", *IEEE Transactions on Software Engineering (TSE)*, 2018 (Under review).

- Chapter 6 proposes a comprehensive system to monitor and maintain the quality of service (QoS) in graph processing systems. This work is derived from:

  - **Safiollah Heidari** and Rajkumar Buyya, "Quality of service (QoS)-driven Resource Provisioning for Large-scale Graph Processing in Cloud Computing Environments: Graph Processing-as-a-Service (GPaaS)", *Future Generation Computer Systems (FGCS)*, 2018 (Second Review: Minor Rev.).

- Chapter 7 concludes the thesis with a summary of the key findings and a discussion of future research directions. This chapter is partially derived from:

  - **Safiollah Heidari**, Yogesh Simmhan, Rodrigo N. Calheiros and Rajkumar Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges", *ACM Computing Surveys*, vol. 51, Issue. 3, No. 60, ACM Press, New York, USA, 2018.

Chapter 2
Taxonomy and Survey of Graph Processing Systems

The world is becoming a more conjunct place and the number of data sources such as social networks, online transactions, web search engines and mobile devices is increasing even more than had been predicted. A large percentage of this growing dataset exists in the form of linked data, more generally, graphs, and of unprecedented sizes. While today's data from social networks contain 100's of millions of nodes connected by billions of edges, inter-connected data from globally-distributed sensors that forms the Internet of Things (IoT) can cause this to grow exponentially larger. Although analyzing these large graphs is critical for the companies and governments that own them, big data tools designed for text and tuple analysis such as MapReduce cannot process them efficiently. So, graph distributed processing abstractions and systems are developed to design iterative graph algorithms and process large graphs with better performance and scalability. These graph frameworks propose novel methods or extend previous methods for processing graph data. In this article, we propose a taxonomy of graph processing systems and map existing systems to this classification. This captures the diversity in programming and computation models, runtime aspects of partitioning and communication, both for in-memory and distributed frameworks. Our effort helps to highlight key distinctions in architectural approaches, and identifies gaps for future research in scalable graph systems.

This chapter is partially derived from:

- **Safiollah Heidari**, Yogesh Simmhan, Rodrigo N. Calheiros and Rajkumar Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges", ACM Computing Surveys, vol. 51, Issue. 3, No. 60, ACM Press, New York, USA, 2018

2.1 Introduction

THE growing popularity of technologies such as Internet of Things (IoT), mobile devices, smart phones and social networks has led towards the emergence of "Big Data". Such applications produce not just gigabytes or terabytes of data, but soon petabytes of data that need to be actively processed. Such large volumes of data gathered from billions of connected people and devices around the world is causing unprecedented challenges in terms of how data can be stored, retrieved and managed; how data security, integrity, availability and sharing can be ensured; how massive datasets can be mined; and how they can benefit from new computing paradigms such as cloud computing for data analysis [178][58].

According to the National Research Council of the US National Academies [55], graph processing is among the seven major computational methods of huge data analysis. Graph computations are used in business analytics, social network analytics, image processing, hardware design and deep learning to an increasing extent. Wide-spread techniques for processing large graphs had, untill recently, been limited to shared memory [97] [19] and High Performance Computing systems, [116] [93]. Although distributed approaches have been proposed for processing big graphs since 2001 [99], graph processing systems for commodity clusters and Clouds have become particularly popular after Google introduced its Pregel [148] vertex-centric graph processing system in 2010. Since then, several distributed graph processing frameworks with diverse programming models and features have been proposed to facilitate operations on large graphs. Each of these frameworks has specific characteristics with its own strengths and weaknesses.

The aim of this chapter is to provide a taxonomy of scalable graph processing systems and frameworks. It identifies strengths and weaknesses in the field and proposes future directions. First, it proposes a comprehensive taxonomy of programming abstractions and runtime features offered by graph processing systems, and maps the existing systems to this taxonomy. Second, it utilizes a top-down approach for investigating graph processing frameworks and their components along with examples to support them. Third, the chapter identifies gaps in existing systems

which need further investigation, and discuss these open problems and future research directions in detail. In summary, this survey gives readers an overarching picture about what graph processing is, what improvements have been gained through recent frameworks, different programming and runtime techniques that have been used, and the applications that benefit from them. It emphasizes scalable graph processing platforms for shared-memory and distributed processing, that fall within the ambit of Big Data processing platforms. It also contrasts them against graph frameworks for supercomputing systems, as evidenced through the Graph500 benchmark[3]. On the other hand, the existing works such as [153] and [60] only focus on surveying and they have limited focus on key elements of graph processing.

The rest of the chapter is organized as follows: Section 2.2 includes a definition of graphs and graph processing systems, contrasts graph processing from other big data processing methods, outlines the lifecycle of a typical graph processing system, and gives examples of real graph-based applications and algorithms. Contemporary graph processing frameworks and architectures are explained in Section 2.3, along with distributed coordination and computational models. Section 2.4 categorizes existing frameworks based on partitioning, communication models, in-memory execution, fault tolerance and scheduling. Graph databases are reviewed in section 2.5. A taxonomy and discussion on challenges is also presented for each section. A gap analysis and open challenges, with a perspective on future directions are discussed in Section 2.6 and 2.7 respectively. Finally, we conclude the chapter in Section 2.8.

2.2 Background

A Graph G = (V, E), consists of a set of vertices, V= {v1, v2, … , vn} and a set of edges, E={e1,e2,…,em} that indicate pairwise relationships, E=V×V. If (vi , vj)∈E, then vi and vj are neighbors [199]. The edges may be directed or undirected. So V and E are the two defining characteristics of a graph which most of graph processing frameworks implement. Frameworks typically support a single attribute value associated with the

[3] Graph 500 benchmark, http://www.graph500.org/

vertex and edge (e.g., label, weight). In addition, some of the platforms also support a set of named and/or typed attributes for their vertices and edges as part of their data model.

Table 2-1 Graph-like application and environments

Application	Item (Vertices of the Graph)	Connection (Edges of the Graph)
Social network	Members	Friendships
Computer network	Computers	Network connectivity
Web content	Web Pages	Hyperlinks
Transportation	Cities	Roads
Electrical circuit	Devices	Wires
Commerce	Customers, Goods	Purchase transactions
Factory	Machines	Production lines
Supply chain	Providers	Distances
Telecommunication	Mobile Phones	Phone calls

Pairwise relationships between entities play an important role in various types of computational applications. These relationships that are implied by different connections (edges) between items (vertices) give rise to domain questions to draw value from the data, such as: Is it possible to identify transitive relationships between items by following the connections? How many items are connected to a typical item? What is the shortest distance between these items? Which groups of items are similar to each other? How important is an item relative to others? Various graph-like applications and environments are mentioned in Table 2.1 [199]. As can be seen in Table 2.1, many applications process data that naturally fits into a graph data model. Several of these applications from social networks, eCommerce and telecom domains handle large graph datasets which need to be processed and mined to draw disparate business intelligence, ranging from the interests of people about products for targeted advertising, to tracing call logs for cyber-security. Processing large graphs poses some intrinsic challenges due to the nature of graphs themselves. These characteristics make graph processing ill-suited to existing data processing approaches, and usually inhibit efficient parallelism [177]. According to [144], their properties are noted below:

(1) *Data-driven computations*: Graph computations are usually entirely data-driven. Graphs are made up of sets of vertices and edges that dictate the computations performed by every graph algorithm.

(2) *Irregular problems*: Graph problems are highly irregular due to the non-uniform edge degree distribution and topological asymmetry rather than being uniformly predictable problems which can be optimally partitioned for concurrent computation.
(3) *Poor locality*: The inherent irregular characteristics of graphs leads to poor locality during computation, which is in conflict with locality-based optimizations supported by many existing processors, making it difficult to achieve high performance for graph algorithms.
(4) *High data access to computation ratio*: A large portion of graph processing is usually dedicated to data access in graph algorithms. Therefore, waiting for memory or disk fetches is the most time-consuming phase relative to the actual computation on a vertex of edge itself.

To streamline the processing of big data, MapReduce, a distributed programming framework for processing large datasets with parallel algorithms, was introduced by Google in 2004 [56]. MapReduce has two significant advantages: 1) The programmer has a simple and familiar interface using Map and Reduce functions, inspired by functional programming concepts [100], and 2) the application is automatically parallelized when defined using Map and Reduce methods, without the programmer needing to know how data will be distributed, grouped and replicated, and how the tasks are scheduled.

Although MapReduce addresses many deficiencies in traditional parallel and distributed computing approaches, it has several limitations that make it less efficient for processing large graphs [51] [2] [82]: 1) MapReduce is limited to a two-phased computational model that is not naturally suited for graph algorithms that run over many iterations, 2) In common MapReduce implementations, the input graph and its state are not retained in main memory across even these two phases, let alone across iterations, and consequently requires repetitive disk I/O, 3) MapReduce's tuple-based approach that is unaware of the linked nature of graph datasets is poorly suited to design many graph applications, and 4) Graph operations using MapReduce have poor I/O efficiency - because of frequent checkpoints on completed tasks and data replication - which is a bottleneck for many graph algorithms [131].

Apache Hadoop [12] is a popular open-source implementation of the MapReduce programming model. Besides flexible batch processing applications that can be built using Hadoop, it is also the basis for NoSQL querying platforms such as Pig [16] and Hive [15] to work with large datasets. In addition, various high level languages such as SCOPE [266], Sphere [86] and Swazal [179] are available for MapReduce-like systems. Platforms like Apache Spark [255] have extended the programming model of MapReduce, and offer incremental batch and in-memory computation with better performance. Further, Hadoop's distributed file system storage mechanism (HDFS) as well as a Map-only model of Hadoop is used as the storage and distributed scheduling mechanism in many graph processing frameworks we discuss.

While a number of systems such as PEGASUS [114] have brought innovative approaches for processing and mining peta-scale graphs, those systems are based on the MapReduce model and suffer from the above limitations. As a consequence, iterative graph processing systems started to emerge in 2010 with Google's Pregel [148], a graph processing framework that uses Valiant's Bulk Synchronous Parallel (BSP) processing model [231] for its computation. Pregel was the first system that promoted a "Think Like A Vertex" notion for processing large graphs, similar to MapReduce that operates on <key,value> pairs to process large data volumes. These and other contemporary graph processing systems are discussed further in this survey.

2.2.1 Overall Scheme of Graph Processing

In general, a typical graph processing systems execute a graph algorithm over a graph dataset across different logical phases, as shown in Figure 2.1.

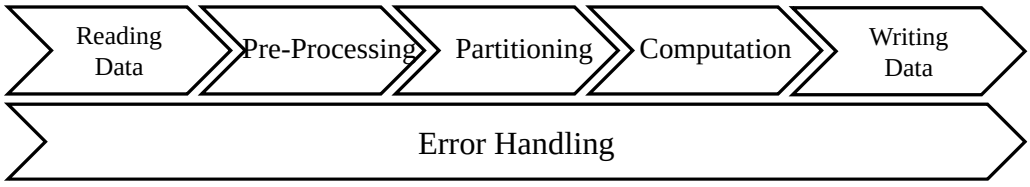


Figure 2-1: Graph processing phases.

(1) Read/Write input/output datasets: The first step is reading the graph data from a source dataset which can be either on disk or in memory. In the last phase, the

processed data should be written back again, either to disk or memory. Graph processing systems typically do not have a custom persistence layer optimized for reading and writing graph datasets, and tend to use the standard file system such as HDFS. Hence, they can present a bottleneck when reading and writing large graph datasets. Many studies even ignore read/write time when they measure the execution time for evaluation. Instead, they try to improve other aspects of the systems like efficient in-memory data structures for computation.

(2) Pre-processing: In some approaches, the graph data will be partitioned before being passed to the graph processing system to decrease the overall burden and runtime of the system. The main advantage of this approach is that the programmer does not need to worry about the complexity of the partitioning and it is a one-time cost that is paid up-front. Also, partitioning mechanism and computation mechanism can be two different modules which work independently and thus can be designed and implemented separately. The main drawback for this approach is that it works well only for static partitioning strategies, not dynamic partitioning or repartitioning.

(3) Partitioning: In this phase, partitioning will be done dynamically within the graph processing system and not as a separate module. Both partitioning and computation phases can collaborate to choose the best partitioning method at each step, so, dynamic partitioning and repartitioning can be implemented in the processing system. Although programming such a system is more complicated than implementing two independent modules, it provides more runtime flexibility and can be well suited to support diverse graph algorithms.

(4) Computation: Different graph processing systems have different computation approaches. This programming model and runtime is at the heart of the whole framework and there have been many proposals for efficient computation methods to decrease the graph application's runtime. More details about this phase, with a taxonomy on various computational models, is presented in Section 2.3.4.

(5) Error Handling: This fault-tolerant and failure recovery phase will be applied to the system either during the computation phase or after the computation phase is completed. There are various techniques that can be used here, such as check-pointing or restarting applications. Typically, the time taken for error handling is not considered

in experimental results due to the overheads it causes and some frameworks even avoid considering this capability. However, given the use of large commodity clusters that are prone to failures and long running big data applications, fault tolerance is essential for graph processing frameworks used in an operational setting.

2.2.2 Large Graph-Oriented Applications

As noted in Table 2.1, there are many fields and applications that generate and provide big data in the form of graphs. With improvements in computer hardware and processing models such as cloud computing and emerging concepts like IoT, an even greater growth in datasets is expected. Here, we present some typical applications and environments where large graph data are generated and used.

(1) Social Networks: Social Networks and applications have grown exceedingly popular during the past decade and are constantly adding features to make effective use of the data they collect and to grow their customer network. Social networks are an important source of big graph data, and even big data in general, with large amounts of data created every day [52]. People are sharing their personal activities with their friends and the whole world, talking about their beliefs, sharing photos and videos, and posting their interests and health information [38] [213]. In the year 2014, in each minute, 2,460,000 content posts are shared on Facebook, 3,472 photos are pinned on Pinterest, 72 hours of new videos are uploaded to YouTube, 278,000 tweets are shared on Twitter, 20 million photos are viewed on Flicker. These rates continue to grow, and form just a part of the whole big data social network landscape [87].

Social networks are native generators and consumers of graph datasets, with an additional temporal dimension added to them. “Users” form the vertices of a huge social graph while “friendship” connections between them form the edges of the graph. Connections can be probabilistic and node’s states change over time. Each node or edge can contain different values and information about a member’s personal details, his/her interests, friends, groups and people, his/her followers, the pages that are visited, locations, business information and so on with many other meta-data about

his/her history of activities. All these form a digital trail for every user that needs to be processed and analyzed by social network providers.

From the users point of view, the network provider needs to suggest relevant pages or communities for them to follow based on their interests and offer meaningful service offerings [3] [21]. The providers themselves benefit from leading users through targeted advertising to paid services. Although popular, social network sites are still in their infancy as they figure out how to monetize this massive dataset they have access to and make their business model sustainable. New methods and mechanisms are emerging in the area of analyzing social network data on distributed systems, clusters and clouds [133].

(2) Computer Networks and the Internet: Every machine in a computer network, including clients, servers, routers and switches, is a node of a network graph and physical or network connections between these machines form the edges of the network graph. When various networks from all over the world are connected together to provide different services, it forms the "Internet" which is an extremely large graph [43]. Computer networks need to be analyzed to discover whether there might be intruders, resource wasters, low efficiency, dead paths, and also to gain statistical reports about the states of the network [7]. This is particularly the case as a bulk of the network traffic moves toward rich content such as streaming video and multi-player gaming. These types of graphs should be processed in real-time as their state changes, and need a fast response, say to configure switches to allocate bandwidth to traffic, or detect malware and denial of service attacks. Network delays lead to customer dissatisfaction or worse, outages can cripple the functioning of modern society.

(3) Smart Utilities: Many large graph datasets are owned by public utility and service providers such as city and rail roads, and power and water grids. Take city road datasets as an example. Logistics companies need to find the shortest path between cities and streets to decrease their fuel consumption and ensure timely delivery of their goods, Governments need to plan maintenance and provision emergency services in case of power disruptions or natural disasters, and people need to find the most convenient means for travel between different locations [138].

Further, with a wider deployment of IoT and city services getting smarter [176], the ability to monitor and collect real-time information about these physical infrastructure networks will grow, and graph analytics will be essential for ensuring the smartness of these utilities. In fact, IoT will be a natural extension and an exponential expansion of the Internet. Graph applications can be used to drive real-time management of power grid operations with back-to-grid intermittent renewables like solar and wind, pumping operations for water networks, signaling of traffic lights based on current flow patterns, and even scheduling of public transit on-demand.

There are many other examples such as telecommunication [151], web search engines [173], environmental analysis [55], astronomy [219], mobile computing [224], machine learning [256] and so on where large graph data is required to be processed and, as we mentioned before, traditional approaches are not suitable.

2.2.3 Algorithms in Graph Processing Studies and Experiments

Table 2-2 Graph algorithms categorization

Traversal	Breadth first search (BFS) Single source shortest path (SSSP)
Graph Analysis	Diameter Density Degree distribution
Components	Connected Components Bridges Triangle Counting
Communities	Max-flow min-cut K-means, Semi Clustering
Centrality Measures	PageRank Degree centrality Betweenness centrality
Pattern Matching	Path/subgraph matching
Graph Anonymization	K-degree anonym. K-neighborhood anonym.
Other Operations	Structural equivalence, Similarity, ranking, etc.

We discuss algorithms that are commonly used in most large graph processing studies and experiments. These algorithms are not essentially graph-designed algorithms in terms of the level of parallelism but they have been used for experiments in papers. So, the categorization presented in the table below provides a hint for researchers.

However, several works have been done on designing parallel versions of these algorithms to fit them into the graph processing domain [134] [147]. Table 2.2 shows the taxonomy of algorithms according to [61], which are used in a number of works.

(1) Graph Traversal Algorithms: These algorithms travel through all the vertices in a graph according to a specific procedure to check or update the vertices' values [199]. Amongst the most common algorithms in this type are Breadth-First-Search (BFS) and Depth-First-Search (DFS) [123]. Both of these algorithms traverse the graph tree to find a particular node, or visit every node in a specific order. Single Source Shortest Path (SSSP) is used to find the shortest path between a particular node and any arbitrary node of the graph that might be based on the minimum cost or weight [190]. Dijkstra's algorithm and Bellman-Ford algorithm are popular algorithms in this category [22].

(2) Graph Analysis Algorithms: These algorithms peruse the topology of the graph to specify graph objects and analyze its complexity. These graph statistics and topological measures are extensively used in protein interplay analysis and social network analysis [31] [128].

(3) Components: Connected components algorithms find subgraphs in which a path exists between any two nodes in the subgraph and none are connected to nodes in other subgraphs [96]. So, each vertex only belongs to one connected component of the graph. Weakly connected components work on undirected graphs, while strongly connected components are relevant to directed graphs. Another component identification problem is counting triangles.

(4) Communities: A community is a set of vertices in which each vertex in the community is closer to other vertices of the same community than any other vertices of the graph. Various topological and attribute measures can be used to define the closeness and quality of communities, and K-Means clustering and semi-clustering are popular algorithms in this category [106] [30].

(5) Centrality Measures: The aim of these algorithms is to give an approximate indication of the importance of a vertex in its community according to how well it is connected to the network. The most used algorithm of this type is PageRank [173], an algorithm that is used by Google search engine to rank websites. Betweenness centrality is another common metric [146].

(6) Pattern Matching: These algorithms are used to recognize the presence of input patterns in the graph, which can be an exact or approximate recognition [216].

(7) Graph Anonymization: These algorithms are used to create a new graph based on an original graph where the latter emulates specific topological or attribute properties of the original one. This prevents any possible intruders to re-identify the network [239].

(8) Other Operations: There are also other algorithms such as random walk algorithms where we choose a vertex randomly from neighbors of a vertex to start or continue process from there and try to converge in a probabilistic point [72].

In another categorization used by [91] and [112], algorithms have been categorized based on the types of graph queries which result in two classes of algorithms (other types of query classification can be found in [197] [108]):

(1) Global Queries: These queries need to traverse the whole graph. So, algorithms such as diameter estimation, PageRank, connected components, random walk with restart (RWR), degree distribution, etc. are in this group.

(2) Targeted Queries: These queries only need to access part of the graph, not all the graph. [112] has formulated seven types of queries including neighborhood (1-step and n-step), induced subgraph, egonet (1-step and n-step), k-core and cross-edges.

Although there are many algorithms that can be implemented on a graph processing system, there are some challenges that these algorithms faced. First, according to [144], many graph systems have limited memory that can be exclusively allocated to the processing algorithm, in addition to other processes and threads that simultaneously use and access the memory. Graph algorithms, in particular those that operate in a shared-memory system, can exceed available physical memory for large graphs processed on single machines [159]. Recent graph processing systems address this by using a distributed computing paradigm. In addition, utilizing external memory algorithms is another approach to reach out of memory (core) in order to have access to more space. Second, the level of granularity in an algorithm can influence the level of parallelism it can exploit, especially those with linear runtime. So, a more fine-grained level of parallelism results in better scaling of such algorithms [94]. Third, algorithms should deal with diverse workloads and need to reassign tasks to

processors when the visited nodes in a graph algorithm have spatial locality in the global memory. Finally, graph processing systems and algorithms should deal with the additional degree of parallelism exposed by submitting multiple concurrent queries when working on a large graph, or algorithms that operate over dynamic graphs. However, most of the systems that we review operate one graph algorithm or query over a single (large) graph.

2.3 Graph Programming Model

We present different dimensions of graph programming and computation models, and classify and analyze prominent literature on graph frameworks based on these categories. A comprehensive list of graph processing systems based on this taxonomy is tabulated in Table 2.3.

2.3.1 Graph Processing System Architectures

Graph processing systems can be categorized into three types of architecture models as depicted in Figure 2.2.

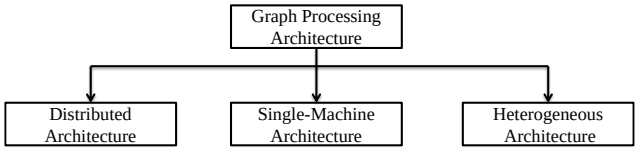


Figure 2-2 Graph processing architectures.

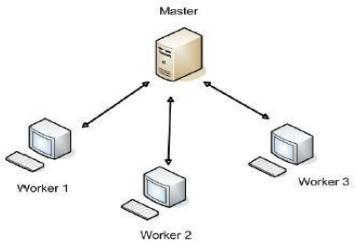


Figure 2-3 Master-workers architecture.

2.3.1.1. Distributed Architecture

A distributed system includes several processing units (host) and each host has access to only its own private memory. Each partition of the graph is typically assigned to one host to be processed while the hosts interact with each other by explicit or implicit message passing [214]. Such systems are meant to weakly scale by supporting larger graphs as more hosts are added to the system. From a cloud computing point of view, these map to an infrastructure as a service (IaaS) [35] architecture, where the hosts are Virtual Machines (VMs). Distributed graph processing systems utilize master-slaves

(workers) architecture, as shown in Figure 2.3, where there is one master that is responsible for managing the whole system, assigning partitions to workers, managing fault-tolerance, coordinating the operations of the workers and so on; and there are multiple workers that are responsible for performing computation on the partitions.

Although the programmer has to adapt their algorithms and applications to suit the abstractions provided by the distributed graph processing systems, such systems ease the scaling of the applications on distributed environments, without the challenges of races and deadlocks that are associated with distributed computing [53]. In contrast, shared-memory frameworks that have been developed for single machines are easier to program but are limited by their ability to hold only parts of large graphs in memory [204].

Google' Pregel is a distributed vertex-centric framework that uses a master-worker architecture on multiple hosts of a cluster. GraphLab, developed in Carnegie Mellon University and later supported by GraphLab Inc., was developed for single machine processing [142], but evolved into a distributed one [141]. There are other Pregel-like systems such as GPS [195], Mizan [118] and GoFFish [208], and non-Pregel-like systems such as Presto [233], Trinity [202], and Surfer [46], which have been developed as distributed graph processing systems. Even frameworks such as GraphX [81] are built on top of Spark distributed dataflow system. All of these systems use multi-node clusters or cloud VMs for their execution environment. However, as yet none of these exploit the elasticity property of Clouds, and rather treat captive VMs as a commodity cluster.

Beside the aforementioned graph frameworks, there are several graph processing libraries developed for high performance computing (HPC) clusters. Boost graph library (BGL) [205] is a generic graph processing library that provides generic interfaces to the graph's structure and common operations, but hides the details of its implementation. This allows graph algorithms using BGL to have interoperable implementations on shared-memory and parallel computing platforms. Graph500 is a graph processing benchmark by which various metrics of supercomputers such as communication performance, memory size for graph storage and the performance of random access to memory are measured. It contrasts with Top500 [226] which is

designed for compute-intensive applications. Although there have been many other attempts for providing parallel graph frameworks for high performance computing including several libraries such as MPI[68], PVM [76], BLAS [129], JUNG [109] and LEAD [156], none of them provide the required flexibility for a general-purpose graph processing platform [221].

2.3.1.2. Shared-memory Architecture

Prior to the recent growth in distributed graph processing systems, there have been several works on processing large scale graphs on a single machine. A single machine consists of one processing unit (host) which can have one or more CPU cores, and physical memory that ranges from a few to hundreds of gigabytes that is shared across all the cores.

In 2012, Microsoft researchers conducted a study [189] on whether using Hadoop on a cluster for analyzing big data is the right approach for data analytics. They concluded that for many data processing tasks, a single machine with large memory is more efficient than using clusters. They also investigated the cost aspect of using a single machine in big data processing and mentioned that *"… for workers that are processing multi gigabytes rather than terabytes+ scale, a big memory server may well provide better performance per dollar than a cluster."* [140].

Shared memory frameworks are inherently limited in the amount of memory and CPU cores present in that single machine [60]. The main challenge is that single hosts often have limited physical memory whereas processing large, real-world graphs can require a significant amount of memory to retain them fully in memory for many graph applications, or keeping and managing a subset of the graph out-of-memory. Novel techniques to address this limitation have been proposed.

In Strata Startup Showcase 2013, SiSense, which is a business intelligence solutions provider company, won the audience award with a software system called "Prism" that can exploit a terabyte of data on a single machine with only 8 GB of RAM [139]. It relies on disk for storage, transfers data to memory when needed and benefits from L1/L2/L3 caches of the CPU. It utilizes a column store and an interface which allows scalability to a hundred terabytes. Among major IT companies, for instance, Twitter

uses Cassovary [229], an open-source graph processing system that has been developed to handle graphs that fit in the memory of a single machine. It has been claimed that Cassovary is a viable system for "most practical graphs" because of using a space efficient data structure. WTF (who to follow) [89] is a recommendation algorithm which is used by Twitter to suggest users with common interests and connections that is implemented on Cassovary.

GraphChi [127] is a vertex-centric graph processing framework that proposes a parallel sliding window (PSW) method for leveraging external memory (disk) and is suited for sparse graphs. PSW needs a small number of sequential disk-block transmissions, letting it to perform well on both SSD (solid state drive) and HDD (hard disk drive). Besides, GraphChi can process an ongoing in-flow of graph updates while performing advanced graph mining algorithms simultaneously, like Kineograph [44]. GraphChi uses space-efficient data structures such as a degree file that is created at the end of processing to save in-degree or out-degree for each vertex as a flat array. It also uses dynamic selective scheduling that lets *update function* and *graph amendments* to enlist vertices to be updated. It was extended later as a graph management system called GraphChi-DB [126] and tried to address some of these challenges.

Many other graph processing systems have been developed based on single machines. Signal/Collect [215], for example, is a vertex-centric framework made to improve the semantic web computational performance. In this model signals will be sent along edges where they will be collected in vertices. The advantage of this model is that it provides flexibility for synchronous, asynchronous and prioritized execution. Other systems such as RASP [252], X-stream [192] which provides an edge-centric framework, FlashGraph [262], Galois [165], TOTEM [77], BPP [162], etc. also make processing graphs possible on single machines using various computational models and processing systems.

Graph processing on a single machine would be easier to program and execute than on distributed systems if the entire graph fits within the local resources on that machine. This is because of efficient communication, simpler debugging and easier execution management on a single machine. But this limits their scalability beyond a certain graph size. New approaches for processing graphs on single machines are

targeting flash and SSDs [247] [122] whose speeds are matching main memory and offer advantages for graph processing [262] [167].

2.3.1.3. Heterogeneous Architecture

In a heterogeneous environment, not every processing unit is equally powerful [88]. This may be a single machine and additional on-board accelerators and specialized devices, or it can also consist of distributed, non-homogeneous systems. Because of this, we considered them as a separate group in this taxonomy. For example, processing systems such as RASP [252] and FlashGraph [262] have tried to optimize the storage part of the system by using SSD which is much faster and more reliable than traditional hard drives [75]. Many graph processing systems have proposed utilizing graphic processing units (GPU) alongside CPU for computation [234]. Medusa [263], for instance, was developed to make processing graphs using GPUs easier. Medusa is a programming framework that enables users to write C/C++ APIs to promote the capabilities of GPUs to execute the APIs in parallel. Its extended version also can be run on multiple GPUs within a single machine. Gharaibeh *et al* [77] developed a system called TOTEM that assigns the low-degree vertices to the GPU and operates high-degree vertices processing on the CPU. On the other hand, systems like CuSha [119] compute the entire graph on GPU. Another possibility is to exploit non-uniform VM sizes on Clouds for a distributed, heterogeneous architecture, which has been less explored.

Recently some research works have started exploring the use of field-programmable gate arrays (FPGAs). FPGA is an integrated circuit made of matrix of configurable logic blocks and their programmable connections that can be configured by the user after being manufactured. GraphGen [169] is a generic vertex-centric FPGA-based graph processing framework. It has been designed to get vertex-centric specifications and create FPGA implementations for targeted platforms. The problem with GraphGen is that it keeps the entire graph inside the on-board DRAM that limits the scalability of the system remarkably. FPGP [54] is another framework that enables interval-shared vertex-centric processing on FPGA. FPGP has also being used to analyze the performance bottleneck of other processing frameworks on FPGA. Although it has

been shown that FPGP does not perform as good as CPU-based single server frameworks, it shows the mechanism of FPGA-based generic graph processing systems well. Overall, FPGA is still a new research area in graph processing context compared to CPU and GPU based systems but it is getting more attention [70].

2.3.2 Graph Processing Frameworks

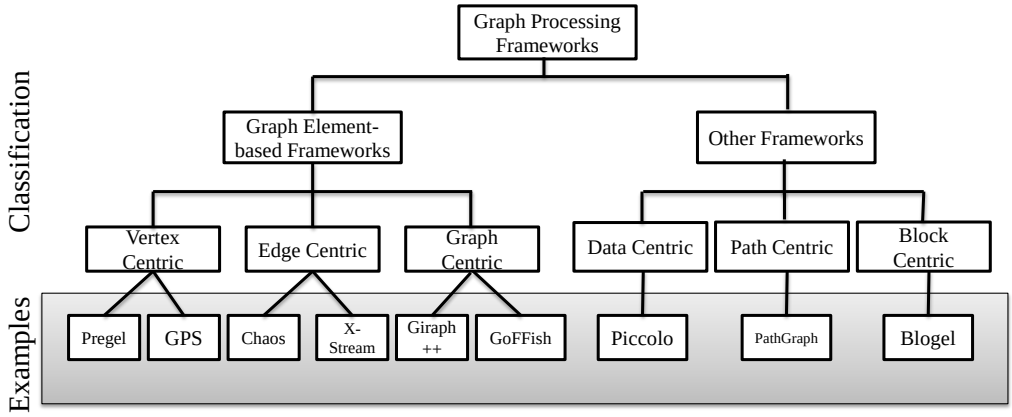


Figure 2-4 Taxonomy of programming models used by graph processing frameworks

Graph processing frameworks enable graphs to be processed on different infrastructures such as clusters and clouds. Here, we restrict ourselves to distributed memory systems that are designed for commodity, rather than High Performance Computing or Supercomputing clusters. The programming abstraction for each framework is designed either based on a graph topology element, such as vertices and edges, or other alternative approaches. Figure 2.4 depicts the taxonomy of graph processing frameworks according to main characteristics of the graph and other alternatives. We discuss these further below.

2.3.2.1. Vertex-Centric (Edge-Cut) Frameworks

Vertex-centric programming is the most mature distributed graph processing abstraction and several frameworks have been implemented using this concept [60]. A vertex-centric system partitions the graph based on its vertices, and distributes the vertices across different partitions, either by hashing them without regard to their connectivity [148] or by trying to reduce the edge cuts across partitions [195]. Edges

that connect vertices lying in two different partitions either form remote edges that are shared by both partitions or owned by the partition with the source vertex.

In the vertex-centric programming abstraction introduced by Google's Pregel [148], computation centers around a single vertex – its state and its outgoing edges – and interactions between vertices are through explicit message passing between them. This gives a fine-grained degree of vertex-level data parallelism that can be exploited for concurrent execution. Pregel's execution follows a bulk synchronous parallel (BSP) model, where vertex computation and inter-vertex messaging are interleaved, and the application iteratively progresses along barrier-synchronized supersteps. The Pregel API allows developers to focus on the vertex-centric graph algorithms while abstracting away communication and coordination details to the runtime. In Pregel, the domain of a vertex's user-defined *compute* function is restricted to the vertex and its outgoing edges, while LFGraph [98] considers incoming edges to be restricted. Figure 2.5.b shows vertex-centric processing approach for a sample graph shown in Figure 2.5.a.

A vertex-centric model makes programming of graph processing *intuitive and easy*, similar to the advantages of Map-Reduce for tuple-centric programming. Parallelization is done *automatically*, and *race conditions* on distributed execution are avoided. Primitives like *combiners* and *aggregators* are available for application-level message optimizations and global state exchange. The model also allows for *graph mutations*, where the structure of the graph can be changed as part of the execution (useful, for e.g., when iteratively coarsening the graph for partitioning, clustering or coloring).

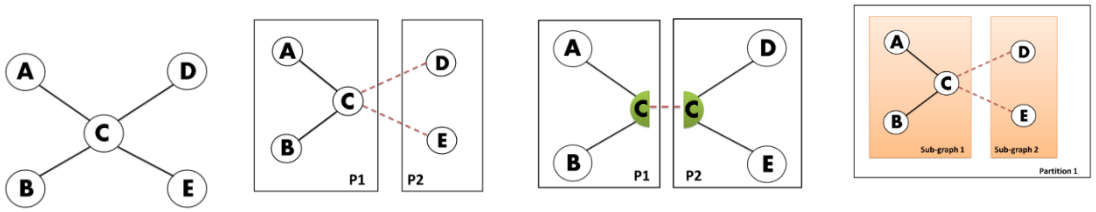
However, Pregel have several shortcomings: 1) While the vertex-centric model exposes parallelism at the level of individual vertices, which can be computed in negligible time, massive graphs can impose coordination overheads on this degree of parallelization that may out-weigh the benefits [223], 2) The number of barrier-synchronized supersteps taken for traversal algorithm can be proportional to the diameter of the graph with the number of message exchanges required between partitions also being high, proportional to the number of edges [208], 3) Mapping shared memory graph algorithms to this model is not trivial and requires new vertex-

centric algorithms to be developed [208] and 4) Using a vertex-centric programming model without regard to the graph partitioning and data layout on disk can lead to punitive I/O initialization and runtime performance [208]. These shortcomings have been addressed in some other vertex-centric frameworks such as GoFFish [208] and GPS [195].

Apache Giraph [11], is a popular open-source implementation of Pregel. Giraph uses Map-only Hadoop jobs to schedule and coordinate the vertex-centric workers and uses Hadoop distributed file system (HDFS) for storing and accessing graph datasets. It is developed in Java and has a large community of developers and users such as Facebook [105] [194]. Giraph has a faster input loading time compared to Pregel because of using byte array for graph storage. On the other hand, this method is not efficient for graph mutations which lead to decentralized edges when removing an edge. Giraph inherits the benefits and deficiencies of the Pregel vertex-centric programming model. Its performance and scalability is algorithm and graph dependent, and works very fast, for e.g., on stationary algorithms like PageRank but not as fast on traversal algorithms like single source shortest path (SSSP) [190] and weakly connected components (WCC) [196], particularly for graphs with a large diameter. However, the ease of use of this framework and the community support has made it a popular platform over which to develop other Pregel-like systems with feature enhancements to the vertex-centric concept.

Other distributed platforms like Apache Hama [13] and GraphX also offer a vertex-centric programming model, with features comparable to Giraph. GraphX, developed on top of Apache Spark, determines *transformation on graphs* where every operating action produces a new graph. This framework uses a programming abstraction called Resilient Distributed Graph (RDG) interface, which builds upon Spark's in-memory storage abstraction – Resilient Distributed Datasets (RDD). The graph in GraphX includes the *directed adjacency structure* along with *user defined attributes* connected to each node and edge, and both are encoded as RDGs. Using RDG, the implementation of frameworks such as Pregel and PowerGraph on Spark needs less efforts.

Pregelix [32] is a vertex-centric framework that tries to model Pregel as an iterative dataflow on top of the Hyracks [28] parallel dataflow engine. Pregelix has been developed to address three main challenges in distributed Pregel-like systems: 1) Many Pregel-like systems have limitations to support out-of-core vertex-storage, 2) Existing Pregel-like systems have specific strategies and implementations for communication, node storage, message delivery, and so on. Therefore, a user cannot choose between different implementation strategies based on what is better for a particular algorithm, dataset or cluster. Pregelix improves physical flexibility and scalability of the processing system to address this challenge, and finally, 3) Pregelix tries to leverage current data-parallel platforms to streamline the implementation of Pregel-like systems.



(a) A sample graph (b) Vertex-centric (edge-cut) approach (c) Edge-centric (vertex-cut) approach (d) Component-centric approach

Figure 2-5 Graph element-based approaches for graph processing frameworks

2.3.2.2. Edge-Centric (Vertex-Cut) Frameworks

In edge-centric frameworks, edges are the primary unit of computation and partitioning, and vertices that are attached to edges lying in different partitions are replicated and shared between those partitions. It means that each edge of the graph will be assigned to one partition, but each vertex might exist in more than one partition. Figure 2.5.c depicts this approach. While edge-based partitioning is more costly, this model shows better graph processing performance compared to vertex centric approaches [186]. However, programming an edge-centric system is more difficult than vertex-centric systems [253]. It is also important to create edge-balanced partitions in this method to load balance the computation across workers, just as vertex balancing is important for vertex-centric frameworks. Decreasing the vertex cuts has been investigated in some research [71] [26] [137].

[37] and [57] have suggested a vertex-cut method for distributed graph placement in hyper graph partitioning, where the edge-centric problem can be solved by converting

each edge into a vertex and vice versa. The motivation for developing vertex-cut frameworks is that systems such as Pregel and GraphLab [142] are effective for flat graphs but have shortcomings with graphs that follow a *power law edge degree distribution* due to low quality partitioning and vertices with high edge degrees. Real-world graphs such as social networks are such power-law graphs where a small set of vertices have high edge degrees that connect to a large part of the graph, e.g. *celebrities* in social networks. Partitioning and representing power-law graphs in a distributed environment is also difficult [135] [1].

X-Stream [192] is a well-known edge-centric system that processes out-of-core and in-memory graphs using a gather-scatter approach (Section 2.3.4). It bases this approach on the intuition that storage media such as solid state drives (SSD), main memory and magnetic disk perform significantly better with a sequential access to data than random access. The authors have implemented different algorithms on their system and observe that many of them can work on edge-centric mode. It can even return results from unsorted edge lists. However, it causes overheads when new edges are added to the graph. X-Stream is not suitable for very large graphs that do not fit onto the SSD, it wastes remarkable amount of bandwidth for certain algorithms and finally, X-Stream is not suitable for graphs and algorithms that require many iterations [253].

Chaos [191] is another edge-centric framework that is created based on X-Stream's streaming model. It introduces a scalable distributed framework that can be scaled from secondary storage to several hosts on a cluster. Unlike some other graph processing systems, Chaos does not strive to attain locality and load balance and claims that network bandwidth in a small cluster surpasses storage bandwidth. Instead, it is designed to partition the graph for sequential access on storage. So, it spreads data uniformly at random on the cluster's machines which are not necessarily sorted edge-lists. Chaos also utilizes GAS (Gather-Apply-Scatter) computation model (subsection 2.3.4) by which the edge-centric characteristic of its model is proven by iterating over edges and getting updated in the gather and scatter stages.

2.3.2.3. Component-Centric Frameworks

Component-centric approaches have been recently introduced, where components are collections of vertices and or edges that are coarser than a single vertex or edge. Tian *et al.* from IBM introduced a "Think Like A Graph" instead of "Think Like A Vertex" [223] abstraction after observing shortcomings in vertex-centric and edge-centric methods of graph processing. In their partition-centric view, they divide the whole graph into partitions and assign those partitions to machines for being processed. A *partition*, which is a collection of vertices and edges in the graph, forms the unit of computing. Figure 2.5.d shows this approach. Giraph++, based on Apache Giraph [11], implements this model and uses this coarse-grained parallelism. In contrast to vertex-centric model that hides partitioning and component connectivity details from users, Giraph++ exposes the partition's structure to the users to allow optimizations. So, the performance of the system depends on the partitioning strategy that is used and how effectively users exploit the access to the coarse components in their execution. On the other hand, communication within a partition is by direct memory access, which is faster than passing messages between each single vertex in vertex-centric model. This results in fewer network message passing and lower time of execution per iteration (superstep), with a reduction in the number of iterations needed for convergence. It also benefits from local asynchrony in the computation which means that vertices in the same partition can exchange their state and perform consequent computations to the extent possible in the same iteration.

Simmhan *et al* developed GoFFish [208] which has a subgraph-centric computation model to merge both the scalability and flexibility of vertex-centric programming approach with the extensibility of shared-memory subgraph computation. A *subgraph* (weakly connected component) is the unit of computation. A partition may contain one or more subgraphs, whereas each subgraph only belongs to one partition of the graph. Vertices in the same subgraph have a local path between each other, so existent shared memory graph algorithms can directly be exerted to each subgraph. This gives a programming and algorithm design advantage over partition-centric frameworks like Giraph++ that offer no guarantee on connectivity between vertices in a partition, while retaining the advantages of fewer iterations and shared-memory access of those

frameworks. Subgraphs, or vertices that span subgraphs, communicate by passing messages, similar to a vertex-centric model.

GoFFish consists of two major components: 1) A distributed graph oriented file system, *GoFS*, which partitions, stores and provides access to graph datasets in a cluster across hosts, and 2) A subgraph-centric programming framework, *Gopher*, which executes applications designed using the subgraph-centric abstraction using the Floe [206] dataflow engine on top of GoFS. However, subgraph-centric programming algorithms are vulnerable to imbalances in the number of subgraphs per iteration as well as non-uniformity in their sizes. The time complexity per iteration also can be larger since it often runs the single machine graph algorithm on each subgraph, even as it often takes much fewer iterations. The benefits are also more pronounced for graphs with large diameter, where algorithms tend to be several times faster than a vertex-centric equivalent, rather than small-diameter power-law graphs. GoFFish supports applications that operate on single property-graphs as well as on time-series graphs [209].

2.3.2.4. Other Graph Frameworks

In addition to the aforementioned programming abstractions, other alternatives have been developed as well. Several data-centric models offer a declarative dataflow interface to users to access and process data without needing to explicitly define communication mechanisms. For example, MapReduce provides a dataflow programming model that is popular for processing bulk on-disk data, but not for in-memory computations across multiple iterations, and applications do not have online access to the intermediate states. Piccolo [182], was developed in New York University as a data-centric programming method for writing parallel in-memory applications in several machines. It uses a key-value interface with a user-defined accumulator function that automatically combines concurrent updates on the same key. Like many other data-flow models such as Pig, Hive, Dryad [104] [254], Flume Java [39] and Swazal, developers in Piccolo operate at a higher level of dataflow programming abstraction but need to know the framework and system behavior well to leverage its scalability for different applications. For example, the programmer has to *a priori*

specify the number of partitions while creating a table. Further, these are tuple-oriented data flow models rather than graph specific ones.

Yuan *et al* [253] introduces PathGraph which aims to leverage memory and disk locality on both out-of-core and in-memory graphs using a path-centric approach. Their path-centric abstraction utilizes a set of tree-based partitions to model the graph and benefits from a path-centric computation instead of a vertex or edge centric computation. It means that the graph will be partitioned into paths including two forward and reverse edge traversal trees for each partition. It applies iterative computation per traversal tree partition in parallel, and then merges partitions by examining border vertices. Two functions, *gather* and *scatter* (Section 2.3.4), are used to traverse each tree by a user-defined algorithm. In addition to the computation tier, PathGraph has a path-centric storage tier to better the local accessibility for the computation. The storage structure is based on a tree partition and uses vertex-based indexing for tree-based edge chunks. The system outperforms the vertex-centric GraphChi [127] and edge-centric X-Stream frameworks.

Frameworks such as Blogel [248] adopt blocks as units of computation. Blogel introduces the concept of "think like a block" rather than "think like a vertex" and argues that existing systems do not address three main characteristics of real-world graph including 1) skewed degree distribution, 2) large diameter, and 3) high density. Considering these characteristics, the basic idea in Blogel is to put a high degree vertex with all its neighbors in one block and assign the whole block to one host. It also uses three computing mode (B, V, VB-mode) depending on the algorithm along with two different partitioners (graph Voronoi diagram partitioner and 2D partitioner).

2.3.3 Distributed Coordination

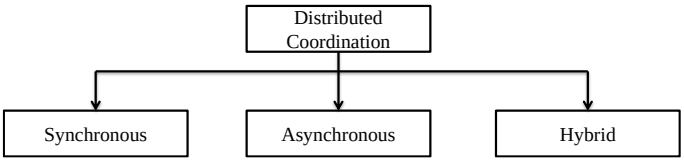


Figure 2-6 Distributed coordination

2.3.3.1. Synchronous

When a graph algorithm executes synchronously, it means that concurrent workers process their share of the work iteratively, over a sequence of *globally coordinated* and well-defined iterations. Synchronization may be applied to vertex-centric, edge-centric and component-centric models, and both on distributed and single machine systems. For example, Pregel-like systems call their barrier synchronized iterations as superstep, and workers coordinate their computation and communication phases in each superstep – everyone completes a superstep before starting the next. Initially, the master assigns partitions to the workers in the first iteration; the workers update their set of vertices based on the assigned partitions and wait for a global barrier, which tells them all workers are ready with their partition [148]. Subsequent supersteps indicate actual computation based on the application logic. Updated vertices in each partition send messages to (typically) vertices in neighboring partitions between iterations. Within an iteration, vertices can only access information about their local vertex's state and messages received from the previous iteration. Such a synchronized execution is possible even in a shared-memory system, across workers (threads, processes) on a single server.

These regular intervening periods make the system appropriate for algorithms where sizeable computation and communication can take place within each iteration since there is an overhead associated with the coordination. The bulk messaging at iteration boundaries can utilize the bandwidth efficiently if there is heavy communication between partitions [62] [240]. It is easy to program, debug and deploy such systems, without concerns of distributed race conditions and deadlocks. Another advantage of synchronous processing is that the outcome of each superstep is known immediately and provides real-time response of incremental application progress and easier error recovery in case at superstep boundaries. Synchronous execution is also suitable for balanced workloads that are computed symmetrically, with all workers having adequate work, so that the overhead of the global barrier and idle time for faster workers waiting for slower workers to synchronize is reduced [240]. These advantages make synchronous execution very popular such that several graph processing systems like Pregel, GPS, Kineograph, Mizan, GasCL [41] and Medusa [263]

use this model. Some like GoFFish [208] have two levels of such synchronized supersteps, an outer loop over different graphs in the context of time-series graphs, and an inner loop as supersteps over a single graph from the outer loop.

Synchronous execution model has some disadvantages as well that should be considered while designing or choosing a processing system for graphs. First, this model is not suitable for unbalanced workloads in which computation converges asymmetrically [218]. Likewise, if the distributed machines are not homogenous, the performance of the hardware may also cause some partitions to operate slowly. In such cases, it is possible to have stragglers when all partitions have been computed on workers except one slower worker which has not finished and hence delays all workers in the superstep. So, the runtime in this model is completely dependent on the slowest machine in each iteration [195]. Some of these shortcomings have been identified and addressed through elastic load balancing of partitions across workers [59]. Another drawback is that the intermediate processing updates between supersteps, in the form of messages or state, has to be retained in memory and this causes additional memory pressure [187]. A third disadvantage is that a synchronous execution model is ill-suited for applications and algorithms that need coordination between adjacent vertices [60]. For example, in a graph coloring algorithm in which vertices try to choose a different color from their neighbors, two adjacent vertices might pick conflicting colors frequently [80] [220] and the algorithm will converge slowly. Lastly, based on the drawbacks mentioned above, the cost for the systems that use synchronous model of execution is higher because the throughput must always remain high and running time would be longer [258].

2.3.3.2. Asynchronous

An asynchronous execution model does not have any global barrier and a subsequent phase of execution will be started on a worker immediately after its current iteration finishes its computation [240]. Hence, some of the challenges of load balancing and long tail computation in the synchronous model are addressed by asynchronous computation, where workers do not have to wait for the slowest worker to start their subsequent iteration. This approach is useful when the workload is imbalanced and

convergence can occur faster than synchronous approach. Therefore, we can say that asynchronous model is the preferred model when computation across workers is heavily skewed and there is little communication that can benefit from bulk operations [240]. In other words, this model is preferable for CPU-based algorithms while synchronous model would perform much better on I/O-bound algorithms. Another advantage for this model is that it can use dynamic scheduling to implement prioritized computation to execute more units of computation before others, to obtain better performance [259]. Normally, asynchronous execution provides more flexibility than synchronous execution by utilizing dynamic workloads which makes it outperform synchronous methods in many cases; however, the exact comparison between these two models depends on various properties of the input graph, platforms that the system has been deployed on, execution stages and applications [240]. Finally, using asynchronous approach provides a non-blocking process because resources could be free and become available for the next iteration, whereas in a synchronous approach, they are blocked until the global barrier declares the end of superstep which leads to a competition for resources at the beginning of next superstep.

As before, there are disadvantages to this model as well. The key disadvantage is that programming asynchronous processing systems is more difficult than synchronous systems. The programmer should deal with irregular communication intervals, unpredictable response time, complex error handling and more complicated scheduling issues. For example, for error recovery in such a system, many factors have to be considered: which machine has faced a fault, in which iteration of a particular worker the error happened, which resources caused the errors, should new resources be allocated to the computation or it should only be rearranged, and so on. This also results in more complex debugging and deployment, and careful programming to avoid deadlocks. In case of pull-based communication model (Section 2.4.2), which is usually implemented in an asynchronous manner, many redundant communication may happen because there are several intertwined reads and writes while adjacent vertices values do not change [92] [257]. On the other hand, regardless of these drawbacks, many single machine systems have preferred an asynchronous execution

approach since the shared memory makes it easier to asynchronously use the latest data without waiting for a barrier.

2.3.3.3. Hybrid

There are many systems that use only synchronous execution mode; for example Pregel, GoldenOrb [36], GBASE [112], Chronos [92] and GraphX [81], while many other systems utilize asynchronous mode like GiraphX [220], GraphHP [42], Ligra [204], RASP [252] and GraphChi. But recently a new approach called hybrid execution model has been implemented in a few systems that tries to take advantages of both asynchronous and synchronous approaches or incorporate them with new additional solutions. Such graph processing systems have been developed to improve system performance by overcoming the shortcomings of existing methods, and use both synchronous and asynchronous models of coordination to benefit from their relative strengths.

GRACE [236], for instance, is a single machine framework that combines synchronous programming with asynchronous execution features. It actually separates execution policies from application logic. In an asynchronous execution, a processing sequence of vertices can be intelligently ordered by dynamic scheduling to remarkably speed up the convergence of computation. GRACE uses the BSP computational model and message passing communication model as two primary paradigms of synchronous model. It helps GRACE to improve its automatic scalability by applying prioritized execution of vertices and receiving messages selectively outside of the last iteration. Various workloads like topic-sensitive PageRank, social community detection and image restoration have been used in GRACE and it shows comparable running time to other asynchronous systems such as GraphLab with even better scalability.

Another hybrid approach distinguishes between local vertices that are within a partition and remote vertices connecting across partitions. These types of systems exploit both local asynchronous computation when they still need global barrier for synchronization of remote vertices values [42]. In the local computation phase, messages will be passed very fast across local vertices using shared memory. In the next phase (synchronous phase), remote nodes (boundary nodes) that are connected by

edges across the partitions, will be updated by exchanging messages. In fact, component-centric frameworks such as Giraph++ and GoFFish allow users to exploit this. Others like Giraph Unchained [90] also allow incremental forward progress within a future superstep based on partial messages that are received, even before the previous superstep completes. These straddle synchronous and asynchronous models.

Apart from these methods, a novel approach has been introduced by Xie *et al.* [240] in the PowerSwitch system, which sequentially switches between synchronous and asynchronous execution mode according to a heuristic prediction. This is because some properties of the processing might change as it progresses. For example, processing single source shortest path (SSSP) algorithm begins with just a few vertices active, which means that few messages are passed; this is suitable for an asynchronous model. But as the process goes further, the number of vertices involved in the computation will increase which means that the number of messages passing among them increase as well, and this is suitable for synchronous execution model. PowerSwitch can effectively predict the proper heuristic for each step and it can switches between the two modes if required.

2.3.4 Computational Models

Performing computation is at the heart of a graph processing system where data (vertex or edge) will be processed and updated. Computational models that are used in existing graph processing systems can be divided into two major groups: 1) two-phase models, and 2) three-phase models. Figure 2.7 shows the classification of these two models with examples from each group. Computational model of some systems is a combination of these methods with other approaches.

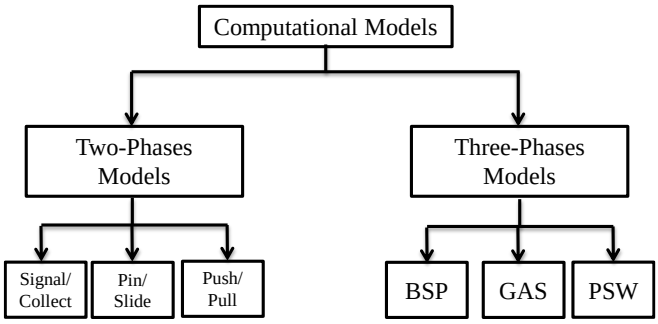


Figure 2-7 Classification of computational models in graph processing systems

2.3.4.1. Two-Phase Computational Models

There are usually two functions that are applied on data (vertices or edges) in a two-phase computation model. Signal-collect approach is the first two-phase programming model for large scale graph processing on the semantic web within a system with the same name (signal/collect) [215]. Computation in the vertices are completed by collecting the signals that are coming from edges and performing some processing on them using the vertex's state and then sending (signaling) their adjacent vertices in the compute graph. Signal/collect has been implemented for working with both synchronous and asynchronous execution models. In both models, some parameters should be set to determine when the computation should be stopped: *signal_threshold* and *collect_threshold* parameters in which a minimum level of "importance" will be set for the execution, and a *num_iterations* parameter which is the number of iterations in synchronous mode, and *num_ops* that is the number of executed operations in asynchronous mode. All these parameters should be set by the user.

Pin-and-slide was first introduced by TurboGraph [91] for parallel execution of large datasets on a single machine. A pin-and-slide mechanism consists of a graph dataset, a buffer pool and two different threads, as explained in Section 2.3.1, callback threads and execution threads. When the buffer manager is being sent an asynchronous input/output by a callback thread, it sends the demand to the FlashSSD via the operating system after when the control of execution goes back to the calling thread immediately. The main goal in this system is to reach all related adjacency lists efficiently. To achieve this goal, first, the pages that contain these adjacency lists should be identified. The most important challenge here is pinning large adjacency pages (LA pages) which means that a number of smaller adjacency pages must be unpinned first, then the LA page can be pinned. To overcome this challenge, LA pages will be pinned when all of related LA pages for a big vicinity list are completely loaded to maximize the buffer exploiting. When execution threads or callback threads terminated the processing of a page, this page will be unpinned and an asynchronous input/output demand will be sent to the FlashSSD by the execution thread. As soon as the

processing has been completed, the execution window can be slid by the size of the pinned pages in the buffer [91].

Nguyen *et al* [165] have used another two-phase approach called push/pull styles. The value of an active vertex will be pushed (flowed) from that vertex to its neighbors, which is more like scatter phase. The pull style occurs when the data from an active vertex's neighbors flow into that vertex which is more like a gather phase. In an algorithm like SSSP, the push-style is applied to the active vertex neighbors by updating the destination label of the siding nodes of the active vertex by doing a relaxation with them; and the pull-style function updates the destination label of the active vertex by doing relaxation with all neighbors of that node. The pull mode also needs less synchronization because there is just one writer for each active vertex. KineoGraph [44] is another system that uses this model for computation.

2.3.4.2. Three-Phase Computational Models

Bulk synchronous parallel (BSP) is a parallel programming and also the most representative model in this category that has been used in several graph processing systems [148] [195] [232] [118]. To deal with the scalability challenges of parallelizing tasks across a number of workers, BSP, which utilizes a message passing interface, was developed. In BSP, as a vertex-centric computational model, each node is able to have two modes of "active" or "inactive". The computation comprises a series of supersteps that come with synchronization hurdle between them. So, in each superstep: 1) The node that is involving in computation obtains its adjacent nodes updated values from the last superstep (except the first superstep), 2) Then, the node will be updated based on the obtained values, and 3) The node forwards its updated value to its neighbors that will be available for them in the next iteration. In each iteration, a vertex may choose to vote to halt, in case it does not receive any messages from its neighbors or it has reached a locally stable state. That means it will not participate in the processing anymore until it receives new messages that convert its state from inactive to active. So, in each iteration only active vertices will be computed. If there is no active vertex in the graph, then the computation is finished.

Some research have modified the BSP model and introduced new models. For example, temporally iterative BSP (TI-BSP) [207] is a computational model for time-series graphs on a subgraph centric model such as GoFFish. It has used BSP as a building block to support the design pattern. TI-BSP is a series of BSP loops (nested supersteps) in which each *outer loop*, as a *timestep*, runs on one graph instance in time. Supersteps using a subgraph programming model form the *inner loop* that operate over a single instance. As a result, the design pattern will be decided based on the order of timesteps execution and the messages between them.

There is another BSP model that stands for "BiShard Parallel" and has been introduced by the single machine based system, BPP [161], to empower full CPU parallelism for graph processing. This model has also three phases that include: 1) Loading a sub-graph of the large graph from disk, 2) Performing compute operation on the sub-graph and update the values of edges and vertices, and 3) Writing back the modified values on disk. BiShard Parallel performs under an asynchronous execution model and needs more disk space compared to one shard mechanism that was used in GraphChi, because two copies of each edge is managed in this model.

GAS (Gather-Apply-Scatter) is another three-phase computational model that was introduced by PowerGraph. The data about the adjacent nodes and edges is obtained and collected using a general summation over all adjacent vertices and edges of a vertex in the *gather* phase. The *apply* operation should be defined by user and must be both associative and commutative, and can vary from a numeric summation to the aggregation of data across all adjacent edges and vertices. The results from gather phase is used to update the central vertices values in the apply phase. Finally, the recent data of the central vertex is used to renew the values on neighboring edges in the *scatter* phase. The critical challenge in this model is that graph parallel abstractions should be able to perform computations with high fan-in and high fan-out where both of them are specified by gather/scatter phases. GAS model is used to develop a runtime system mapping in parallel on GPUs as a graph application called GasCL [41]. This model is like the one that have been used in systems such as Pregel and GraphLab, but in a different way.

GraphChi uses a different computation model called parallel sliding window (PSW). PSW is an asynchronous computation model that can efficiently process the graph with changeable edge value from disk, with a few number of non-consecutive disk access. PSW performs three phases for processing a graph as follow: it loads a subset of the graph from disk, then, applies update operation on vertices and edges, and eventually, the new updated values will be written on disk. The number of "reads" from disk is exactly equal to the number of "writes" to the disk in this model.

The comparison between two-phase and three-phase models shows that two-phase model is generally used in frameworks with single machine architectures. Since these systems usually use asynchronous coordination, using the two-phase approach, is more efficient as they do not need to wait for the synchronization barrier. On the other hand, three-phase approach is mostly used by distributed frameworks because one or two phases of the model will be affected after the global barrier. Therefore, hosts on such distributed systems have to wait for each other. However, very few distributed frameworks such as GraphLab [141] and Trinity [202] tried to use three-phase computation mode while utilizing asynchronous coordination.

2.4 Runtime Aspects of Graph Frameworks

2.4.1 Partitioning

Graph partitioning is a method in which graph data is divided into smaller parts with specific properties [34]. For example, in a *k-way* partitioning, the graph is partitioned into K smaller parts of equal size while minimizing the edge cuts between partitions. This is an NP-complete problem [8]. Graph partitioning is a fundamental research problem and several reviews have been done on different methods and perspectives of graph partitioning [34] [20]. In a graph processing system, partitioning is applied on the large graph in order to assign each smaller partition to a worker to be computed. The most important challenge in this context is *"how do we partition the graph to achieve better cuts while taking load balancing and simplicity of computation into consideration?"*.

Many novel heuristics have been proposed for partitioning large graphs. METIS [116], for instance, is a popular tool that uses multi-level partitioning. It is able to perform high quality partitioning that decreases the overall communication (edge cuts) and reduce imbalances across partitions. However, METIS is computational costly and high random access needs make it unsuitable for large graphs. ParMETIS is a parallel MPI-based version of METIS that mitigates some of these performance limitations.

There are distributed partitioning algorithms, some of which have been implemented on top of graph processing frameworks as well. Spinner [152], for instance, runs on top of Giraph and utilizes an iterative node migration approach using Label Propagation Algorithm (LPA) to deal with scalability and changing partitions. It allows Spinner to scale to billion-vertex graphs by avoiding costly synchronization among vertices. Blogel implements a *Graph Voronoi Diagram (GVD)* partitioner using a vertex-centric computing method by operating as a multi-source breadth-first search (BFS). It partitions the vertices into blocks using multi-source BFS over linear workloads.

Some graph processing systems create additional topological constructs on top of the partitioned graph. In GoFFish, which is a subgraph centric framework, each partition may have more than one subgraph (weakly connected component), and these subgraphs by definition cannot have an edge between them. So GoFFish has a post-processing stage once the graph is partitioned, in which it identifies all subgraphs within a partition that form the unit of processing during the programming model's execution.

In general, two partition creation strategies can help to improve the runtime performance during distributed graph processing: 1) Creating more partitions than workers and allocate more than one partition to each worker, and 2) allocating one partition per worker, yet using multiple workers on each processing host [195]. We next discuss alternative perspectives towards partitioning to support graph processing systems, also shown in Figure 2.8.

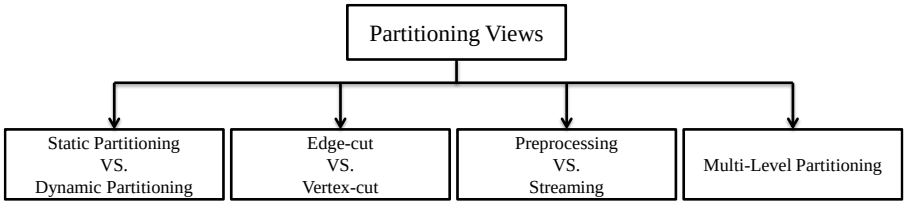


Figure 2-8 Partitioning views in graph processing systems

2.4.1.1. Static Partitioning vs. Dynamic Partitioning

Several graph processing frameworks utilize static partitioning which means that they consider the graph and the processing environment to stay unchanged [198]. These systems assume that the I/O bandwidth, latency, processing units and the graph itself are constant and predictable. So, this method of one-time *a priori* graph partitioning is easy to program and load balancing can be easily achieved, if the assumptions hold and the problem domain does not change [69].

On the other hand, dynamic repartitioning assumes that runtime behavior of an algorithm, the processing environment and even the graph itself can be variable. They try to repartition the current state of the graph according to the system and algorithm behavior at a given point in time, and assign them to the available workers. This approach has been used for graph databases and a number of graph processing systems [195] [166]. Dynamic repartitioning can be applied in-flight when, for instance, workers are waiting for a straggler worker to finish. In this situation, the vertices that have been assigned to the slowest worker can be repartitioned and reassigned to other idle workers to be computed faster and also balances the workload to reduce overall runtime. This does need support for dynamic migration by the graph framework [195]. Another reason to use dynamic repartitioning is when the number of active vertices in the graph change due to mutations or when the algorithm is non-stationary, causing vertices to become inactive, and it is suitable for iterative programming models such as Pregel [244].

According to GPS [195], three major questions should be answered in a dynamic repartitioning process: 1) Which nodes should be reallocated?, 2) When and how to transfer the reallocated nodes to their new workers? and 3) How to place the reallocated nodes? These decisions can impose a heavy cost and affect the overall run-time. Some researchers have also shown that dynamic repartitioning does not offer

significant performance improvements except under particular conditions. For example, the vertices in a PageRank algorithm are always active which makes dynamic repartitioning moot due to predictable and stationary load through the entire application's lifetime [143]. But, despite these concerns, systems such as GPS, xDGP [232], Mizan and XPregel [222] have incorporate dynamic repartitioning and migrate the vertices synchronously across the workers, along with their incoming messages.

2.4.1.2. Edge-cut Partitioning vs. Vertex-cut Partitioning

Vertex-centric (edge-cut) frameworks partition the graph by assigning vertices to partitions and cutting some edges across partitions, in the process, while minimizing the number of such crossing edges. This is a common and well-supported partitioning approach. On the other hand, edge-centric (vertex-cut) frameworks partition the graph by cutting vertices and assigning edges to each partition. This approach minimizes the number of crossing vertices which is useful for many real-world graphs that have a power-law degree distribution to balance edges across the partitions well [80] [1]. As was shown in Figure 2.10, vertices shared by edges belonging to different partitions would be cut and replicated across all the partition. One copy of the vertex is considered as the master and other copies are ghosts or mirrors. When updated, each ghost vertex sends its locally updated value to the master, and the master vertex applies updates from all ghost vertices to itself and sends the globally updated value back to the ghost vertices. We can see that many messages need to be passed across the network to maintain the cut vertices up to date.

To avoid this, PowerGraph does partitioning based on high-degree vertices of the graph and many systems have adopted such edge-centric partitioning [120] [186] [241] [107]. There are also a number of additional optimizations that have been done [186] [120]. Authors in [71] and [29] have investigated several edge-centric (vertex-cut) approaches with vertex-centric (edge-cut) approaches and found that in many cases the former outperforms the latter. The reason is that because the degree distribution is skewed, balancing the number of vertices in each partition does not guarantee workload balance; therefore, for natural graphs (that have power-law degree distribution), vertex-cut partitioning approach can obtain better workload balance.

They also conclude that for any edge-cut, a vertex-cut can be constructed directly which needs strictly less storage and communication.

2.4.1.3. Pre-processing vs. Streaming

As seen in Section 2.2.1, there might be a pre-processing phase before the computation or the main processing starts. In the pre-processing approach, the large graph which is present on disk, will be partitioned before entering the system. Single-machine frameworks such as GraphChi [127], TurboGraph [91], BPP [162] and CuSha [119] use this method because they do not have enough memory to keep all the processing states in the single system. So, they partition the graph before starting the processing to help cope with large graphs. It also limits the partitioning operation, which can be costly, to a single time. Distributed frameworks like GoFFish do partitioning and subgraph identification in such a pre-processing phase for the same reason.

In streaming partitioning, the graph is partitioned once or as it is loaded into the graph processing system. The graph data enters the system sequentially, say a vertex and its adjacency list at a time, and the vertex is mapped to a partition on the fly. In this model, the order in which the vertices enter the system is important as each placement depends on the previous placements [212]. Streaming can also benefit from a pre-processing model of partitioning, where specific vertex or edge ordering has been performed. *Random partitioning*, *round-robin* and *range algorithms* are the three most common algorithms for steaming whereas linear deterministic greedy (LDG) [212] and FENNEL [228] are two greedy heuristics that improve the performance and quality of such online partitioning.

2.4.1.4. Multi-level Partitioning

Some graph processing frameworks have proposed multi-level approaches for partitioning the graph. In this method, there will be more than one strategy for partitioning that may even be applied to the graph in different times. GridGraph [268] for example, is a single-server block-based framework that uses a two-level hierarchical method to partition a given graph. First, it partitions the graph once at pre-processing phase in which it divides the graph into 1D-partitioned vertex and 2D-partitioned edge

chunks respectively. Then at the runtime, it uses a dual sliding windows approach to partition the graph by streaming the edges and apply updates on vertices which guarantees the locality and improves I/O.

GraphMap [132] is a distributed framework that also uses a two-level partitioning mechanism to improve the locality and workload balancing. In the global level, GraphMap utilizes a hash method to partition the graph and assign the partition blocks to the workers; and in the local level, it applies range partitioning to each block partitions of a worker. It has also been designed to use other partitioning techniques such as METIS and ParMETIS both on-disk and in-memory.

Using multi-level partitioning has two edges. It can worsen the performance of the system by prolonging the execution time and unnecessary computations, or it can improve the performance particularly when it is applied as a layered mechanism. For example, apply one partitioning technique to the entire graph and at the same time performing another technique on the partitions on workers which can be designed asynchronously.

2.4.2 Communication Models

Graph processing systems use different approaches to communicate among their vertices, edges and partitions. In this Section we discuss these methods as shown in Figure 2.9 and enumerate the advantages and disadvantages of each method.

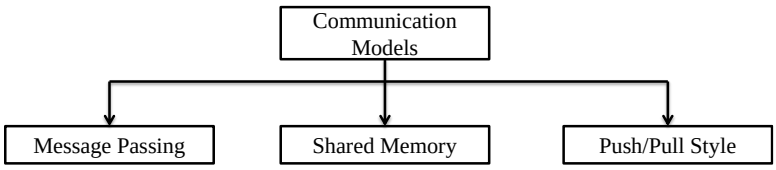


Figure 2-9 Communication models in graph processing systems

2.4.2.1. Message Passing

Many distributed graph programming models offer explicit or implicit communication between their entities. In the message passing technique, communication is carried out by sending messages explicitly from one entity to another in the graph. The entity can be a vertex, edge or a component in a local or remote partition [148]. Message passing is performed in many graph processing systems using communication libraries. For

example, Pregel allows developers to pass messages from a vertex in the graph to another by calling an API. As part of the BSP execution model, messages sent in one iteration are received by the destination vertex in the subsequent iteration using bulk messaging. The receiving vertex updates its state based on incoming messages and sends its modified state to one or more of its neighbors by sending additional messages. Each source vertex maintains a list of its adjacent vertex IDs or outgoing edges IDs. Further, vertices also have queues where incoming messages from its neighbors and outgoing messages to its neighbors are stored between superstep boundaries.

A message passing model of communication is used by many graph processing frameworks and architectures, including vertex-centric, edge-centric and component-centric frameworks. Programming using synchronous message passing is also intuitive and the complexity is limited to an API to send a message to a destination entity, which is a familiar model for many programmers [100]. Although message passing is common in the frameworks with synchronous model of execution, it can be used in asynchronous execution models as well. Asynchronous message passing method is used extensively between vertices in the same partition or subgraph where they do not need to wait for other vertices to send their message in-bulk. Vertices and subgraphs can communicate asynchronously while communication between partitions can be synchronous. However, the asynchronous model of message passing brings more complexity to the programming paradigm because it requires more resources for storing and rebroadcasting data in a system where its components do not execute concurrently [74]. On the other hand, buffer management is an important issue that should be considered by message passing implementation. Issues like: How many buffers should each worker have? What should be the size of each buffer? When should a buffer block a sender? And what if the buffer is full but there are new messages coming? This model also has overheads because of the number of message replicas that exists in the network.

The Message Passing Interface (MPI) [68] is a common protocol used in graph processing systems, and is used by systems such as Pregel, GraphLab, Piccolo and Mizan. Portability and velocity are two significant advantages of using MPI where

creating overhead is its most noticeable disadvantage. Communication can be done by passing actual messages between servers or by serialization. For Java based systems like Giraph and Hama, protocols such as Thrift[4] and ActiveMQ[5] can be used for message passing. They utilize remote procedure call (RPC) to communicate seamlessly without the need to change the messages structures. Also protocols such as Avro [6]and Protocol Buffer [7]can be used for serialization by which the data will be serialized to be able to be sent between different platforms.

Systems also propose optimizations on top of these standard messaging libraries to reduce the communication overhead and minimize the runtime of the algorithm by reducing the number and size of messages that are passed. For example, GasCL has considered two different message buffering strategies: first, messages from the same source are stored together, second, messages for the same destination are stored together. The second approach is more common and when a message is dispatched from the origin, it is instantly put down to the right target node. It also uses a reverse edge index to store the message which utilizes array offset to facilitate message combining. Giraph++ [223] introduces two types of messages in its hybrid model. "Internal messages" that are messages sent from a vertex within a partition to another vertex within the same partition, and "External messages" that are messages sent from the vertices of one partition to another partition. It provides two incoming message buffers for each vertex inside a partition as well: $inbox_{in}$ for internal messages and $inbox_{out}$ for external messages. In GPS [195], instead of sending multiple copies of the same message to multiple vertices in another partition, the system only sends one message to the remote partition and then, in the remote partition, the message will be copied to the vertices that need to receive it. This can dramatically reduce the network traffic.

[4] https://thrift.apache.org/
[5] http://activemq.apache.org/
[6] https://avro.apache.org/
[7] https://github.com/google/protobuf

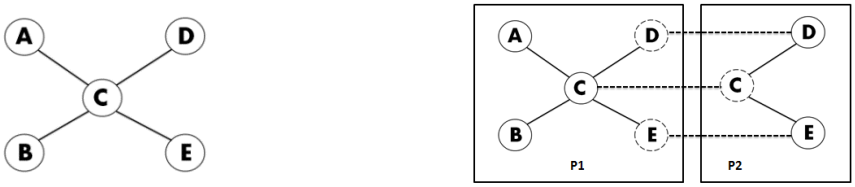
2.4.2.2. Shared Memory

Using shared memory for communication is well suited for frameworks running on a single server, but can also be used for distributed systems in place of explicit message passing. In this model, the memory location can be simultaneously accessed by multiple processing modules, including both read and write to that location. In contrary to message passing, the shared memory method avoids extra memory copying and intermediate buffering. As single machine frameworks have limited memory and CPU resources, this shared-memory model that is often natively supported by the operating systems is preferred [168]. Locks or semaphores are usually used in this model to prevent race conditions because concurrent tasks can read and write to the same memory [141] [45]. To maintain memory consistency, shared memory machines provide mechanisms for invoking the appropriate job (sequential consistency) or reordering a collection of jobs to be executed consecutively (relaxed consistency).

In distributed systems, it is also possible to have a distributed shared memory, where changes to memory locations are internally transfer using messages between different machines. From the programmer's points of view, they only perform memory accesses and the development is easier as explicit messages need not be passed. The concept of data "ownership" is lacking as well since anyone can write to that location. On the other hand, data locality cannot be controlled easily and if many remote workers access a particular memory location, it puts pressure on the processor and memory holding that location and can also lead to higher bus traffic and cache misses.

Virtual shared memory can be realized by using ghost vertices or mirrors which are the copies of distant adjacent vertices [142]. One machine keeps the main vertex and another machines work on copies of this vertex. Main vertex and ghost copies are shown in Figure 2.10. By keeping the mirror copies immutable during the computation with distributed write locking or an accumulator, the consistency is maintained [141] [182]. Both GraphLab and PowerGraph use this approach for communication. In particular, this is suitable for edge-centric frameworks because vertices should be cut in these frameworks and the partitioning is done based on edge-grouping. So, vertices can be easily cut (Figure 2.10). Trinity [202] is another memory-based graph processing

system (Section 2.4.3) that uses ghost vertices for communication. The Trinity specification language (TSL) maps the data storage and graph model together. The parallel boost graph library (PBGL), is a parallel graph processing library also uses ghost nodes but with message passing mechanism [205].



(a) Sample graph (b) Shared memory model

Figure 2-10 Shared memory model with ghost (mirror) vertices

2.4.2.3. Push/Pull Styles

Pull/Push model is used with active messages. Active messages are those that carry both data and the operator that should be applied on them [92] [257]. This model is utilizing by [24] direction optimization in breadth-first search (BFS). The reason behind using this model is that the synchronization and communication in large-scale data is very expensive due to the poor-locality and irregular patterns of communication in graphs. To reduce the random communication and memory access on either shared-memory or distributed implementations, Beamer incorporates the conventional *top-down* BFS with a new *bottom-up* method. In the push style, the information flows (is pushed) from an active vertex to its adjacent vertices and in the pull style, the information flows (is pulled) from the neighbors of an active vertex to that active vertex. This kind of communication is using the push/pull computation model that was discussed in Section 2.3.4.1. In terms of consistency, the pull style is naturally consistent because the active vertex would be updated in this phase, but the push style needs to use locks for every neighbor's update. Active messages are sent asynchronously in this model and they will be executed when they are received by the destination vertex. The sending and receiving messages are even combined in a framework such as GRE [251] and it does not need to save intermediate states anymore. This mechanism can help in enhancing efficiency of algorithms such as PageRank [78]. It has been used by frameworks such as Ligra [204] and Gemini [267] on shared memory and distributed processing, respectively. A detailed analysis of push/pull approach has been provided in [27], investigated the impacts of both

push/pull mechanisms individually and also together on various graph algorithms and programming models. They have illustrated that a *push/pull (PP) dichotomy* approach can avoid extreme amount of locks in pushing and more memory access in pulling.

2.4.3 Storage View

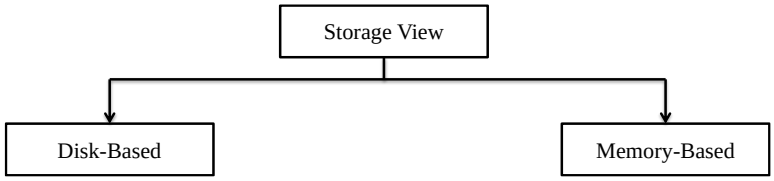


Figure 2-11 Storage view

Memory has become less of a problem these days as computing service providers such as Amazon are starting to provide machines with Terabytes of memory. However, as described in [203], social networks such as Facebook and Twitter not only have graph of users but also graph of connections between users, their likes, their locations, their posts and shares, photos, etc. that are heterogeneous. As a result, to store all these large graphs and datasets, a single server cannot provide enough space. Hence, many investigations have been done to process graphs on both single server and distributed environments such as clouds.

2.4.3.1. Disk-based

According to a storage view of execution models, two common approaches can be used: 1) *disk-based* approach, 2) *memory-based* approach. A disk-based execution fetches the graph data from physical disks, not just when loading the graph initially but also actively writes and reads parts of the graph state to/from disk during the execution. The advantages of using a disk-based approach is that it is cheaper to add disk capacity rather than memory, some large graphs do not fit in distributed memory either, and one can persist the partial state of execution in the middle of the processing to enable recovery from faults [50]. Disk management is also easy, so many graph processing systems use this approach (Table 2.3).

On the other hand, the computation that is performed by graph algorithms is data-driven [144], and they need many random data accesses and hard disks are still slow and inefficient compared to main memory. One of the challenging issues for disk-based graph processing is how to make disk access more efficient. BPP (BiShard Parallel Processor) [161], for example, provides a disk-based engine for processing large graphs on a single server. A novel storage structure called BiShard (BS) has been introduced which divides the graph into subgraphs containing equal number of edges and stores the in-edges and out-edges independently. This technique decreases the number of non-sequential I/O considerably and has two advantages compared to single shard storage mechanism that is presented by GraphChi. First, by storing in-edges and out-edges separately, access to each of them becomes independent and the system does not need to read the whole shard for every subset of vertices. Second, each edge has two copies in BS (one in each direction) which eliminates race condition among vertices to access their edges. Furthermore, BPP uses a novel asynchronous vertex-centric parallel processing model that leverages BS to provide full CPU parallelism for graph processing.

Other frameworks such as Giraph also support out-of-core execution using disk. When a graph is too big to fit into main memory (like small clusters) or a certain algorithm creates very large message sets (many messages or large ones) these frameworks can spill the excess messages or partitions to disk, later to be incrementally loaded and computed from disk. In addition, some frameworks such as FlashGraph [262] and PrefEdge [167] use SSD instead of HDD to make data transmission and computation faster for out-of-core computation.

2.4.3.2. Memory-based

In the memory-based approach, the graph and its states are exclusively stored in memory during runtime for storing and processing the big data. For example, Giraph runs the whole computation in memory and reaches the disk for checkpointing and I/O; and Blogel keeps all neighbors of a typical high-degree vertex in the same block to be processed by in-memory algorithms and avoid message passing. The most important benefit of this approach is that using RAM or cache for processing is much

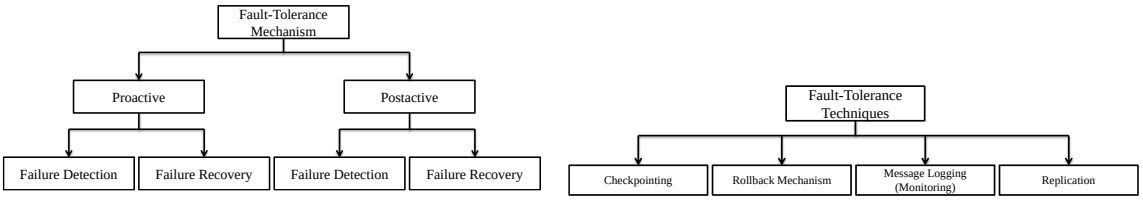
faster than disk-based approach since the CPU can access memory much quicker than disk [158]. However, memory is much more expensive than spinning disks and this becomes challenging when we consider larger graphs. So, memory-based systems must be efficient in retaining only relevant data in memory and in a compact form. Memory-based models also have less scalability than disk-based models, especially in a single machine system.

In GoFFish, the framework only loads a subset of properties for a given property graph from disk into distributed memory based on those attributes that are used by the algorithm, in addition to the complete topology of the graph that is always loaded [108]. This limits the memory footprint of the graph application during runtime. Many memory-based systems also use columnar representation since this offers a compact representation of data. Zhong and He [264] have indicated that GPU acceleration cannot reach considerable speedup if the data has to be loaded from disk because of the I/O costs that are themselves comparable to the total query runtime.

Microsoft's Trinity [202] is a distributed graph processing engine over a memory cloud. It is supporting both online query processing which requires low latency (finding a path between users in a social network), and offline query processing which requires high throughput (PageRank). Trinity uses TSL (Trinity specification language) for communication that supports both synchronous and asynchronous modes. It stores objects as blobs of bytes that is economical, compact, with no serialization and deserialization burden. As a storage infrastructure, it structures the memory of numerous hosts to a distributed memory address space which is universally addressable, for maintaining huge graphs. Trinity has three main components including: 1) *slave* that stores graph data and computes on them, 2) *client* that acts as a user interface between Trinity and the user that communicates with Trinity proxies and slaves through the APIs provided by Trinity library, and 3) *proxy* that is an optional component for handling messages as a middle tier between clients and slaves. These components, along with other features like user-defined communication protocol, graph schema and computation models through TSL, enable Trinity to process the graph efficiently on memory cloud.

2.4.4 Fault Tolerance

Fault tolerance enables a system to continue performing properly even if some of its components face failures [66]. Since graph processing systems are created from distributed and commodity components, it is possible that components confront failures which in turn will affect the execution and correctness of the applications. In order to improve the reliability and robustness of these systems, several techniques have been developed to support error handling and fault tolerance of the graph framework. Figure 2.12 shows the techniques that are used in many graph processing systems.



(a) Fault-Tolerance Mechanism (b) Fault-Tolerance Techniques

Figure 2-12 Fault-tolerance in graph processing systems

Error handling in a graph processing system, as with other systems, has two main phases: 1) *failure detection* in which the system discover the error, and 2) *fault recovery* in which the system tries to resolve the problem and resume the operation. Several researches have been done on various fault-tolerance techniques on parallel and distributed systems [117] [227] [67]. In [227] for example, two types of components in an application, called central components and parallel components, are investigated where both mostly use rollback and replication methods for fault recovery. On the other hand, some graph processing systems do not support any error handling because it increases the complexity of the system, and the overheads can strongly affect the execution time.

Most graph processing systems use *checkpointing* and *rollback* mechanisms [64] for failure recovery, such as Pregel and Pregel-like systems like Giraph. Pregelix [32], for instance, checkpoints states to HDFS at any superstep boundary that is selected by the user. The checkpointing applies to vertices and messages at the end of each superstep and ensures that the user does not need to know anything about the failure. Whenever a host or disk failure occurs, the unsuccessful machine will be added to a blacklist. For

recovery, Pregelix reloads the state of the latest checkpoint to a set of "failure-free" workers that is periodically updated.

Piccolo [182] uses a global checkpoint-restore method to recover from failures by providing synchronous and asynchronous checkpointing APIs. Synchronous checkpoints are suitable for iterative algorithms such as PageRank where the state in different iterations are decoupled by global barriers and it is adequate to checkpoint the state every few iterations. But asynchronous checkpoint is used to save the state of long running algorithms, such as distributed crawler, periodically. Piccolo also utilizes Chandy-Lamport (CL) distributed snapshot algorithm [40] for checkpointing. Once a failure is detected in a worker, the master will reset the status of all other machines and recover the operation from the latest finished universal checkpoint. The interior status of the master will not be checkpointed in Piccolo.

PowerGraph is another system that uses snapshots of the data-graph for fault-tolerance. The synchronous engine in PowerGraph creates the snapshot at the end of each superstep and before the start of the next superstep while the asynchronous engine suspends the execution of the system to create the snapshot. Many of these systems provide task rescheduling after the recovery phase. Some systems such as Pregel, Piccolo and GraphLab benefit from rollback which allows them to continue the computation from the point that failure happened while in a number of systems, fault recovery is not completely provided and they need to restart the processing from scratch [92] [124] [181]. All these mechanisms are post-active fault tolerance approaches which means they handle the failure after it has happened.

Trinity [202] uses message logging and replication for pro-active fault-tolerance, where the failure will be considered before scheduling and releasing a job for execution. Trinity utilizes heartbeat messages to proactively detect failures in machines. In addition, machines that unsuccessfully try to access the address-space in other machines also report the inaccessible machine to the master, and await for the addressing table to be updated before retrying the memory access. Meanwhile, in the recovery phase, the master reloads the data in the failed machine to another machine, updates the addressing table and distributes it. Trinity provides checkpointing after every few iterations for synchronous BSP-based computations and provides "periodic

interruption" mechanisms to generate snapshots in asynchronous computations. Buffered logging approach that is suggested in RAMCloud [171] has been used in Trinity to recover from failures in online queries while for read-only enquiries it only restarts the faulty machine and load the data again from the steady disk storage. GraphX [81] uses lineage-based fault tolerance that assumes its RDDs cannot be updated but only created afresh. It has a very light overhead compared to the systems which use checkpointing as well as arbitrary dataset replication. So, it attains fault-tolerance without explicit checkpoint recovery while the retaining in-memory performance of Spark.

2.4.5 Scheduling

Scheduling techniques help assign and manage jobs on the system resources [180]. This is particularly useful in parallel and distributed multitasking systems in which several computations have to be done on a limited number of resources. In graph processing systems with large scale graphs having billions of vertices and edges, the vertex or edge (depending on the programming model) will need to be scheduled for computation on a processing host. Typically, collections of vertices or edges are grouped into a coarser unit for scheduling, such as a partition or a subgraph, and it is the coarse unit that is actually scheduled on a CPU core. Within a processor, there may be multiple threads that execute individual vertices or edges in partition, leveraging, say, vertex level parallelism in a vertex-centric model.

According to [60], three different types of scheduling methods have been used for graph processing in general that is shown in Figure 2.13.

In batch scheduling method, the entire graph would be scheduled for processing across computing resources. This is more beneficial in bulk iteration model of computing such as Pregel. There is no priority or precedency in executing the partitions of the graph and they will be processed in any arbitrary order [47]. This model has been used widely in dataflow frameworks such as Hadoop, Haloop [33] and Twister [65]. There is always a preferred situation, like a limited number of iterations that is used as a condition for finishing the process.

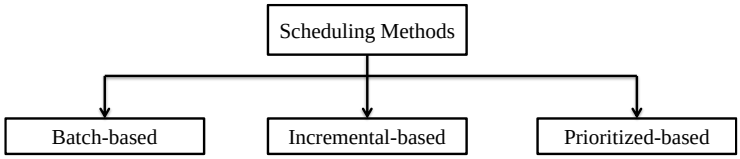


Figure 2-13 Graph processing scheduling methods

Scheduling can be done once at the beginning of the application or redone at the start of each superstep. For e.g. Giraph allows partitions of the graph to be mapped to workers both at the start of the application and at each superstep, while TOTEM maps partitions to workers at the start of the application and retains that mapping. Remapping of partitions to workers as the application is executing also requires the ability to migrate both the graph and its updated state and messages to different workers. The mapping of vertices and edges to partitions may also change as a result. For example, Mizan migrates the vertices from busy workers to the one with fewer vertices to load the balance, and GPS repartitions the graph to distribute the load among idle workers during the computation.

Another aspect is on whether the partitions are mapped to a static set of compute resources or the resources themselves can be elastic over the execution of the application. For example, [59] looks at mapping partitions to an elastic set of VMs based on the expected computational complexity of the partition for stationary and non-stationary graph algorithms.

In contrast, in prioritized scheduling method, jobs will be processed according to a priority condition that is defined by the user. System such as Maiter [258] shows that using this method results in quicker convergence for many graph algorithms. For instance, a defined prioritizing function can schedule jobs based on the number of vertices in each partition. In GoFFish, the largest subgraphs in a partition are executed first so that the computing of smaller subgraphs can be interleaved with the message passing from the large subgraph. Prioritized scheduling can be helpful in processing imbalanced workloads.

Doekemeijer and Varbanescu [60] believe that incremental scheduling only processes a subdivision of data like active vertices. This model is used in a number of graph processing systems, e.g., Stratospher [4] and GraphLab [141], in which the processing continues until there are active vertices.

2.5 Graph Databases

A graph database is one where the data is natively stored as a graph structure that can be queried upon [9]. The data itself is typically a property graph with not just vertices and edges but also name-value properties or labels defined on vertices and edges. Graph query models support different types of traversal queries such as path, reachability and closure, in addition to filter queries over their properties [108] [197]. Graph databases contrast with relational databases that store graphs – the latter require multiple joins for traversal of graphs rather than having direct references from a vertex to its neighbors that allows for faster query processing in graph databases. Graph databases leverage the topological properties of graphs, including graph theory and query cost models, in answering the queries. The queries also provide a high-level declarative interface for processing and accessing the graph compared to a graph framework that requires users to write a program using their graph programing abstractions and executes it in batch [234]. Another distinction from the graph frameworks discussed above is the need for low latency (*O(seconds)*) execution of hundreds of queries rather than high throughput analysis of single programs over large graphs. The aim of this section is not to provide a survey on graph databases, but to emphasize the increasing popularity of graphs and graph databases which provide a broader view of graph usages. For a detailed survey on graph databases, we refer the readers to works that appeared in ACM Computing Survey [9], VLDB Journal [115].

Graph datasets are receiving more attention every day and several companies are starting to utilize graph databases to perform interactive queries to support their business needs. Even traditional database provides such as Microsoft has added extensions to its product by which the graph will be natively stored and queried inside the database on Azure SQL DB[8]. According to DB-Engines[9], which is an industry observer, *"graph DBMSs are gaining popularity faster than any other database categories"*, that shows remarkable growth in the last few years (Figure 2.14).

[8] https://blogs.msdn.microsoft.com/sqlcat/2017/04/21/build-a-recommendation-system-with-the-support-for-graph-data-in-sql-server-2017-and-azure-sql-db/

[9] DB-Engines Ranking Per Database Model Category. (2015, August 5). Retrieved August 15, 2015, from DB-Engines: http://db-engines.com/en/ranking_categories

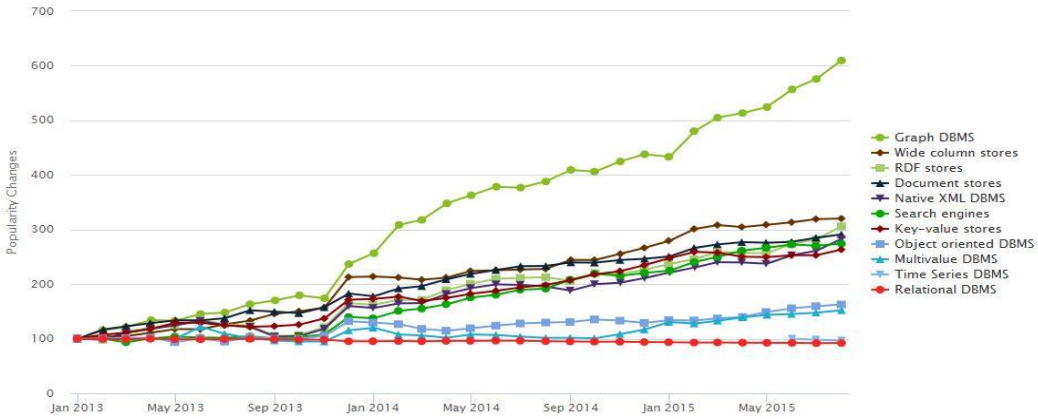


Figure 2-14 Popularity changes in using databases

Neo4j [164] is a popular graph database designed as an open-source NoSQL database. It supports ACID (Atomicity, Consistency, Isolation, Durability) properties by implementing a Property Graph Model efficiently down to the storage level. It is useful for single server deployments to query over medium sized graphs due to using memory caching and compact storage for the graph. Its implementation in Java also makes it widely usable. Besides the single server model, it also provides master-worker clustering with cache sharding for enterprise deployment. However, according to some reports, the scalability of the distributed version is not as good as even relational databases and it has deadlocks problems such as not being able to handle two concurrent *upserts* if they touch the same node[10].

OrientDB and Titan are two other well-used graph databases [170]. OrientDB can save 220,000 records per second on ordinary hardware. It supports multi-master replication and sharing which give it better scalability. It also provides a security profiling system based on roles and users in the database. Titan [225] is another open-source distributed transactional graph database that provides linear elasticity and scalability for growing data, data distribution and replication for fault-tolerance and performance. It supports ACID and different storage back-ends such as Apache Hbase [14] and Apache Cassandra [10]. Titan also uses the Gremlin query language[11] in which

[10] https://news.ycombinator.com/item?id=9699102
[11] http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html

traversal operators are chained together to form path-oriented expressions to retrieve data from the graph and modify them.

Twitter has developed its own graph database called FlockDB [230] to store social graphs such as "who blocks whom" and "who follows whom". FlockDB is an open-source fault-tolerant distributed graph database which aims to support online data migration, add/delete/update operations, complicated set of arithmetic queries, replication, archive/restore edges and so on. In April 2010, the FlockDB cluster had stored more than 13 billion connections (edges) and supported a peak traffic of 20K writes per second with 100K reads per second [84]. But it appears that FlockDB is not able to traverse graphs deeply as it is designed to only deal with Twitter's single-depth following/follower model and is not implementing the full stack of storage services[12].

There is also research on distributed graph databases, though this is an emerging area. Horton+ [197] from Microsoft offers a graph query language that supports path, closure and joint queries over property graphs. It converts the query into a Deterministic Finite Automaton (DFA) that is executed over a distributed database using a vertex-centric BSP model based on Giraph. GoDB [108] is another research database that leverages GoFFish to offer similar query capabilities over property graphs, but with support for scalable indexes and using a subgraph-centric model of execution that offers a much faster performance relative to Titan and Horton+. GBASE [112] introduces *compressed block encoding* graph storage method that utilizes adjacency matrix representation to store homogeneous regions of graphs. It also uses a grid-based selection strategy for query optimization to provide quicker answers by minimizing disk accesses. Quegel [249] handles enquiries as "first-class citizens" by which the user is only required to determine the Pregel-like algorithm for a general enquiry. Then, it sets up the computing and processing of multiple inbound enquiries on demand.

There are several other open source and commercial graph databases such as HyperGraphDb [101], AllegroGraph [5], InfiniteGraph [102], InfoGrid [103], JCoreDB

[12] http://stackoverflow.com/questions/2629692/how-does-flockdb-compare-with-neo4j

Graph [110], ArangoDB [17], GraphBase [83], MapGraph [150] [73], and Weaver[13]. All these projects try to provide modern solutions for storing and retrieving large-scale graph data and it seems that this area is a very promising field of research and commercial investments for the future. Jouili and Vansteenberghe [111] have compared some of these graph databases.

2.6 System Classification And Gap Analysys

Table 2.3 presents the key graph processing systems with their characteristics according to the proposed taxonomy. The notations in the table for each category are as follow:

- Programming model: vertex-centric (V), edge-centric (E), component-centric (C), path-centric (P), data-centric (Da) or block-centric (B).
- Architecture: distributed (D), single machine (S) or heterogeneous (H).
- Computational Model: different names are used by different systems
- Communication Model: message passing (MP), shared memory (SM) or dataflow (DF)
- Coordination: synchronous (Synch), asynchronous (Asynch) or both timing approach together
- Storage: disk-based (DB) or memory-based (MB) storage approach.
- N/A means that there is no specific name or method mentioned by the paper that is describing the system.

[13] http://weaver.systems/

Table 2-3 Overview of existing graph processing frameworks

Year	System	Programming Model	Architecture	Computational Model	Communication Model	Coordination	Storage
2009	PEGASUS [114]	N/A	D	N/A	DF	Synch	DB
2010	Pregel [148]	V	D	BSP	MP	Synch	DB
2010	Signal/Collect [215]	V	S	Signal/Collect	MP	Both	DB
2010	Surfur [46]	V	D	Transfer-combine	MP	Synch	DB
2010	JPregel [183]	V	D	BSP	MP	Synch	DB
2010	GraphLab [142]	V	S	N/A	SM	Asynch	DB
2010	Piccolo [182]	Da	D	Three phases	Dataflow	Synch	DB
2011	GoldenOrb [36]	V	D	BSP	SM	Synch	DB
2011	GBase [112]	E	D	N/A	Dataflow	Synch	DB
2011	HipG [124]	V	D	BSP	SM	Both	DB
2012	Giraph [11]	V	D	BSP	MP	Synch	DB
2012	Distributed GraphLab [141]	V	D	GAS	SM	Both	DB
2012	KineoGraph [44]	V	D	Push/Pull	MP	Synch	MB
2012	PowerGraph [80]	E	D	GAS	SM	Both	DB
2012	Sedge [250]	V	D	BSP	MP	Synch	DB
2012	GraphChi [127]	V	S	PSW	SM	Asynch	DB
2013	TOTEM [77]	V	H	BSP	Both MP and SM	Asynch	MB
2013	Mizan [118]	V	D	BSP	MP	Synch	DB
2013	Trinity [202]	V	D	TSL	SM	Asynch	MB
2013	Grace [236]	V	S	Three phases	MP	Asynch	DB
2013	GPS [195]	V	D	BSP	MP	Synch	DB
2013	Giraph++ [223]	C	D	BSP	Both MP and SM	Both	DB
2013	Naiad [160]	V	D	Timely Dataflow	SM	Both	MB
2013	PAGE [201]	V	D	Partition-aware	MP	Synch	DB
2013	Stratospher [4]	V	D	Push/Pull	Dataflow	Synch	DB
2013	TurboGraph [91]	V	S	Pin-and-slide	SM	Asynch	DB
2013	xDGP [232]	V	D	BSP	MP	Synch	DB
2013	X-Stream [192]	E	S	Scatter-gather	MP	Synch	DB
2013	GiraphX [220]	V	D	BSP	SM	Asynch	DB
2013	GraphX [81]	E	D	GAS	Dataflow	Synch	MB
2013	Galois [165]	V	S	ADP	SM	Asynch	DB
2013	GRE [251]	V	D	Scatter-Combine	MP	Synch	DB
2013	Ligra [204]	C	S	Push-pull	SM	Asynch	MB
2013	LFGraph [98]	V	D	N/A	SM	Synch	MB
2013	PowerSwitch [240]	V	D	Hybrid	SM	Both	DB
2013	Presto [233]	V	D	N/A	Dataflow	Synch	DB
2013	Medusa [263]	V	H	EMV	MP	Synch	MB
2014	RASP [252]	V	S	Scatter-gather	SM	Asynch	DB
2014	GoFFish [208]	C	D	Iterative BSP	Both MP and SM	Synch	MB
2014	GasCL [41]	V	H	GAS	MP	Synch	MB
2014	CuSHa [119]	V	H	GAS	SM	Asynch	MB
2014	BPP [162]	V	S	BSP	SM	Asynch	DB
2014	Imitator	V	D	BSP	MP	Synch	DB
2014	GraphHP [42]	V	D	BSP	MP	Synch	DB
2014	PathGraph [253]	P	S	Scatter-gather	SM	Asynch	DB
2014	Seraph [245]	V	D	GES	MP	Synch	DB
2014	GraphGen [169]	V	H	N/A	SM	Synch	MB
2014	Blogel [248]	B	D	N/A	MP	Synch	MB
2015	Pregelix [32]	V	D	Join-operator based	MP	Synch	DB
2015	FlashGraph [262]	V	S	BSP	Both MP and SM	Asynch	DB
2015	GraSP [23]	V	D	N/A	MP	Synch	MB
2015	Chaos [191]	E	D	GAS	MP	Synch	DB
2015	GraphMap [132]	V	D	BSP	MP	Synch	DB
2015	GridGraph [294]	E	S	Streaming-Apply	SM	Asynch	DB
2015	GraphQ [237]	V	S	Check/Refine	SM	Asynch	DB
2016	Gunrock [235]	Da	H	BSP	SM	Synch	MB
2016	GraphIn [200]	V	D	I-GAS	MP	Synch	MB
2016	DUALSIM [121]	V	S	N/A	SM	Asynch	DB
2016	iGiraph	V	D	BSP	MP	Synch	DB
2017	GraphMP [217]	V	S	VSW	SM	Asynch	DB
2017	GraphGen [243]	V	S	N/A	SM	Asynch	MB
2017	Mosaic [145]	V/E	S	PRA	MP	Synch	DB

Although, many frameworks have been proposed for processing large-scale graphs, there are still several gaps that need to be addressed, as highlighted by this table. Among these observations are: 1) Many graph processing systems have been developed based on vertex-centric programming model because it is the simplest way of partitioning and processing large-scale graphs. Although edge-centric and component-centric systems are more difficult to implement, it has been empirically shown that frameworks such as PowerGraph and GoFFish can scale more efficiently than vertex-centric ones. So, those types of systems need to be investigated more. 2) Disk-based approach is the dominant mechanism that is used by most frameworks. It also includes the frameworks that support out-of-core computation. Disks are cheap but much slower than memory. On the other hand, memory is faster than disk but it is more expensive and memory management makes it more complicated to develop a system based on this approach. 3) Synchronous programming is popular on distributed systems as they avoid race conditions, but often require message passing and have longer runtimes due to the coordination. While asynchronous methods work well on single machine and heterogeneous based frameworks, its effect on distributed frameworks is less studied.

2.7 Different Viewpoints On Categorization Of Graph Processing Systems

Graph processing systems can be categorized based on different intuitions. In this chapter, we have categorized various features as depicted in Figure 2.15. We consider both graph programming models and runtime aspects as two broad aspects of graph processing systems while each can contain multiple sets of features to simplify the understanding of graph processing mechanisms. As part of graph programming models, we explained various system architectures (Section 2.3.1), current frameworks and how they look at the processing paradigm (Section2.3.2), possible distributed coordination that conveys timing (Section 2.3.3) and computational models (Section 2.3.4). On the other side, runtime aspects discuss partitioning as the heart of the system (Section 2.4.1), different communication models (Section 4.2), storage views (Section

2.4.3), fault tolerance (Section 2.4.4) and scheduling (Section 2.4.5). This allows readers to obtain a clear insight about each part of the system, the relationship among various components and possible improvements.

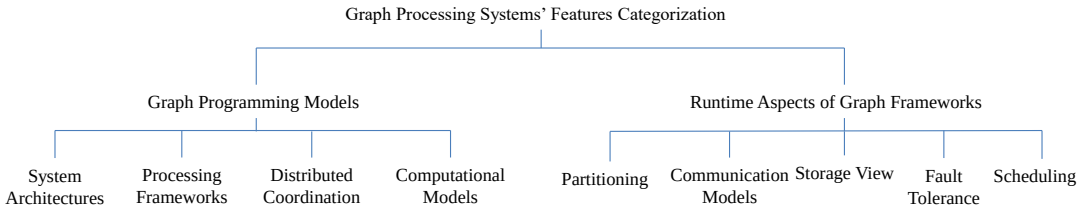


Figure 2-15 Proposed graph processing features' categorization in this chapter

However, there can be different viewpoint on this. One might separate the features according to application-related and computing-related aspects of the features (Fig. 2.16). In this viewpoint, application characteristics refer to the features that are designed based on the graph system itself so they might have different implementation accordingly. For example, programming models, partitioning, computational models and communication models are specific characteristics of the system. Computing platform aspects refer to more general features that can vary in different systems but are not specific characteristic of the system. This view is illustrated in Figure 2.16.

Another viewpoint [153] has classified frameworks based on four major characteristics and called it "four pillars of think like a vertex frameworks": 1) timing, 2) communication, 3) execution model, and 4) partitioning. Overall, regardless of various viewpoints about categorizing graph processing features, there are certain characteristics in every system that is usually considered to be improved in research works.

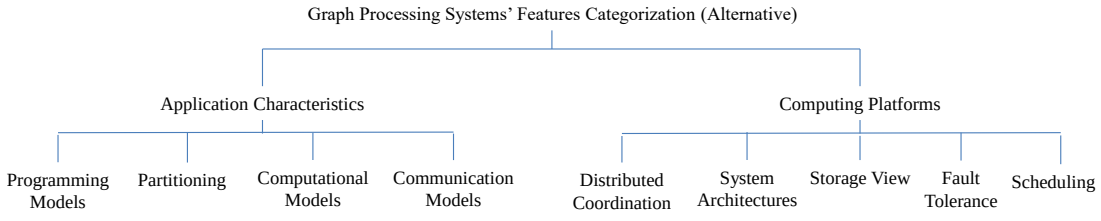


Figure 2-16 Graph processing features' categorization according to application characteristics and computing platforms

2.8 Summary

Huge quantities of data are being created, analyzed and used every day in the contemporary world of Internet communications and connected devices. “Big Data” is the term used to signify the challenges posed by this massive data influx. A growing majority of big data is in the form of “Graphs” which is one of the major computational methods of huge data analysis. Social network applications and web searches, Internet of Things, knowledge graphs and deep learning, financial transactions, and neuroscience are some examples of large-scale graphs that need to be analyzed for various domains. Several works have investigated the creation of effective systems for processing large-scale graphs in recent years.

In this chapter, we have investigated and categorized existing graph processing frameworks and systems from different perspectives. First, we explained how different parts of a graph processing system including read and write from/to disk or memory, pre-processing, partitioning, communication, computation and error handling work together to process large-scale graphs. Second, we presented a taxonomy of different abstractions and approaches that are used in existing graph processing systems within each of these phases. In addition, we described notable frameworks that have used these techniques, and analyzed their advantages and disadvantages to support our discussions. We further summarized the features of graph processing frameworks developed since 2009 in Table 2.3. It gives a comprehensive overview of current systems and enables making comparison between them. Finally, future research directions are discussed which shows that scalable graph processing is still at a nascent stage and there are many issues that remain unsolved

Chapter 3
iGiraph: A Cost-efficient Graph Processing Framework

Large-scale graph analytics has gained attention during the past few years. As the world is going to be more connected by appearance of new technologies and applications such as social networks, Web portals, mobile devices, Internet of things, etc, a huge amount of data are created and stored every day in the form of graphs consisting of billions of vertices and edges. Many graph processing frameworks have been developed to process these large graphs since Google introduced its graph processing framework called Pregel in 2010. On the other hand, cloud computing which is a new paradigm of computing that overcomes restrictions of traditional problems in computing by enabling some novel technological and economical solutions such as distributed computing, elasticity and pay-as-you-go models has improved service delivery features. In this chapter, we present iGiraph, a cost-efficient Pregel-like graph processing framework for processing large-scale graphs on public clouds. iGiraph uses a new dynamic re-partitioning approach based on messaging pattern to minimize the cost of resource utilization on public clouds. We also present the experimental results on the performance and cost effects of our method and compare them with basic Giraph framework. Our results validate that iGiraph remarkably decreases the cost and improves the performance by scaling the number of workers dynamically.

This chapter is partially derived from:

- **Safiollah Heidari,** Rodrigo N. Calheiros and Rajkumar Buyya, "iGiraph: A Cost-efficient Framework for Processing Large-scale Graphs on Public Clouds", in Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016), Cartagena, Colombia, Pages 301-310, 2016

3.1 Introduction

AS Internet continues to grow, the world is becoming a more connected environment and the number of data resources is increasing beyond what had been predicted before [210]. Amongst various data modeling approaches to store huge data, graphs are widely adopted to model complex relationships among objects. A graph consists of sets of vertices and edges which demonstrate the pairwise relationship between different objects. Many applications and technologies such as social networks, search engines, banking applications, smart phones and mobile devices, computer networks, the semantic web, etc, are modeling and using data in the form of graphs [199]. These applications generate massive amounts of data which are represented by graphs. Facebook [184], for example, has more than one billion users that are considered as the vertices of a huge graph where the relationships between them are considered as the edges of the graph. To gain an insight and discover knowledge from these applications, the graph that represents them should be processed. However, the scale of these graphs poses challenges to their efficient processing [177].

In order to process large graph problems, every solution confronts with some challenges due to the intrinsic properties of graphs. These properties include data-driven computation, unstructured problems, poor locality and high data access to computation ratio [144]. Therefore, graph problems are not well matched with existing processing approaches and usually prevent efficient parallelism. MapReduce [56], for example, which addresses many shortcomings in previous parallel and distributed computing approaches, is not an appropriate solution for large graph processing. This is because first, MapReduce uses a two phased computational model (map and reduce) which is not well suited for iterative characteristic of graph algorithms. Second, its tuple-based approach is poorly suited for most of graph applications [2].

Cloud computing is a new paradigm of computing that has changed software, hardware and datacenters design and implementation. It overcomes restrictions of traditional problems in computing by enabling some novel technological and economical solutions like using distributed computing, elasticity and pay-as-you-go

models which make service providers free from previous challenges to deliver services to their customers [35]. Cloud computing presents computing as a utility that users access various services based on their requirements without paying attention to how the services are delivered or where they are hosted. Public cloud computing services, for instance, offer Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) for large distributed processing, are becoming more popular among companies who want to focus on their business instead of being concerned about technical issues. Rapid on-demand compute resource provisioning brings cost scalability based on utilization. As an example, Amazon EC2 provides three cost models for its customers based on their requirements –spot, on-demand and reserved provisioning. Using these commercial services, the customer may choose to pay more to achieve better performance or reliability. So, making a proper decision between using the number of resources the user wants to use and the money that the user wants to pay for the service is an issue while using public clouds.

Some graph processing frameworks such as Surfer [46] and Pregel.Net [187] were developed to support processing large graphs on public clouds, but they have considered some specific issues on these frameworks and do not address the impact of their solutions on the monetary cost of the system. For example, Surfer has proposed a graph partitioning method based on network latency and Pregel.Net, which is the .Net-based implementation of Pregel, has analyzed the impact of BSP graph processing models on public clouds using Microsoft Azure. On the other hand, there are some services such as Amazon relational database service (RDS) which is designed for traditional relational databases[14]. It is aimed at facilitating the set-up, operation and scaling a relational database and comes with two reserved and on-demand instance packages. According to a recent report from DB-Engine[15], graph databases are getting more and more attentions every day and many companies are going to use this kind of

[14] Amazon Relational Database Service (RDS) : https://aws.amazon.com/rds/
[15] DB-Engines Ranking Per Database Model Category. (2015, August 5). Retrieved August 15, 2015, from DB-Engines: http://db-engines.com/en/ranking_categories

database for their businesses. So, in the near future, public cloud providers will introduce new graph processing services on their infrastructures.

However, current frameworks for graph processing have limitations that hinder their adoption in cloud platforms. First, the majority of them have been designed and tested on cluster environments and not clouds, hence they have not considered monetary optimization, which is a very important factor for service selection on clouds. Second, many graph processing frameworks focus on reducing the operation's execution time, reducing memory utilization, considering task priorities and so on to reduce the cost of processing, but considering a static pool of resources with known size. On the other hand, cloud computing provides high scalability on-demand resources that can help users to perform their tasks using various services. Therefore, a graph processing approach with performance guarantees and optimal cost is a must in a cloud setting.

In this chapter, we propose a graph processing framework called iGiraph. It uses a cost-efficient dynamic re-partitioning approach that utilizes network traffic message pattern to reduce the number of virtual machines (workers) during the processing by migrating partitions and vertices to minimize the cost. The new repartitioning method also mitigates network traffic results in faster execution. Our work, iGiraph makes the following **key contributions**:

- iGiraph repartitions the graph dynamically across workers considering network traffic pattern to reduce the communication between compute nodes.
- iGiraph uses high degree vertices concept in partition level, with the convergent level of the algorithms that are running on the system. iGiraph manages the number of compute nodes using a proper combination of these methods.
- While cost is a very critical factor in service selection procedure for any user on a public cloud, iGiraph significantly reduces the cost of processing large-scale graphs with reasonably close runtimes to Giraph by its new approach.

The rest of the chapter is organized as follow: section 3.2, explains the basic Apache Giraph framework and its features following by the vertex and algorithm

categorization is used for our work. Section 3.3 gives details about iGiraph solutions. Section 3.4 shows iGiraph's implementation. Performance evaluation of the system is discussed in section 3.5 and finally, related works and conclusions and future works are explained in sections 3.6 and 3.7, respectively.

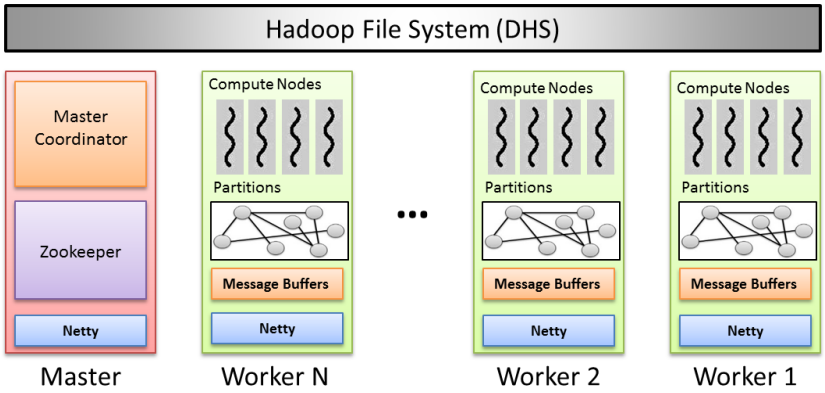


Figure 3-1 Giraph's Architecture

3.2 Background

In this section, we first introduce Apache Giraph[16] which is the fundamental framework for our system. Then, we explain Bulk Synchronous Parallel (BSP) [231] model following by describing a vertex categorization that is effectively used for our re-partitioning model. Finally, we explain the graph algorithm classification we used in this chapter which has a great impact on choosing the right strategy to reduce the cost of the whole system.

3.2.1 Giraph

Apache Giraph is an open-source implementation of proprietary Pregel. It is a distributed graph processing framework that uses a set of machines (workers) to process large graph datasets. One of the machines plays the role of master to

[16] Apache Giraph: http://giraph.apache.org/

coordinate with other slave workers. The master is also responsible for global synchronization, error handling, assigning partitions to workers and aggregating aggregator values. Giraph is a Hadoop-based framework that runs workers as map-only jobs and uses Hadoop data file system (HDFS) for data I/O. It also employs Apache ZooKeeper[17] for checkpointing, coordination and failure recovery scheme. Giraph added many features beyond the basic Pregel including sharded aggregators, out-of-core computation, master computation, edge-oriented input and more. Finally, having a growing community of users and developers worldwide, Giraph has become a popular graph processing framework that even big companies such as Facebook are using it to process their huge datasets [48].

Giraph utilizes vertex-centric programming model like Pregel in which each vertex of the graph is identified by a unique ID. Each vertex also has other information such as a vertex value, a set of edges with an edge value for each edge, and a set of messages sent to it. To process a large graph in vertex-centric model, it should be partitioned into smaller parts by a partitioner where each partition is connected to other partitions by cross-edges between them. The partitioner also distributes partitions to a set of worker machines. In Giraph, a partitioner determines which partition a vertex belongs to based on its ID. Giraph uses a default hash function on the vertex ID to partition a graph while other customized partitioners also can be used. To improve the load balancing, the number of partitions is often greater than the number of workers.

Using simple static partitioning methods makes Giraph to run and process various graph algorithms slower and with more costs than other Pregel-like frameworks such as GPS [195] or Giraphx [220]. These systems have shown that using more complicated static partitioning algorithms such as METIS [116], rather than using a simple hash partitioning method, can remarkably improve the performance. GPS for example, uses a combination of a static partitioning algorithm to partition the graph and a dynamic re-partitioning algorithm during the computation to distribute the remaining non-processed vertices to idle workers to reduce the execution time within a superstep. In

[17] Apache Zookeeper: https://zookeeper.apache.org/

this chapter, we choose the hash partitioning algorithm to start partitioning the graph with, but during the computation we replace that with a dynamic traffic-aware re-partitioning algorithm to reduce the cost of the whole processing operation and improve the performance of the system.

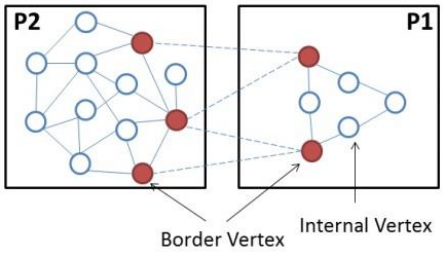


Figure 3-2 Internal vertices and border vertices

3.2.2 Bulk Synchronous Parallel Model

Bulk Synchronous Parallel (BSP) is a vertex-centric computational model in which every single vertex of the graph can carry two states of *active* or *inactive*. All vertices are *active* when the computation starts. The processing consists of a series of iterations, called *supersteps*, followed by global synchronization barriers between them. In each iteration, every vertex that is involved in computation, 1) receives its neighbors updated values from previous iteration, 2) the vertex then will be updated by received values, 3) and finally, the vertex sends its updated value to its adjacent vertices that will be available to them in the next superstep and changes its state to *inactive*.

The advantage of using BSP model in Giraph is that all the aforementioned operation is executed by a user-defined *Compute()* function of the *Vertex* class. After all the vertices completed executing *Compute()* function in a superstep, data will be aggregated during the synchronization phase and the messages generated by each vertex will be available to their destinations at the beginning of next superstep. If a vertex does not receive any messages during a superstep, it can deactivate itself by calling *voteToHalt()* function. However, a deactivated vertex can be activated by

receiving messages from its neighbors. If there is not any active vertex, the computation is finished.

3.2.3 Internal Vertices and Border Vertices

A graph G=(V,E) consists of a set of vertices V={v_1, v_2, …, v_n} and a set of edges E={e_1,e_2, …, e_m} where $E \subset V \times V$. In the vertex-centric graph processing approach, the graph is divided into smaller partitions based on vertex divisions so that $P_1 \cup P_2 \cup \ldots \cup P_k = V$ are *k* partitions of V where $P_i \cap P_j = \varnothing$, $\forall i \neq j$. Therefore, each vertex basically belongs to only one particular partition [223].

An internal vertex is a vertex that all its adjacent vertices are inside the same partition as this particular vertex is. So, the messages coming out from an internal vertex only flow within the partition. On the other side, a border vertex is a vertex that at least one of its neighbors is placed in another partition. Hence, a border vertex's outgoing messages need to be sent to at least one different partition than the partition this particular vertex belongs to. Passing messages between partitions leads to increasing network traffic which results in longer execution time, inefficient resource utilization and higher costs in turn. Internal vertices and border vertices are shown in Figure 3.2. One of the approaches for avoiding message passing side effects is to partition the graph in a way that reduces the number of border vertices and cross-edges between partitions so that the number of messages passing between partitions will be reduced.

3.2.4 Graph Algorithms

Different research use different classification of algorithms. For example, one may classify graph algorithms into traversal algorithms, graph aggregation algorithms, random walk algorithms and so on, while another one classifies them as global queries and targeted queries [112]. In this chapter we use our own classification which categorizes graph algorithms based on their behavior in network traffic making and generating messages during the processing. We classify algorithms into two groups as follow:

- Non-Convergent Algorithms: Non-convergent algorithms are the algorithms that generate almost the same number of messages during processing. They complete the processing by passing the same number of messages in the last superstep as the number of messages they passed during first supersteps. So, the number of messages are generated using these applications never tends to become zero. PageRank [173], for instance, is a non-convergent algorithm.
- Convergent Algorithms: In contrast to non-convergent algorithms, the number of messages are generated using convergent algorithms tend to fall down to zero by the end of processing operations. Computing shortest paths [190] and connected components [196] algorithms are among convergent algorithms.

Here, we give a brief explanation of the algorithms we use from each category.

1) PageRank

PageRank is an algorithm which is used to measure the significance of website pages. PageRank works by measuring the number of links (hyperlinks) to a page to specify an importance estimation of a website. The more important the page is, the more links it receives from other pages. PageRank does not rank a website as a whole, but is assessed by each page exclusively. The PageRank of page P_i does not impress the PageRank of a typical page P uniformly because of different weights that each page has. The summation of weighted PageRanks of all pages P_i then is multiplied by an alleviation factor 'd' that usually is set between 0 and 1. PageRank is also a non-convergent algorithm according to above classification because it produces the same number of messages in each superstep during a processing operation.

2) Connected Components

A connected component algorithm finds different sub-graphs of a particular graph in which there is a path between any two vertices and that is not connected to any further vertices in the super-graph. We use HCC that starts with having all vertices in an initial active state. Each vertex starts computing by considering its ID as its component ID and update this component ID when it receives a smaller component ID. The vertex then propagates the updated value to its adjacent vertices. Connected component is a

convergent algorithm because the number of passing messages between vertices tends to fall down to zero as the states of vertices change to inactive until the end of computation.

3) Single Source Shortest Paths

The shortest path in graph theory is the problem of discovering a path between two nodes such that the summation of the weights of its edge components is minimized. This is a well-known problem in graph theory and there are different approaches and applications applying various solutions to various problems in this field.

Single source shortest path (SSSP) problem is one derivation of the main shortest path problem. This problem needs to find a shortest path between a single source node and all other vertices in the graph. In this algorithm, each vertex initializes its value (distance) to INF (∞), while the source node put 0 as its distance. INF is larger than any possible path from the source node in the graph. In the first superstep, only the source node updates its neighbors; in the next superstep, the updated neighbors will send messages to their own neighbors and so on. The algorithm completes when there is no more updates happening and the states of vertices also changed to inactive. So, SSSP is a convergent algorithm according to the aforementioned definition.

3.2.5 Graph Processing Challenges on Clouds

A large-scale graph processing operation that includes a series of iterations to process a graph usually causes considerable overheads due to its large memory consumption, CPU utilization, error handling, etc. Accordingly, various frameworks are proposed to optimize and improve the performance of graph processing operations. Although many of these frameworks offer specified scalability improvements on high performance clusters with fast interconnections, their performance on cloud environments in which some critical factors such as service cost is determinative, is less studied. So, there are not many works that considered monetary optimizations. Besides, many existing frameworks consider memory utilization, runtime reduction, tasks prioritization and so on by using constant number of resources. So, they are not utilizing clouds elasticity and scalability that are important characteristics of cloud

environments and can have significant impact on monetary costs. Our work is scoped to reduce the monetary cost of processing large-scale graphs on public clouds by proposing a Pregel-like framework.

3.3 iGiraph

3.3.1 Motivation

iGiraph utilizes a distributed architecture on top of Hadoop and uses its distributed file system for data I/O. It is a Pregel-like graph processing system which means it employs vertex-centric processing solutions to process a graph and follows Pregel-like systems' behaviors. The problem with many of existing graph processing systems, particularly Pregel-like frameworks, is that although they propose methods to run the processing faster and improve the performance of the system, resource utilization and monetary cost factors are less studied. Nonetheless, cost is a crucial factor for every business that wants to use public cloud infrastructure. As cloud providers are using pay-as-you-go models for the services they are providing, considering the factors that have impacts on the cost of the services is very important for customers to choose the right services. There are many factors that influence the whole processing costs in a cloud environment including:

- Execution time: The longer the operation takes, the more user has to pay.
- Resource costs: Every resource has its own price. So, choosing the right number of machines with the right size can make huge differences.
- Communication: Sending and receiving data in a cloud environment is not free hence reducing the cost of communication for each operation is vital.
- Storage: Storing data could also become costly specially, for big data related services.

One of the most important parts of a graph processing system is the partitioning method that is used to partition and distribute data across the workers. Choosing between various static partitioning methods or between static and dynamic

partitioning approaches can affect the system performance and cost. iGiraph uses a dynamic graph re-partitioning method which considers the main cost factors and improves the processing performance.

3.3.2 iGiraph's Dynamic Re-partitioning Approach

iGiraph's repartitioning algorithm uses the concept of high degree vertices in partition level and merges the partitions to reduce the number of cross-edges between them by migrating partitions from one worker to another. During this process, some workers (resources) gradually become empty and can be released to decrease the cost of resource utilization.

In many real world graphs only a few number of nodes contains a large fraction of all the edges in the graph [195]. These vertices are known as *high degree* vertices. While the number of edges connecting to a vertex states the degree of that vertex, a high degree vertex has much more connected edges compared to majority of the vertices in a graph. For example, in a social network, a singer, an actor or celebrities can have millions of followers in comparison with the average of tens or hundreds of friends and followers for an ordinary user.

High degree vertices can play an important role in causing network traffic and delaying the execution time specially when they are placed as border vertices in partitions or close to border vertices. That is, putting high degree vertices as close as possible to their neighbors can significantly improve the network and system performance. Figure 3.3 shows the importance of this issue. In Figure 3.3.a vertex v from partition P1 is connected to many vertices in P2 results in huge network traffic while passing messages between two partitions and therefore delays the run-time and increases the cost. But as is shown in Figure 3.3.b, moving v to P2 can remarkably reduce the cross-edges between two partitions.

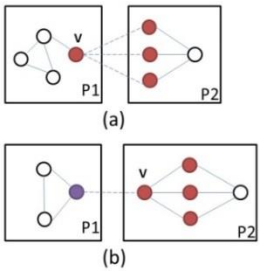


Figure 3-3The role of high degree border vertices in reducing network traffic

iGiraph uses high degree vertex concept in partition level not vertex level. It means that as there are vertices with higher degree than other vertices in the graph, there are also partitions that send or receive more messages than other partitions in the graph of system workers. In order to store the information about which partition has sent or received more messages, iGiraph uses two separate lists. One stores the number of outgoing messages from each partition and the other, stores the number of incoming messages to each partition. We also define α, which is a threshold that is an average value for the number of messages that are transferring between each pairs of partitions. α is defined as follows:

$$\alpha = \frac{\sum_{i=1}^{N_P} \sum_{j=1}^{n} N_m(P_j, P_{j+1})}{N_P} \quad , \qquad (3.1)$$

In the above formula, $N_m(P_j,P_{j+1})$ shows the number of messages between partition *j* and partition *j+1*, N_P shows the number of partitions that are involved in each superstep and *n* is calculated based on the number of partitions to show the number of pairs in each superstep. This formulation is calculated between each supersteps in iGiraph. According to this:

$$n = \begin{cases} N_P \times \left\lceil \frac{N_P}{2} \right\rceil & \text{If } N_p \text{ is odd} \\ (N_P - 1) \times \left\lceil \frac{N_P}{2} \right\rceil & \text{If } N_p \text{ is even} \end{cases} \qquad (3.2)$$

If the number of messages received by a partition is equal or greater than α, then that partition is a potential candidate for migration, otherwise the program looks at the number of outgoing messages at that partition to see if it can host vertices from other

partitions or merge with them. Using factor α alone, border vertices can migrate between partitions.

Although α is a determinative factor to specify which partitions are suitable for migration and merging, there are other important factors that can influence the final decision as well. One factor is the number of total messages transferred between all partitions in a particular superstep compared to the number of total messages transferred between all partitions in previous superstep. Merging (not migration) only can occur if this proportion is decreasing. As long as the number of messages is growing during the processing, no merging will happen.

Another factor that determines whether partitions can merge is the size of partitions and workers' capacities. As the processing continues, for convergent algorithm such as connected components and shortest path, the vertices that complete the computation change their states to *inactive*. So, instead of keeping these vertices in the memory until the end of processing operation, iGiraph deletes them temporarily from memory to provide room for partition merging. On the other side, if a removed vertex is invoked during the computation, iGiraph can bring it back to the memory. So, before merging two partitions, the system checks if the destination worker has enough space or not.

When all above conditions are true, then migrating a partition from one worker to another worker to merge it with the other partition is possible. This has influences on the total cost of the service. For example, according to Figure 3.4, partition P1 is a high degree partition, which means it has the greatest number of incoming messages among other partitions, and is placed on worker W1. Partition P2 which is placed on worker W2 has sent the greatest number of messages to P1, P3 is in the second place after P2, P4 is next and so on. In addition, total number of transferred messages in current superstep (i+1) is less than transferred messages in previous superstep (i) and the workers have sufficient memory after removing inactive vertices. At this time, P1 will merge with P2 until there is free space on W2. Additional vertices will be migrated to W3 and so on. A load balancer balances the number of vertices in each partition on remaining workers.

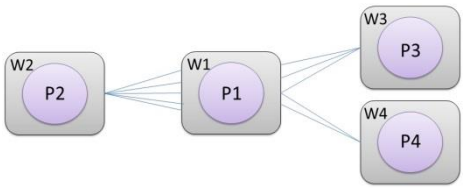


Figure 3-4 Worker W2 has sent more messages to W1 than other workers

According to our experiment results, using the proposed re-partitioning algorithm for convergent applications can reduce the cost of resource utilization while the execution time is close to Giraph's experiment results or with only a bit of increasing in some cases, but still do not affect the whole results.

For non-convergent applications, iGiraph does not merge the partitions. So, the number of workers will remain the same from beginning of the processing to the end. Instead, only border vertices from high degree partitions will be migrated to reduce the cross-edges between partitions. In this case, the total average number of transferred messages is mitigated which leads to faster execution compared to Giraph.

3.4 iGiraph Implementation

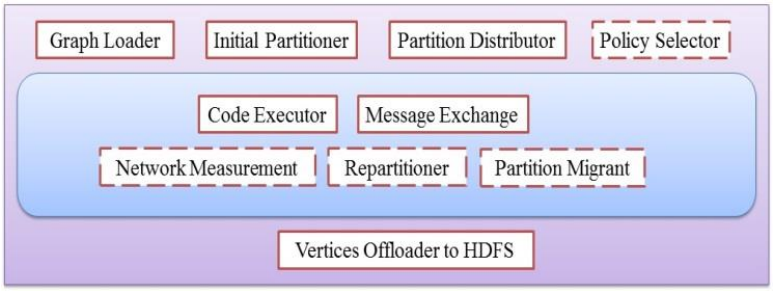


Figure 3-5 System architecture and components

Figure 3.5 shows the iGiraph's system architecture and components added to basic Giraph. The components that are surrounded by simple lines are basic Giraph's that are used in iGiraph too. The components that are surrounded by dashed lines are the components which are added to the basic framework. Like Giraph, data is loaded and stored on HDFS. Then, an initial partitioner function will partition the graph and

prepare the partitions for being distributed across workers. In this chapter we use only a simple hash function as initial partitioner. The hash function partitioning method is proved that results in worst performance compared to other complicated initial partitioning methods. Hence, we want to reach a better performance using this approach to show that our method can work very well even in this case. In the next step, partitions will be distributed across workers. The policy selector selects the appropriate computation method based on the type of application. For example, if the algorithm is convergent it enables partition migration. Code executer is the main *Compute()* function that executes the algorithm on each active vertex. After that, according to the number of messages transferred between partitions during the superstep, a network measurement component will determine which partitions have sent or received messages in a descent order. Then the repartitioner chooses vertices or partitions to migrate or merge according to the policy is selected. This will be done by the partitions migrant. This process will continue until all the vertices in the graph change their states to inactive and there is no more vertices to be computed. Finally, the results will be written back to HDFS.

3.5 Performance Evaluation

3.5.1 Experimental Setup

We chose shortest path and connected components algorithms among convergent applications and PageRank among non-convergent applications for our experiment. We also use three real datasets [125] of varying sizes: Amazon, YouTube and Pokec which is a Slovak social network.

Table 3-1 Evaluation datasets and their priorities [125]

Graph	Vertices	Edges
Amazon (TWEB)	403,394	3,387,388
YouTube Links	1,138,499	4,942,297
Pokec	1,632,803	30,622,564
Twitter	41,652,230	1,468,365,182

We use *m1.medium* NECTAR VM instances for all partition worker roles. NECTAR is Australian national cloud infrastructure facilities [163]. Medium instances have 2-cores with 8GB RAM and 70GB disk including 10GB root disk and 60GB ephemeral disk. All the instances are in the same zone and use the same security policies. We also installed NECTAR Ubuntu 14.04 (Trusty) amd64 on each instance. We use Apache Hadoop version 0.20.203.0 and Apache Giraph version 1.1.0 with its checkpointing characteristic turned off. All experiments run using 16 instances where one takes the master role and others are set up as workers.

3.5.2 Evaluation and Results

First, we investigate the impact of our proposed approach on convergent algorithms and compare the results with basic Giraph. Then, we investigate non-convergent PageRank algorithm on both frameworks.

1) Evaluation of Convergent Algorithms

Figure 3.6 and 3.7 show the results of comparison experiments between Giraph and iGiraph on Amazon and Pokec datasets respectively. Considering that the size of every network message is the same in all experiments, here the computation can converge faster using iGiraph while the number of messages passing through network is reduced significantly. In Figure 3.7, after using factor α the number of messages increased a bit at first superstep, but noticeably decreased after that and still shows significant network message reduction compared to Giraph.

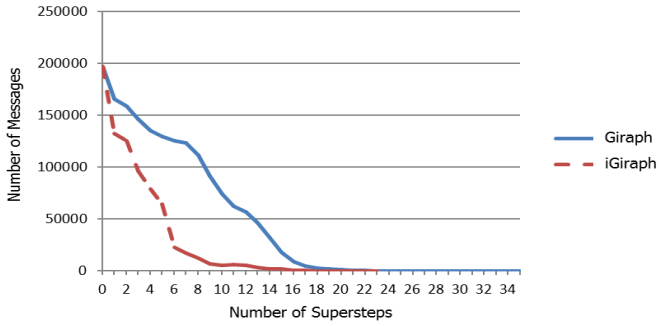


Figure 3-6 Number of network messages transferred between partitions across supersteps for the Amazon graph using connected components algorithm

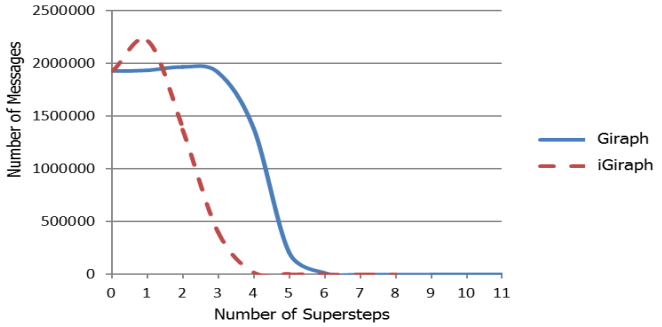


Figure 3-7 Number of network messages transferred between partitions across supersteps for the Pokec graph using connected components algorithm

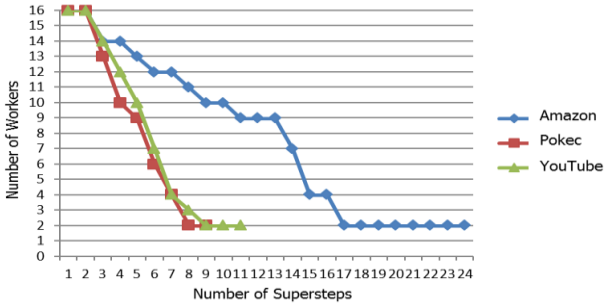


Figure 3-8 Number of machines varying during supesteps while running connected component algorithms on different datasets on iGiraph

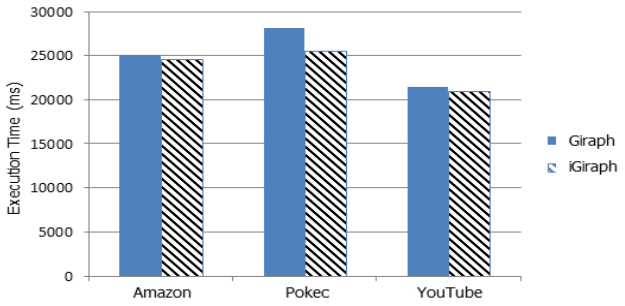


Figure 3-9 Total time taken to perform connected components algorithm

In contrast to Giraph in which the number of workers is kept intact during the whole operation, iGiraph releases compute nodes as the graph get converged. That is because by keeping only active vertices for the operation and doing repartitioning between each supersteps, less computation resources are required to continue the processing. We observed that by removing inactive vertices after each superstep, we could merge more partitions to use the capacities of each worker's memory efficiently. So, the more partition merge, the more resources can be freed which results in more money saving. But this claim only can be true when we consider both resource reduction and execution time together.

$$Cost_{final} = \sum_{i=1}^{n} \left(P(VM_i) \times T_{total}(VM_i) \right) \qquad (3.3)$$

According to the above formulation, total cost of using resources on a cloud environment is equal to the summation of the price of each resource $P(VM_i)$ multiplied by total time of using that resource $T_{total}(VM_i)$. To calculate the final cost for the whole processing operation beside reducing the number of resources, we need to measure the system run-time too. Note that although data transfer also has impact on the final cost calculation, we have not considered that here, but we will take it into consideration for our future works. Figure 3.9 shows the execution time for processing aforementioned datasets using connected components algorithm. It shows that in addition to decreasing the cost of resource utilization, the run time for the operation is also reduced. Therefore, according to formula 3.3, the total cost of the operation falls down too.

Similar to previous evaluations for connected component algorithm, we repeated experiments using shortest path algorithm for both Giraph and iGiraph. From the network traffic point of view, the difference between shortest path and connected component is that the former starts with passing a few number of messages at the beginning of computation and gradually increases until reach a maximum and then starts converging, but connected component starts with passing great number of messages hence it immediately starts the convergence process. Figure 3.10 shows the results of a comparison experiment between Giraph and iGiraph on Amazon dataset using connected components algorithm. It takes 37 supersteps for this process to be completed on Giraph while it converges around superstep 23 using iGiraph. This is because in contrast to Giraph in which the number of messages starts falling down from superstep 14, using factor α, this happens to iGiraph after superstep 8. From this point onwards in iGiraph, three conditions for partition merging are provided and according to Figure 3.13 it can be seen that the number of active workers are decreasing. The results for Pokec and YouTube are shown in Figure 3.11 and 3.12, respectively.

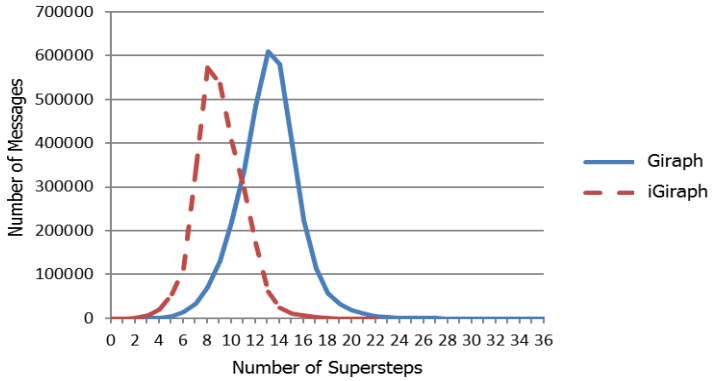


Figure 3-10 Number of network messages transferred between partitions across supersteps for the Amazon graph using shortest path algorithm

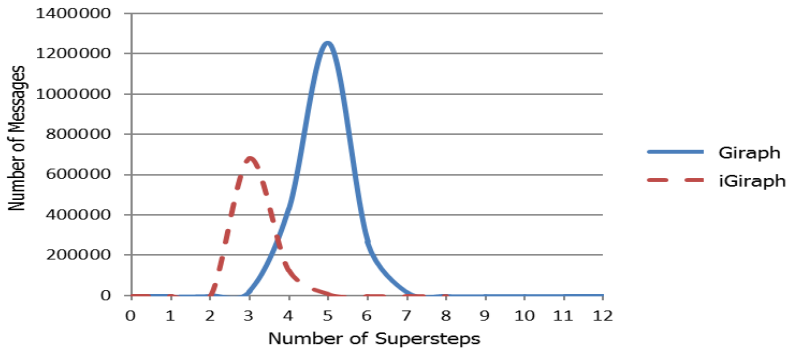


Figure 3-11 Number of network messages transferred between partitions across supersteps for the Pokec graph using shortest path algorithm

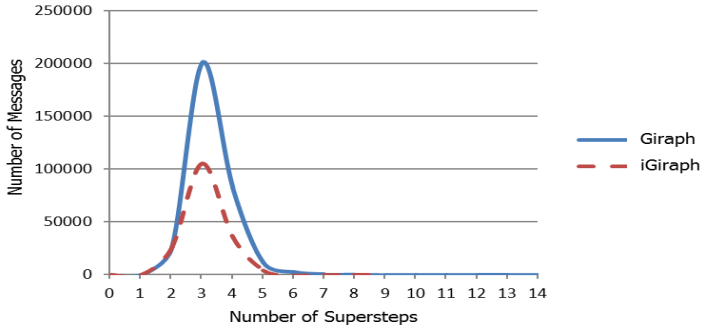


Figure 3-12 Number of network messages transferred between partitions across supersteps for the YouTube graph using shortest path algorithm

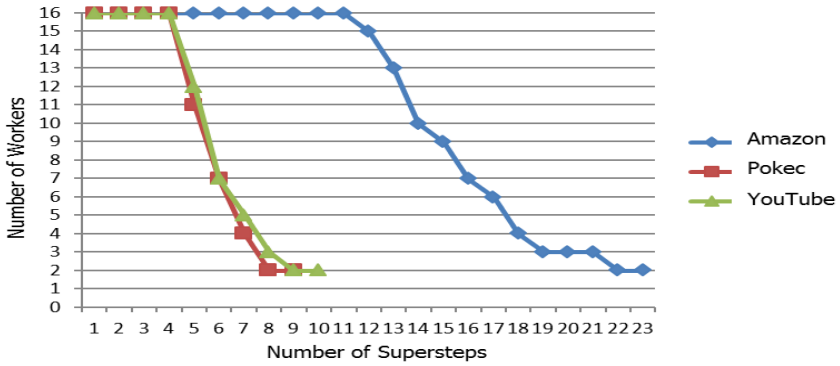


Figure 3-13 Number of machines varying during supesteps while running connected component algorithms on different datasets on iGiraph

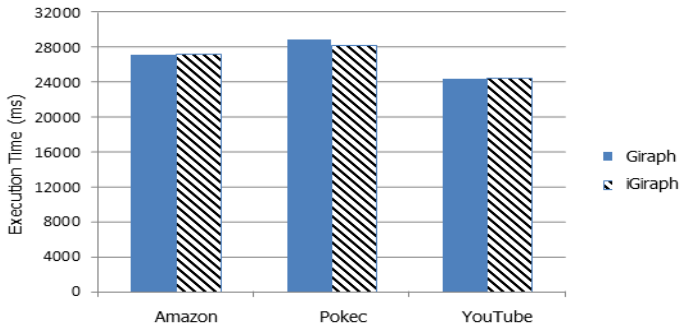


Figure 3-14 Total time taken to perform shortest path algorithm

The above figure shows that the time taken to complete shortest path algorithm on 16 machines using iGiraph is not significantly different than Giraph. As a result, considering total execution time and decreasing number of active workers in each experiment, iGiraph is more cost-effective than Giraph for convergent algorithms on public clouds.

2) Evaluation of Non-Convergent Algorithms

Processing non-convergent algorithms such as PageRank shows that the number of messages generated in each superstep is almost the same as other supersteps during the whole processing. In PageRank for example, vertices always update their neighbors during the computation hence as long as the number of vertices is the same, the number of messages is also the same. But it is still possible to reduce the network messages by using α factor. α determines the partitions that receive more messages through network than the other partitions (high degree partitions). Then, to balance the

messaging pattern, iGiraph selects a number of border vertices from high degree partitions to relocate based on the aforementioned algorithm in section 3.4. After relocating the vertices, a load balancer method will balance the number of vertices in each partition. It can be seen that the average number of network messages falls down a bit in iGiraph results in faster computation. Figure 3.15 shows the average number of network messages in both Giraph and iGiraph. The total execution time for each experiment also can be seen in figure 3.16. According to these figures, although we did not decrease the number of workers like what was done for convergent algorithms, total runtime of the system decreased because there are few messages passing through network compared to Giraph. Table 3-2 is showing the monetary cost of processing for each algorithm on different frameworks.

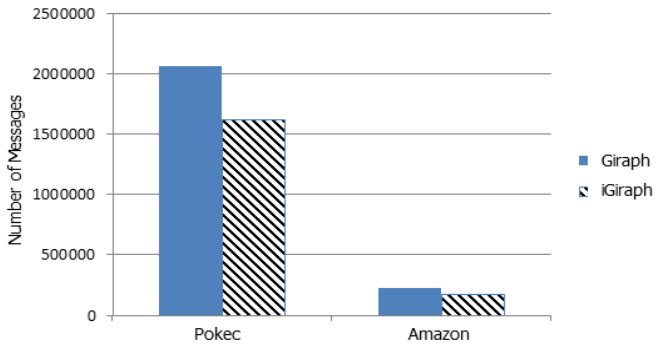


Figure 3-15 The average number of network messages in each experiment

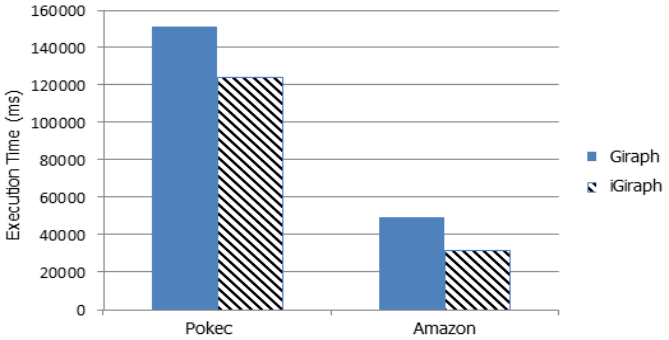


Figure 3-16 Total time taken to perform PageRank algorithm

Table 3-2 Processing cost on different frameworks

Dataset	Giraph (SSSP)	iGiraph (SSSP)	Giraph (PR)	iGiraph (PR)
Amazon	$0.0215	$0.0146	$0.0320	$0.0232
YouTube	$0.0140	$0.0092	$0.0502	$0.0410
Pokec	$0.0235	$0.0170	$0.0816	$0.0725

3.6 Related Work

According to The National Research Council of the National Academies of the United States [55], graph processing is one of the seven computational giants of massive data analysis. Google's Pregel [148] is the first graph processing framework in the literature that uses a bulk synchronous parallel (BSP) model [231] for graph computation based on a vertex-centric approach. Public implementations of this framework include Giraph, GoldenOrb [36], etc. These frameworks are developed based on distributed architectures in which usually one machine acts as the master and one or several other machines act as workers. In the master-worker approach, the input graph is split into partitions and each partition assigns to a worker to process it. Many of graph processing frameworks use a simple hash function for partitioning the graph. However, such simple partitioning leads to huge network traffic in a graph processing task that consequently affects the system performance. To improve the partitioning efficiency, various approaches are proposed in different frameworks [69] [34] [20]. While most graph processing systems offer some specified improvements on HPC clusters with fast interconnects, their conduct on virtualized commodity hardware which is provided by cloud computing paradigm and is accessible to a wider population of users is less investigated [187].

Frameworks designed to process large-scale graphs based on Pregel are called Pregel-like frameworks. They are designed based on distributed architecture on high performance computing systems such as distributed clusters. Although graph processing systems created to overcome previous large data processing solutions such as MapReduce, some of distributed frameworks use series of MapReduce jobs

iteratively. Giraph and Surfer are examples of these systems. Other features of Pregel-like frameworks include using bulk synchronous parallel (BSP), message passing communication method and global synchronization barrier between supersteps. However, systems such as GraphLab [142] provide asynchronous computations. Since iGiraph is a Pregel-like system and developed based on Giraph, it contains all of these specifications with some additional features such as dynamic repartitioning and cost minimization. There are many non-Pregel graph processing frameworks developed on distributed architecture. Among these frameworks are Trinity [202] and Presto [233].

GPS is the most similar to our work. It has an optimization called LALP (large adjacency-list partitioning) by which stores high degree vertices and use the list to send one message, instead of thousands for instance, to the partitions are containing those vertices. After the message gets to the destination, it will be replicated thousands times to the message queues of each vertex in its outgoing neighbors list. Instead of storing the list of vertices, iGiraph stores two lists of the number of outgoing and incoming messages from/to each partition that show which partitions are sending or receiving more messages. These lists are noticeably smaller than GPS's adjacency lists.

Another difference between our system and GPS is that high degree vertices in GPS are defined by the programmer, but in iGiraph, the decisions about migrating the partitions are making based on an automatic formula. In GPS, the programmer specifies a parameter τ. If the number of outgoing messages for any vertex is more than τ, it will be considered as high degree. Here, selecting the right value for τ is very important and can directly affect the system's performance.

There are previous studies on the performance effects of different partitionings of graphs on other systems. The main challenge in partitioning a graph is to find how to partition the data to gain better vertex or edge cuts with considering the simplicity of computation. Pregel, Giraph and GraphLab partition the graph by cutting the edges while PowerGraph [80] and X-Stream [192] cut vertices for partitioning. From another point of view, the majority of graph processing frameworks only use static partitioning approaches that means they only partition the graph once before the processing starts or they do it once during the computation. On the other hand, some frameworks such as GPS use dynamic repartitioning approach that allows them to repartition the graph

multiple times during the computation based on some pre-defined features to achieve better performance.

3.7 Summary

Huge amount of data is created and stored in the form of graphs every day. In this chapter, we presented iGiraph, a Pregel-like system developed based on Giraph for processing large-scale graphs on public clouds. iGiraph uses a new repartitioning method to reduce the number of messages passing through network by decreasing the number of cross-edges between partitions. It utilizes high degree concept in partition level for both convergent and non-convergent types of algorithms. iGiraph also considers processing large graphs as a service on public clouds. Therefore, it reduces the cost of resource utilization by decreasing the number of workers that are using for the operation and executes the applications within a period which is reasonably close to Giraph's time.

Chapter 4
Network-aware Dynamic Repartitioning for Scheduling Large-scale Graphs

Large amount of data that is generated by Internet and enterprise applications is stored in the form of graphs. Graph processing systems are broadly used in enterprises to process such data. With the rapid growth in mobile and social applications and complicated connections of Internet websites, massive concurrent operations need to be handled. On the other hand, the intrinsic structure and the size of real-world graphs make distributed processing of graphs more challenging. Low balanced communication and computation, low preprocessing overhead, low memory footprint, and scalability should be offered by distributed graph analytics frameworks. Moreover, the effects of network factors such as bandwidth and traffic as well as monetary cost of processing such large-scale graphs and the mutual impact of these elements have been less studied. To address these issues, we proposed two dynamic re-partitioning algorithms that consider network factors affecting public cloud environments to decrease the monetary cost of processing. A new classification of graph algorithms and processing is also introduced which will be used to choose the best strategy for processing at any operation. We plugged these algorithms to our extended graph processing system (iGiraph)

This chapter is partially derived from:

- **Safiollah Heidariand** Rajkummar Buyya, "Cost-efficient and Network-aware Dynamic Repartitioning-based Algorithms for Scheduling Large-scale Graphs in Cloud Computing Environments", Software: Practice and Experience (SPE), vol 48, Issue 12, pp: 2174-2192, Wiley & Sons, 2018

and compared them with those supported in other graph processing systems such as Giraph and Surfer on Australian National Cloud Infrastructure. We observed that up to 30% faster execution time, up to 50% network traffic decline and more than 50% cost reduction is achieved by our algorithms compared to a framework such as popular Giraph.

4.1 Introduction

TODAY many applications in domains such as the Internet, astronomy, social networks, information retrieval and particle physics are experiencing data flood and they have already reached peta-scale volume of data. The growth in the volume of data needs large computing power to turn the original data into worthwhile insights. Nevertheless, massive amount of data is saved and modeled in the form of graphs. These graphs provide valuable sources of information for several applications. For instance, by studying social networks and the way that relationships are shaped between users, psychologists and sociologists can investigate their assumptions and hypothesis about people and communities. Analyzing web graphs can make search engines more accurate and effective [173]. By detecting social circles and their influential members in social networks, politicians can spread their thoughts in these communities [256]. Therefore, processing large-scale graphs and unveiling attributes of those graphs have become critical requirements.

Traditional approaches of processing Big Data such as MapReduce [56] are not suitable for graph processing because of the intrinsic behavior of graph algorithms. For example, MapReduce has a two-phase computation model – Map and Reduce, which is not exactly appropriate for the iterative nature of graph algorithms. It also does not retain the input graph and its states in main memory across these two phases and is very inefficient because of requiring repetitive disk I/O.

Many research works on large-scale graph processing frameworks concentrate on the platforms based on commodity clusters. However, not many studies have been

done on cloud platforms, particularly public clouds. Cloud computing is a model of computing that has modified hardware, software and datacenters implementation and design [35]. It has brought novel technologies and economical solutions such as elasticity and pay-as-you-go models by which service providers do not need to worry about previous obstacles of delivering services to their clients. Public cloud services are getting more popular than other cloud services such as private, hybrid and community clouds especially among small and medium size businesses. It is because they do not have sufficient funding to have their own private cloud or it is not efficient for their business models. So, public cloud is a true response to their needs. Another important feature of public clouds is the monetary modeling that different service providers offer to their customers. Amazon, for example, has three cost models: spot, on-demand and reserved provisioning models for providing resources. Using these commercial services, the client might select to pay more to get higher performance or better reliability. So, the challenge with using public clouds is making the right decision between utilizing the number of resources that the user needs and the amount of money he/she can pay for the service. In this research we only consider the reserved model.

Another less studied aspect of graph processing systems is the impact of the network environment on the performance of the whole system. Some systems such as Surfer [46] and Pregel.Net [187] are implemented to support graph processing on public clouds. Although, they consider some network features, none of these systems have explored the effects of provisioning and processing on monetary cost. For instance, Surfer proposes a graph partitioning approach based on the network bandwidth and claims that it could improve the performance. On the other hand, to the best of our knowledge, all existing graph processing frameworks –except iGiraph - concentrate on decreasing the processing runtime, memory utilization and so on to degrade the cost of operation. They only take an unchanged pool of resources with known size into consideration. It means that all existing systems start and finish their computation with the same number of resources (machines). So, in many cases, idle machines have to wait for other busy machines to finish their jobs and all machines be released together which is a waste of money and time. Even systems such as

GraphP[261] and GraphR [211] that are reducing the number of messages passing between partitions and introducing new memory access techniques respectively, do not discuss the monetary costs of the processing. These limitations can be overcome by dynamic management of resources in an elastic manner.

The aim of this chapter is to develop scheduling algorithms that consider characteristics of application workloads and resources along with network factors to improve the performance and reduce the monetary cost of the whole computation. We propose a novel dynamic re-partitioning method that utilizes different factors including: a) the type of the graph application that is going to be used, b) some intrinsic features of natural graphs such as high-degree vertices, and c) the network features of the cloud environment that the system is running on. Our algorithms were plugged in to our extended version of graph processing framework (iGiraph) and we compared them with those supported in other graph processing systems such as Giraph and Surfer on Australian National Cloud Infrastructure. We observed that up to 30% faster execution time, up to 50% network traffic decline and more than 50% cost reduction is achieved by our algorithms in comparison with a framework such as popular Giraph. Our work makes the following **contributions:**

- A new classification of graph applications and processing is introduced in this chapter which affects the policy that will be chosen to process the input graph. We have studied the impacts of combinations of different situations from this classification together on processing large-scale graphs on public clouds for the first time and reduced the monetary costs in each situation.
- A novel mapping strategy is designed to facilitate assigning partitions to the workers based on different features that each partition and worker has.
- A new bandwidth-and-traffic-aware dynamic re-partitioning algorithm and a new computation-aware re-partitioning algorithm have been proposed in this chapter. These algorithms remarkably reduce the monetary cost of processing - which is a vital factor in the procedures of selecting services for any customer on a public cloud

The rest of the chapter is organized as follows: Section 4.2 explains the related work. A new classification of graph algorithms is explained in Section 4.3. Section 4.4 and 4.5

introduce our new proposed bandwidth-and-traffic-aware and computation-aware dynamic re-partitioning algorithms of large-scale graphs respectively, with their implementation on iGiraph. We explain the architecture and details of our system (iGiraph-network-aware) in Section 4.6 followed by a discussion on the evaluation of our works in Section 4.7. Finally, Section 4.8 concludes the chapter and proposes future works.

4.2 Related Work

To overcome the issues on traditional processing approaches, considerable endeavors are made to process large graphs. Some proposed systems try to process the entire graph on a single server whereas the main problem of this method is scalability [144]. However, the utmost size of graph to be processed is restricted by the single host's memory in which the input graph has to be fully loaded. In addition, this method cannot use the strength of other hosts in terms of distribution and parallelization, to reduce the processing time. Another method is to utilize libraries that allow graph algorithms to be executed in parallel in the shared memory approach [205]. This method, tries to solve the issue of the previous method. However, it still has problems with fault-tolerance and scalability [85]. Another way of processing graphs is to adopt graphic processing units (GPU) to accelerate different graph processing tasks. In sampling method, the input graph will be divided into several sub-graphs by the system and then the attribute of the main graph will be estimated based on the attributes of the smaller sub-graphs. The major issue in this method is that there is a big distinction between the actual and estimated solutions.

Unlike the aforementioned methods, a distributed method utilizes a commodity of servers as a generic solution to performance, scalability, and availability issues [53]. This can be specifically utilized for solving large graph problems. Pregel [148], which was proposed by Google in 2010, is a computational model dedicated for processing large-scale graphs. The main inspiration for Pregel is the Bulk Synchronous Parallel (BSP) model [231] which streamlines the implementation of distributed graph algorithms. A program in Pregel contains sequences of iterations called *superstep*.

Within a superstep, a user-defined function called *Compute()* is invoked by Pregel for each vertex that specifies the conduct of the node in the superstep. The *Compute()* reads messages that have been sent to the related node during the prior iteration, applies some processing and dispatches messages to other nodes, that will be collected at the next superstep. This function can also change the states of vertices and their outgoing edges. Pregel uses supersteps to accomplish fault tolerance and high scalability in a cluster of machines. Nevertheless, this might be an impasse for performance when the amount of communications grows in a graph with vertices in millions-scale. Many distributed graph processing frameworks have been introduced after Pregel. Systems such as Giraph [11], Apache Hama [13], ExPregel [193], GPS [195], GraphLab [142] and iGiraph which have been developed based on Pregel are called Pregel-like systems. There are also other frameworks that are not developed based on Pregel.

Pregel-like frameworks are developed based on a distributed architecture in which one machine will act as the master while other machines will be workers (slaves) and do the computation. In this approach, the input graph is splitted into partitions and partitions are assigned to workers by the master to be processed. Therefore, partitioning a graph is a critical job and since it has a direct influence on the performance of the system, various methods have been proposed for achieving better outcomes. A vast majority of graph processing systems propose some determined improvements on high performance computing clusters with fast interconnects. However, their behavior on cloud computing that provides virtualized commodity hardware and is available to a broader crowd of users is less investigated.

Despite introducing various partitioning methods by different frameworks, the impact of network factors on the system's performance and the way that they can be used to optimize or improve the processing is not sufficiently studied. Surfer [46], is the closest framework to our proposed system. But according to the earlier discussion, it has many shortcomings and does not cover many aspects of network bandwidth; particularly its mapping strategy is not quite efficient. Another system that considers network traffic is Pregel.Net. Pregel.Net [187] is implemented based on Pregel but over .Net framework. It has used Microsoft Azure to analyze the impact of BSP graph

processing model on public clouds. However, it does not investigate if its changes will affect the monetary cost of the operation.

Table 4-1 Comparison of the most related works in the literature

System	Architecture	Partitioning Method	Traffic-aware	Bandwidth-aware	Computation-aware	Resource Scheduling
Pregel [148]	Distributed	Static	×	×	×	Static
Giraph [11]	Distributed	Static	×	×	×	Static
GPS [195]	Distributed	Dynamic	√	×	×	Static
GraphX [81]	Distributed	Static	×	×	×	Static
Surfer [46]	Distributed	Dynamic	×	√	×	Static
iGiraph (Chapter3)	Distributed	Dynamic	√	×	×	Dynamic
Our work - iGiraph-network-aware	Distributed	Dynamic	√	√	√	Dynamic

In another research [172], authors have shown that the network does not have a significant impact on the processing and the highest impact that any optimization solution can bring to graph processing system's performance would be something between 2%-10%. GraphX [81] and Spark[18] were used in that experiment and some network factors such as the speed of the network was studied in different situations. However, McSherry [154] showed that this assumption is completely wrong and many other factors have been missed from the study. He showed that using a dataflow framework can achieve much better results to 2X-3X compared to GraphX. This study and ours in this chapter imply that there are still many features that can be taken into consideration and be mixed with novel solutions to leverage the impact of network to reach better performance. Table 4.1 demonstrates the features of some of the most related works in the literature.

In this chapter, we extend iGiraph to support more network factors for its dynamic re-partitioning approach by providing a novel priority mapping solution to customize each machine for each partition. According to this solution, we provide a ranking method for this mapping. To distinguish between basic iGiraph and our proposed

[18] http://spark.apache.org/

network-aware system in this chapter, we refer to the new system as "**iGiraph-network-aware**" for the rest of the chapter.

4.3 Processing Environment Categorization and Graph Applications

Different works use different categorizations for graph applications. For example, GBASE [113] and TurboGraph [91], categorize the queries to *global queries* and *targeted queries*. Algorithms such as diameter estimation that need to traverse the entire graph are identified as global queries while other algorithms such as single source shortest are put in the targeted queries category. A framework such as Giraph++ [223] uses three categories including graph traversal, random walk and graph aggregation for graph algorithms while iGiraph utilizes a one dimension categorization for all graph applications which divides them into convergent and non-convergent.

In this work, we extend the iGiraph's categorization into two-dimensions by adding an extra layer. Figure 4.1 shows the new categorization for all sorts of processing where any kind of application can be either computationally-intensive, communicationaly-intensive or a combination of them.

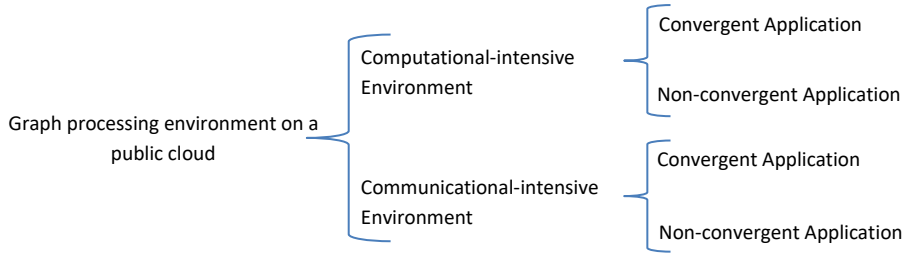


Figure 4-1 Graph applications and processing environment categorization

- *Computationally-intensive processing*: This type of processing often has a large impact on CPU utilization because it spends more time on computing than communication and the memory side. Sometimes the graph processing application itself is computationally intensive and sometimes other applications keep the CPU busy in VMs and the graph application has to find a way to be processed faster. This situation happens mostly in a public cloud

environment.

- *Communication-intensive processing*: This type of processing usually has a big impact on network and memory especially when an application needs to keep the intermediate states of a computation.

In this chapter, we utilize two typical algorithms (convergent and non-convergent) to show the impacts of each algorithm on both types of processing. Here, we give a brief description of sample algorithms that we are going to use for our experiments.

1. *PageRank:* PageRank algorithm was proposed to weigh the importance of web pages and websites by calculating the number of links connected to them. The more hyperlinks the page gets from other websites, the more significant the page is. PageRank assesses every page individually and will not weigh the whole website as a unit. In this algorithm, the importance of a typical web page will not be affected by the PageRank of other pages because each page has its own exclusive approximated weight. According to the categorization we presented in this section, PageRank is a non-convergent algorithm due to generating a constant number of messages in each iteration during the processing.
2. *Single source shortest path*: The aim of solving the shortest path problem is to find a route between two nodes in a graph while the sum of the weights of its edges is minimized. Shortest path is a famous problem in graph theory and various approaches have been suggested to solve it. Single-source-shortest-path (SSSP) problem is a special case of the original shortest path problem. SSSP is about discovering the shortest route between a typical source vertex and all other nodes in the graph. Before SSSP starts, the values (distance) of all vertices are set to INF (∞) except the source vertex which is set to 0. Any possible route from the source vertex in the graph will be shorter than INF. During each superstep, vertices receive messages from their adjacent nodes, update their value using the minimum value received from their neighbors and send any recently found minimum value to all neighbors. In the initial iteration, only the adjacent vertices of the source node will be updated. In each superstep, the

updated nodes will send their new values to their neighbors until the computation ends. The processing finishes when the status of all nodes in the graph is changed to *inactive* and no more updating happens. According to this definition, SSSP is categorized as a convergent algorithm.

The total cost of processing in a graph system is depending on two major factors (considering equal size for the messages in the network): 1) the number of machines, and 2) the time in which a specific type of machine is being used, as shown in Equation 4.1.

$$\text{Cost}_{\text{Total}} = \sum_{j=0}^{m}\sum_{i=1}^{n}(C(VM_i)\times T(VM_i)) \qquad (4.1)$$

In the above equation, $C(VM_i)$ is the price of each machine and $T(VM_i)$ is the time within the machine is used. To reduce the total cost of the operation, either less costly machines must be used, or the total time that each machine is being used should be reduced. In order to achieve this in a graph processing system, partitioning plays an important role. For instance, there is no need to keep all the initial machines in the system for convergent algorithms if there is a way to repartition the graph and place the remaining of the graph on less number of machines and reschedule the resources. In this chapter, we show that to provide an effective dynamic repartitioning mechanism, considering factors such as traffic, bandwidth and computation burden in the network can help to reduce the monetary cost and improve the performance.

4.4 Bandwidth-and-Traffic-aware Graph Scheduling Algorithm with Dynamic Re-partitioning

Assume that the average amount of network traffic sent along each cross-partition is $N_M(P_i, P_j)$, the networks bandwidth between the machines stored P_i and P_j to be $B_{i,j}$, and $C(P_i , P_j)$ to be the number of cross-partition edges from partition P_i to P_j. Because network bandwidth is a scarce resource in the cloud environment, it is considered as the major index for network performance. So, the approximate data transfer time (DTT) from P_i to P_j will be as follows:

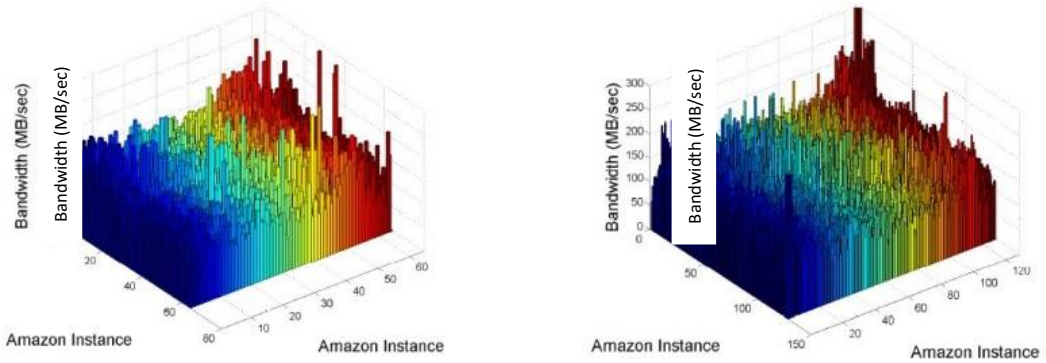
$$\mathrm{DTT}(i,j) = \frac{C(P_i, P_j) \times N_M\left(P_i, P_j\right)}{B_{i,j}} \tag{4.2}$$

This estimation is adequate for large-scale graph processing in both public and private cloud environments. Suppose we have stored P graph partitions on P disparate machines; the overall data transfer time ($\mathrm{DTT_{Total}}$) in the network caused in all partition pairs is as follows:

$$\mathrm{DTT_{Total}} = \sum_{i=0}^{P-1} \sum_{j=0}^{P-1} DTT(i, j) \tag{4.3}$$

Obviously if network bandwidth amongst different machine pairs is constant, the total network data transfer time will be minimized when the total number of cross-partition edges is minimized. Nevertheless, the network bandwidth amongst different machine pairs can change remarkably in the cloud. Cloud providers have noticed such network bandwidth unevenness. The network bandwidth of every machine pair amongst 64 and 128 small Amazon EC2 instances is shown in Figure 4.2. On the other hand, research shows that in public cloud, the network bandwidth between two instances is provisionally steady. This allows us to perform our mapping calculation before each superstep.

Because of the network bandwidth unevenness, an important factor for an efficient graph processing is the mechanism of partitioning the graph and storing its partitions on the VMs. According to [15], because there might be a large number of partitions and workers for processing the graph, there is P! possible ways to store partitions on workers which is a huge solution space. Another issue is finding a solution by which both graph processing and graph partitioning algorithms can be aware of the bandwidth variability for networking efficiency.



(a) (b)

Figure 4-2 Network bandwidth unevenness in Amazon EC2 small instances with (a) 64 instances and (b) 128 instances [46]

To address these issues in a public cloud environment, a new dynamic re-partitioning method is proposed in this chapter. The idea is to place the partitions with larger number of high-degree border vertices – which means they have larger number of cross-partition edges – on workers with higher network bandwidth. This is because those graph partitions need more network traffic. It also helps the partitions to be processed faster.

To achieve performance improvement, we implemented a *mapping strategy* (illustrated in Figure 4.3) in iGiraph. The processing starts with a random partitioning approach as we use this method for all our experiments to start with. This is because random partitioning is shown to have the worst performance among most of the existing well-managed partitioning approaches. So, we aim to improve this situation as the cheapest implementing strategy which is not good performance-wise. According to this strategy, the first iteration (superstep 0) starts with a random partitioning method, the processing of the iteration completes and the global synchronization barrier occurs. Before going to the next superstep, we use the information we collected from the first iteration to plan a new partitioning (re-partitioning) for the next iteration.

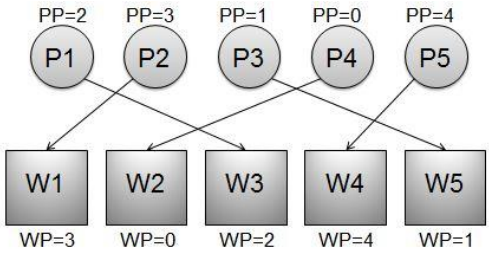


Figure 4-3 Mapping strategy for 5 partitions and 5 workers. Partitions with higher priorities are assigned to the machines with higher bandwidth

After the completion of the first superstep, each partition is assigned a factor called Partition Priority (PP). The partition with PP=0 is the one that receives the larger number of messages over the network when compared to other partitions. In other words, this partition contains more high-degree border vertices than other partitions. It is also a candidate for being merged with other partitions or its vertices being migrated to other partitions. All other partitions also get their own PP which shows their importance based on the amount of network traffic they generate. On the other hand, each worker also will be assigned a factor called Worker Priority (WP). The worker with WP=0 is the one with the highest bandwidth among all workers (machines). All other workers also will be given their own WP based on their bandwidth rating in the network. In case in which two or more partitions have the same priority after calculation, one of them will get the higher PP randomly. The same logic also applies to workers. After assigning PPs and WPs to partitions and workers respectively, the partitions with specific PPs will be assigned to the workers with the same WPs (Figure 4.3). The calculations and assignments are done after each superstep *i* and before each superstep *i+1.*

Algorithm 4.1: Bandwidth-and-traffic-aware dynamic re-partitioning

```
1:  Partition the graph randomly
2:  Set PP=0 for each partition and WP=0 for each worker
3:  For the rest of the computation do
4:      Calculate PP for each partition based on the number of messages that each
        partition receives
5:      Calculate WP for each worker using end-to-end mechanism
6:      If global synchronization happened then
7:          Merge the partitions or migrate vertices if needed
8:          Set the priorities based on PP and WP
9:          Map partitions(based on PP) and workers(based on WP)
10:     If VoteToHalt() then
11:         Break
```

Another issue that should be considered is the time when the priority setting should be done. Due to the possibility of merging or removing the partitions after each superstep, the priority setting is done after these operations, immediately before the next iteration starts. Therefore, the partitions that have received migrated vertices will be given the highest priorities. This is because the reason for vertex migration is to bring high-degree vertices closer to their neighbors. If there is more than one partition receiving migrated vertices, the one that has got more migrated vertices will get the highest priority and so on. Also for the partitions that get merged, the priority of the final partition (combined partition) will be set as the priority of the partition with highest priority (its priority from the previous iteration). At the beginning of the processing (superstep 0), all partitions' priorities will be set to 0 (highest priority).
In a nutshell, according to Algorithm 1, after each superstep, initial priorities for the partitions and workers will be calculated based on the measurement of various network factors (traffic and bandwidth here) that have been completed during the iteration. Then, if needed, partition merges and vertex migrations might happen based on the aforementioned mechanism. Eventually, final priorities will be set for partitions and workers and they will be mapped accordingly.

According to our experiment results (Section 4.7.2), using a mapping strategy that assigns partitions to workers based on the traffic in the network and the bandwidth capacity of workers, combined with iGiraph's re-partitioning method (for both convergent and non-convergent types of algorithms) gives much better results compared to previous solutions. These results would be in regard to reducing the monetary cost of the processing by reducing the cost of resource utilization, reducing network traffic and accelerating the execution time of the whole process.

4.5 Computation-aware Graph Scheduling Algorithm with Dynamic Re-partitioning

Although many graph algorithms are communication intensive, computation unit can still affect the execution of applications. In a public cloud, each VM can host different applications at the same time. Some applications might be computation-intensive and

keep the CPU busy while other applications are not very CPU-dependent but still can be affected by the former. Computation-intensive algorithms or applications can delay the computation and execution time of others.

Various approaches can be applied to deal with such situations. For example, each job can have a different priority by which the host can schedule the computation time for that. There are many prioritization strategies such as first-in-first-out, first-in-last-out, assigning priority numbers to tasks, etc. Another approach for when there is no priority or preference for job execution can be using equal time-slots for computing jobs in an intertwined way.

Algorithm 4.2: Computation-aware dynamic re-partitioning

```
1:  Partition the graph randomly
2:  Set PP=0 for each partition and WP=0 for each worker
3:  For the rest of the computation do
4:      Calculate PP for each partition based on the number of messages that each
        partition receives
5:      Calculate WP for each worker using CPU utilization on each worker and
        CPU idle time
6:      If global synchronization happened then
7:          Merge the partitions or migrate vertices based on machineType if
            needed
8:          Set the priorities based on PP and WP
9:          Map partitions(based on PP) and workers(based on WP)
10:     If VoteToHalt() then
11:         Break
```

We propose a similar *mapping strategy* as we discussed for traffic and bandwidth-aware re-partitioning, but we consider CPU utilization instead of bandwidth in the algorithm and re-partition the graph differently. We have implemented this strategy on iGiraph. As in last section, the computation starts with a random partitioning for superstep 0. At the end of superstep 0 when the global barrier happens –before superstep 1- we use the information we have got so far to initiate the re-partitioning.

At this stage, on one side based on the number of messages that has been passed between workers through the network, we define Partition Priority (PP) again by which the partitions with high-degree vertices can be recognized. On the other side, a

scalable monitoring tool called Ganglia[19] is used to monitor the CPU utilization on each worker. Therefore, the information regarding the computational conditions of all machines will be written and saved on a separate file on the master machine. The information include the percentages of CPU idle times at the end of each supestep so that it can be possible to find which machines are still busy, or how busy they are, and which one is free and ready to use. The reason for choosing the CPU idle time to use in the algorithm instead of CPU working time is that the former is more reliable. There might be situations that a very small task can use most of computation resources for a short time and increase the utilization percentage remarkably but the reality is that the CPU will be idle the rest of the time. From this information, a map of available computation resources can be depicted which will be used for dynamic re-partitioning during the rest of computation. Figure 4.4 shows the computation map of a system with 15 workers where some random computation-intensive applications are running on some machines.

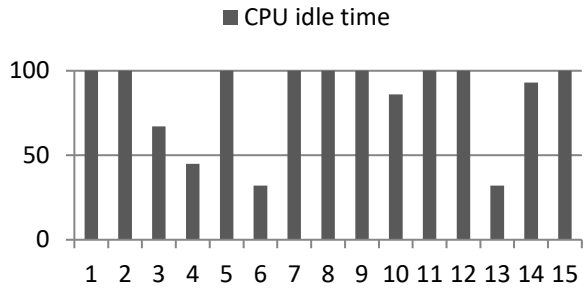


Figure 4-4 Percentage of CPU idle time in a system with 15 workers

According to the aforementioned strategy, there will be four types of machines in the environment after the first superstep: 1) a machine with both a computation-intensive application and high-degree vertices of graph dataset on it, 2) a machine with computation-intensive application running on it but the graph partition that have been assigned to that does not have high-degree vertices, 3) a machine with a partition containing high-degree vertices but no computation-intensive application on it, and 4)

[19] http://ganglia.sourceforge.net/

a machine that has neither computation-intensive application running on it nor the partition that have been assigned to it has any high-degree vertices.

The idea is to move high-degree vertices with their neighbors to the machines that have higher CPU idle time. This is because more computation is needed to be done on these vertices in terms of the number of messages they receive. So the algorithm would be like this: partitions in machine type 1 need to be migrated to or merged with partitions on machines type 4 or 2 respectively. Partitions on machine type 2 can be merged with the one on type 3 and 4. Partitions on machine type 3 can be migrated to type 4 or be merged by partitions on machines type 2. Based on this algorithm, at the start of the processing, all workers have their types set as 0 which will change after the first superstep. Then, at the end of each superstep, this algorithm will re-partition the graph and assign the proper partitions to their best worker. iGiraph-network-aware also considers the available memory on the destination before moving the vertices.

To summarize, similar to Section 4.4, after each superstep, initial priorities for the partitions and workers will be calculated based on the measurement of various network factors which in this case are the traffic and the computation capacity of each machine. After that, some partitions might get merged and some vertices might get migrated using the dynamic repartitioning mechanism. Finally, priorities will be set for partitions and workers and they will be mapped accordingly (Algorithm 2). The algorithm will be terminated when there is no more active vertices to be processed.
As will be shown in Section 4.7, our experiments prove that under the equal situation, the computation-aware re-partitioning on iGiraph-network-aware significantly reduces the execution time of the entire processing compared to Giraph. It is also shown that this approach can reduce the monetary cost of the processing for both convergent and non-convergent types of applications.

4.5.1 Complexity Analysis

We analyzed the time complexity of the two proposed algorithms (traffic-and-bandwidth-aware and computation-aware algorithms) which are very similar in terms of the structure. Both algorithms are dependent to the number of supersteps *(N)* which *N* varies based on the application and the number of vertices in the graph. Also,

prioritizing partitions *(P)* and worker machines *(W)* affect the algorithms as they need to be calculated in each iteration. Therefore, the complexity of these algorithms is *O(N(P+W))*. Since both *P* and *W* are dependent to the number of machines *(m)* (one partition per worker), the complexity also can be written as *O(N(log m))*.

On the other side, the complexity of partitioning algorithm for Surfer is $O(m^2)+O[P+\log P\ (n+\log P)]$ where *P* is the number of partitions and random partitioning is used instead of METIS. For Giraph the complexity is *O(N(n))* (*n*=number of nodes). As can be seen, algorithms are dependent to the applications' complexities as well. According to [27], for instance, the complexity of SSSP and CC algorithms are *O(ne)* and *O((e+n)log n))* respectively, where *n* is the number of nodes and *e* is the number of edges in the graph. In Surfer, the user should define the number of partitions for the processing hence the complexity of the algorithm is dependent to the number of partitions *(P)*.

4.6 System Design and Implementation

Figure 4.5 shows the design of our proposed software system and the components that we have added to iGiraph. The architecture and placement of different components of our system is shown in Figure 4.6.

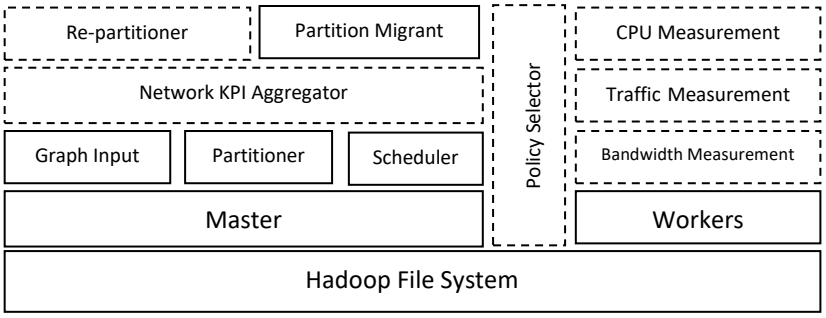


Figure 4-5 The components that we added to original iGiraph are shown in dotted rectangles

4.6.1 Bandwidth Measurement

Bandwidth measurement component is implemented on all machines in the system to be able to calculate the bandwidth between workers by an end-to-end calculation mechanism which is used in [265].

4.6.2 Traffic Measurement

To calculate the network traffic between each pair of machines, the traffic measurement module is implemented and installed on all workers. It basically works based on the number of messages transferring between machines. Using this information, the system ranks the most congested paths and uses that for partitioning purposes.

4.6.3 CPU Measurement

As part of a network characteristic, CPU workload shows the amount of computations occurring on each qmachine and in the whole network. In a public cloud, there may be different jobs running on each machine at the same time and some of these jobs might be computation-intensive. By knowing how busy each worker in the network is, we can avoid overloading occupied workers by assigning more tasks to them. This module uses the information that it receives from Ganglia- a tool by which we can measure many specifications of a network- to calculate the CPU idle times per worker.

4.6.4 Policy Selector

Policy selector is a component of iGiraph which we have expanded to cover our network-aware scheduling algorithms. Using this component, users specify their workloads and based on that they define what algorithm (bandwidth-aware or computation-aware) they want to be used to process their workload.

4.6.5 Network KPI Aggregator

The network KPI aggregator is implemented on the master to aggregate the information from all workers and pass them to the next component for partitioning decision making. Having this component as an independent module that gathers all information in one place helps to reduce the burden of workers and make the execution faster.

4.6.6 Re-partitioner

The re-partitioning component partitions the graph again based on the information that has been gathered from other parts of the system. Since the system utilizes a synchronous approach for execution, re-partitioning happens after each superstep and before the next superstep begins. We will show that using these components and the new re-partitioning strategy, the performance of the system will increase significantly compared to similar frameworks such as Giraph and Surfer.

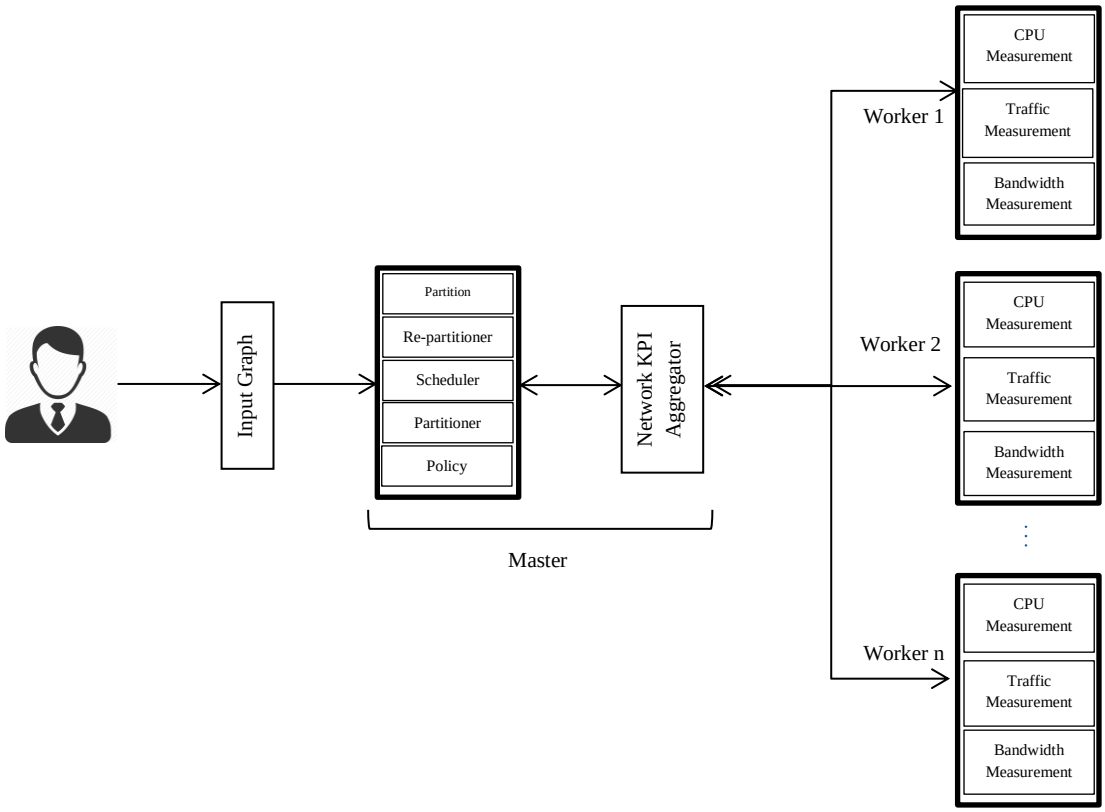


Figure 4-6 System architecture

4.7 Performance Evaluation

4.7.1 Experimental Setup

We use *m1.medium* NECTAR VM instances for all partition workers and the master role. NECTAR [163] is the Australian national cloud infrastructure facilities. Medium instances have 2-cores with 8GB RAM and 70GB disk including 10GB root disk and 60GB ephemeral disk. All the instances are in the same zone and use the same security policies. 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. Hence, NECTAR m1.medium price is put based on AWS m5.large Linux instance which costs $0.12 per hour. However, because our experiments are in second scale (instead of hour scale), the prices are being calculated for the entire operation in second scale. So, we charge the machines only based on the number of seconds they were used and do not charge them for one hour because they were used only for few seconds. We also installed NECTAR Ubuntu 14.04 (Trusty) amd64 on each instance. We plugged in our algorithms to iGiraph (our extended version of Giraph system) with its checkpointing characteristic turned off. To distinguish between the original iGiraph and the current work, we refer to the new system as "**iGiraph-network-aware**" in this chapter. We also use Apache Hadoop version 0.20.203.0. All experiments run using 16 instances where one takes the master role and others are set up as workers.

We chose shortest path and PageRank for communication-bound convergent and non-convergent algorithms respectively. Also to show the effectiveness of the distributed processing on large-scale graphs by using our proposed solution, we utilize three real datasets of different sizes: Amazon, YouTube and Pokec [125] . Properties of these datasets are shown in Table 3.1.

4.7.2 Results

To evaluate the proposed algorithm we chose Giraph as a popular graph processing framework to compare the performance of our system with. We also implemented the bandwidth-aware graph processing method proposed by Surfer on Giraph to use it as

another baseline. Although Giraph has been improved since Surfer was developed, the implemented algorithm still shows Surfer's behavior on the network. We have also compared the results with original iGiraph (chapter 3). In addition, the size of messages in all experiments is the same. Therefore, the communication cost is independent from message size and is calculated based on the number of messages that are transferred through the network.

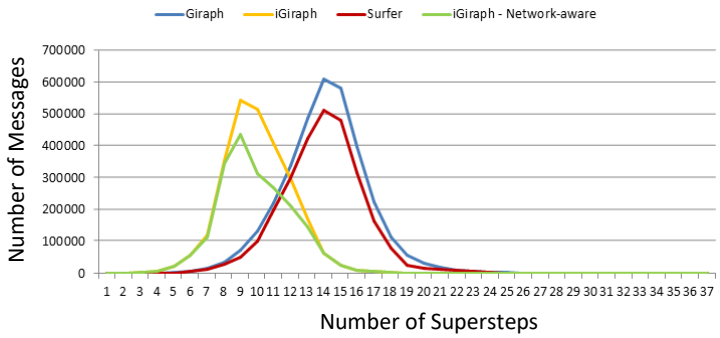


Figure 4-7 Number of network messages transferred between partitions across supersteps for Amazon graph using shortest path algorithm

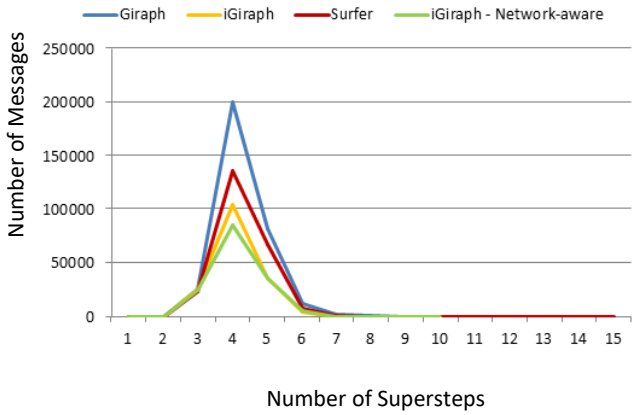


Figure 4-8 Number of network messages transferred between partitions across supersteps for YouTube graph using shortest path algorithm

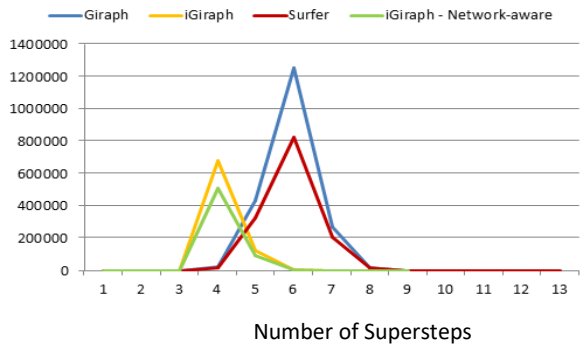


Figure 4-9Number of network messages transferred between partitions across supersteps for Pokec graph using shortest path algorithm

The first group of experiments is carried out for communication-intensive scenarios. Most graph processing applications are classified in this category. As the results show, iGiraph-network-aware could achieve better performance compared to Giraph, original iGiraph and Surfer on both convergent and non-convergent applications. Both Giraph and Surfer start computing with a constant number of machines and finish the computation with the same number of machines; no matter if the graph is shrinking or not during the execution. On the other hand for convergent algorithms, as the processing continues, the number of active vertices decreases. So, iGiraph and iGiraph-network-aware remove deactivated vertices from the memory which means the graph is shrinking during the processing. Our experiments (Figures 4.7-7.9) show that the number of messages in the network is reduced even more significantly compared to original iGiraph by using dynamic bandwidth-and-traffic-aware re-partitioning and mapping approach on iGiraph-network-aware. This leads to reducing the number of active workers during the processing. As a result, when the number of machines declines, the cost of processing will also drop significantly. The results even show that the number of workers tends to be reduced faster compared to original iGiraph (chapter 3) because using the new algorithms in this chapter, the number of messages in the network is decreasing too. This also affects the total execution time as illustrated in Figure 4.11.

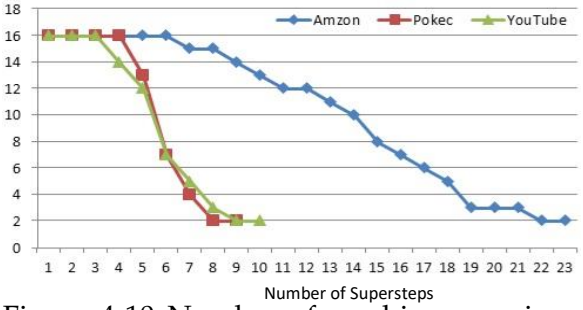


Figure 4-10 Number of machines varying during supersteps while running shortest path algorithm on different datasets

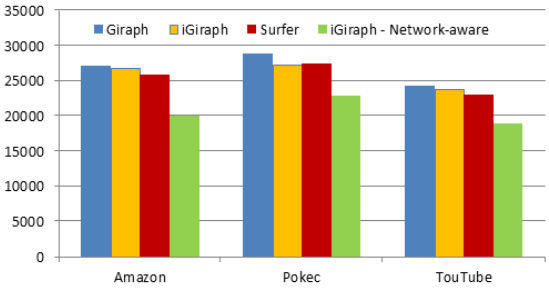


Figure 4-11 Total time taken to perform shortest path algorithm

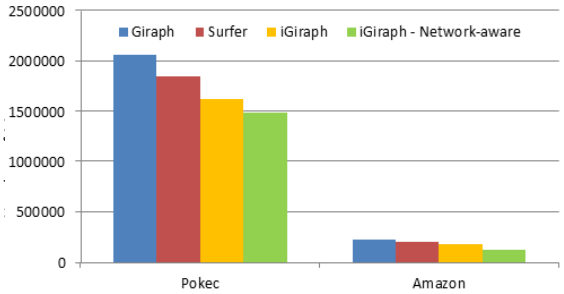


Figure 4-12 The average number of network messages in each superstep

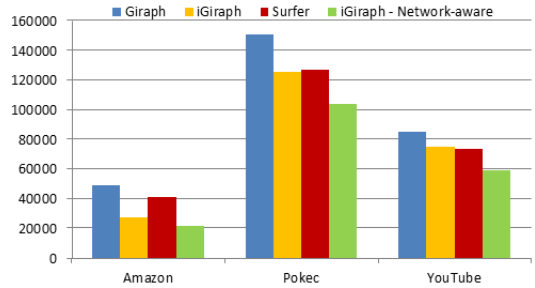


Figure 4-13 Total time taken to perform PageRank algorithm

It is also shown that the new mechanism works well on non-convergent algorithms such as PageRank. According to Figures 4.12 and 4.13, not only the average number of messages in the network is reduced in iGiraph-network-aware compared to Giraph, original iGiraph and Surfer, but also the processing has been completed faster using our bandwidth-and-traffic-aware dynamic repartitioning algorithm. Table 4.2 and 4.3 show the cost comparison for different datasets for shortest path and PageRank algorithms respectively on each framework. Table 4.2 and Table 4.3 show the dollar cost of the operations is much less with the proposed techniques in this chapter.

Table 4-2 Processing cost for SSSP on different frameworks

Dataset	Giraph	Surfer	iGiraph	iGiraph-network-aware
Amazon	$0.0140	$0.0130	$0.0096	$0.0079
YouTube	$0.0125	$0.0120	$0.0082	$0.0067
Pokec	$0.0145	$0.0140	$0.0099	$0.0071

Table 4-3 Processing cost for PageRank on different frameworks

Dataset	Giraph	Surfer	iGiraph	iGiraph-network-aware
Amazon	$0.0250	$0.0210	$0.0140	$0.0110
YouTube	$0.0430	$0.0370	$0.0380	$0.0295
Pokec	$0.0760	$0.0640	$0.0630	$0.0525

The second group of experiments is carried out for computation-intensive scenarios. It is shown that using computation-aware re-partitioning that considers CPU idle time on each worker for mapping, the system performs better compared to Giraph. For this experiment, we have created two 500×500 matrices with random integer numbers and multiply them to keep the CPU busy on a random number of machines. The results of multiplication will not be saved because we do not want to decrease the memory of workers during the experiment. The results of the experiments have only been compared to original Giraph under the same conditions. It means that, for example we have done the experiments on both iGiraph-network-aware with computation-aware dynamic re-partitioning algorithm and Giraph when matrices multiplication is running on six workers and the same workers every time. The results have not been compared with Surfer because it does not have such capability to process the graph using computation information on the network.

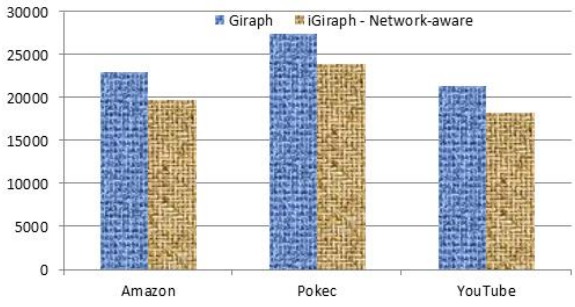


Figure 4-14 Total time taken to perform shortest path algorithm

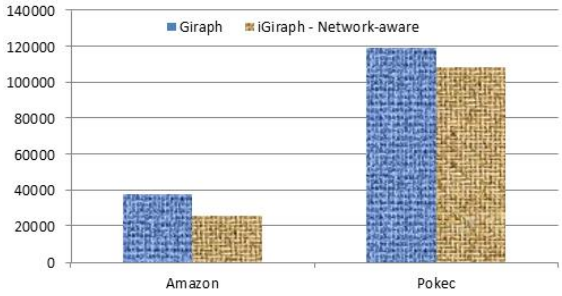


Figure 4-15 Total time taken to perform PageRank algorithm

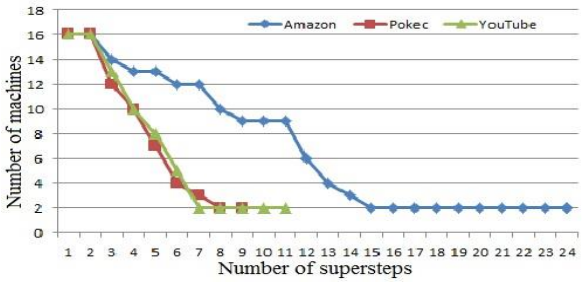


Figure 4-16 Number of machines varying during supesteps while running shortest path algorithms on different datasets

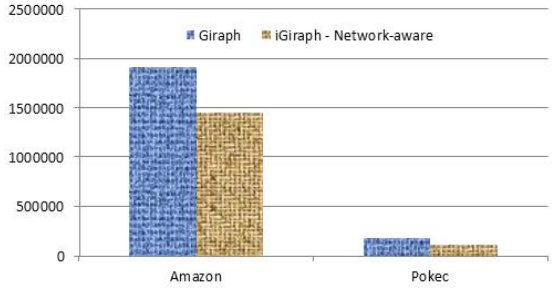


Figure 4-17 The average number of network messages in each experiment

As shown in the Figure 4.16, again the number of machines has noticeably decreased in iGiraph-network-aware using computation-aware dynamic re-partitioning approach for a convergent algorithm such as shortest path algorithm. So, processing the graph on iGiraph-network-aware is much cheaper than doing so on Giraph (Table 4.4). The same results have been obtained for non-convergent algorithm PageRank. It shows that our proposed mechanism has reduced the average number of messages in the network while completing the computation faster. Table 4.4 and Table 4.5 show that the dollar cost of the operations is much less with the proposed techniques in this chapter.

Table 4-4 Processing cost for SSSP on different frameworks

Dataset	Giraph	iGiraph-network-aware
Amazon	$0.0115	$0.0055
YouTube	$0.0110	$0.0042
Pokec	$0.0140	$0.0068

Table 4-5 Processing cost of PageRank on different frameworks

Dataset	Giraph	iGiraph-network-aware
Amazon	$0.0195	$0.0130
Pokec	$0.0600	$0.0545

4.7.3 Discussion

We compared our algorithm with Surfer's algorithm [46], due to its relevance to our work. Both approaches use mapping strategy to map partitions and worker machines for computation. They both consider bandwidth as an important factor that affects the performance of processing which shows the role of network to make the processing costly. Both methods try to reduce the number of cross-partition edges to reduce the number of messages transferred between machines so that they can decrease the communication cost.

Apart from the similarities, there are significant differences between Surfer and our work. First, Surfer partitions the graph before the processing starts and never repartitions data during computation. It creates the partition map at the beginning of the operation along with the workers map, but it only changes the workers map during the processing. The problem is that after each iteration, a new map is generated for workers and partitions have to be moved to a different worker every time. It is specifically very costly when all *active* and *inactive* vertices are meant to be transferred together. This is the reason that iGiraph-network-aware distinguishes between convergent and non-convergent algorithms and is using a re-partitioning algorithm to make a new partition map and workers map after each superstep. Second, the Surfer authors evaluate their approach using METIS and ParMETIS to initiate the partitioning the graph while iGiraph-network-aware uses a random approach. METIS and ParMETIS have been shown to give better partitioning results than random partitioning. So, we believe that this is the reason that Surfer's approach does not work well by being initiated with random partitioning. However, initiating iGiraph-network-aware by either METIS or ParMETIS will still give better results compare to Surfer because of different strategies that they are using. Third, all experiments on Surfer have been done on random graph datasets which is generated by a graph generator and not real-world datasets. Therefore, the impact of high-degree vertices has not been investigated by Surfer, although it is an important feature of real-world

graphs. Fourth, Surfer has not investigated monetary cost of the processing. This is the unique feature of iGiraph-network-aware as it reduces the number of using machines as the operation progresses whereas both Surfer and Giraph maintain the same higher number of machines during the entire operation.

Overall, there are many factors that need to be considered for scheduling resources in cloud environments [28]. However, factors such as monetary cost and networks aspects of clouds have not been investigated much in graph processing context. Our work is one of the first works that combines all those factors to not only improve the performance but also to minimize the cost of using public clouds.

4.8 Summary

As the amount of data is growing every day, processing and analyzing them in a cost-efficient way is a challenge. Distributed graph processing frameworks have emerged in the past few years to facilitate the processing of large-scale graphs that are made and stored by applications such as social networks and mobile applications. On the other hand, cloud computing has brought new facilities to streamline large-scale computing and storage. It has brought different models of computing with new paradigms such as pay-as-you-go model, scalability and elasticity. In this chapter, a new graph processing framework was proposed to analyze large-scale graph data. To achieve this, a new two-dimension classification of graph applications was used for the processing strategy. A novel dynamic re-partitioning was also introduced which considers network factors such as bandwidth and network traffic to process the graph by reducing network, communication and monetary costs. According to our experiments, this model could significantly outperform other frameworks such as famous Giraph.

Chapter 5
Auto-scaling Algorithm for Graph Processing with Heterogeneous Resources

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 on commodity clusters where they will not be charged for the resources that they consume. In this chapter, we have developed algorithms to take advantage of resource

---

This chapter is partially derived from:

- **Safiollah Heidari** and Rajkumar Buyya, "A cost-efficient Auto-scaling Algorithm for Large-scale graph processing in Cloud Environments with Heterogeneous Resources", IEEE Transactions on Software Engineering (TSE), 2018 (Under review)

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 dollar 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% of the dollar cost compared to Giraph.

5.1 Introduction

GRAPH-LIKE 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). Many other areas and applications such as healthcare, search engines, maps, mobile computing and machine learning are also generating and using large-scale graph data. Traditional data processing approaches such as MapReduce [56] has been well-considered for processing large-scale graphs in spite of its initial purpose of operating on tuple-based databases, due to its comprehensiveness in big data processing. Nevertheless, MapReduce is not suitable for the inherent iterative characteristic of graph algorithms. This has led to the development of many specialized graph processing systems such as Pregel [148], GraphLab [142] and others [192] [241]. These systems can inherently represent and perform efficiently on iterative graph algorithms including PageRank [173], shortest path and connected components [96] by developing optimizations to specialized graph abstractions. As a result, graph processing systems perform significantly better than multi-purpose data flow systems such as MapReduce [2]. Recently graph processing application models are also adopted even for stream data processing applications [25].

While it has been shown that graph processing frameworks offer a good level of scalability on fast interconnected high-performance computing machines, their behavior on "virtualized commodity hardware" which is available to a broader range of users is less studied [187]. Cloud computing brought on-demand and scalable distributed storage and processing services by which it overcomes challenges and

restrictions of traditional computing. Meanwhile, public cloud has become more popular by offering services such as infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS) and software-as-a-service (SaaS). It provides cost scalability by facilitating on-demand compute resource provisioning based on its *pay-as-you-go model* where resource access is democratized. However, issues such as the overhead for virtualizing infrastructure on a commodity cluster, performing in controlled situations and environments, lack of complete control on communication bandwidth and latency due to imperfect virtual machine (VM) placement affect the advantages of using such systems. Additionally, performance consistency will be affected by multi-tenancy. On one side, in some cases, 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.

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 [187] and Surfer [46] 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. For instance, Surfer has offered a latency-based graph partitioning approach by which partitions will be placed on workers based on their bandwidth while the .Net-based version of Pregel, Pregel.Net, has evaluated the influence of Bulk Synchronous Parallel (BSP) model [231] on processing graphs using Microsoft Azure public cloud.

Cloud providers usually provide a wide range of resources including various types of virtual machines so customers can choose between resources and find the best option to fulfill their requirements with different priorities. In fact, the adoption of *heterogeneous computing resources* (VMs with different configurations) by cloud users will permit for promoting the efficiency of resources and hence reducing energy usage

and costs. *At the moment, none of the existing graph processing frameworks are taking advantage of this feature.*

Scalability is another important feature that can help cloud applications to gain optimal performance and minimize the cost. iGiraph is a Pregel-like graph processing framework which is developed based on Apache Giraph [11] and deploys a scalable processing approach on clouds. iGiraph proposes a dynamic repartitioning method to decrease the number of VMs during the operation (scalability) by using network message traffic pattern to merge or move partitions across workers. It also executes faster by reducing network traffic based on its mapping strategy. Therefore, iGiraph reduces the cost of processing on public clouds. What distinguishes iGiraph from its other counterparts such as Giraph is that not only it proposes methods for faster execution and provides better performance, but also offers approaches for the less investigated side of such frameworks on public clouds which is the dollar cost of resource utilization. However, iGiraph works only with homogeneous resources instead of heterogeneous VMs. Since the cost model for cloud service providers is based on pay-as-you-go approach, it is very important for customers to choose suitable services by considering the factors that affect the cost of various services.

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 system is to provide the best combination of resources (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 have used iGiraph to propose an auto-scaling algorithm for optimizing the cost of processing on public clouds. Our approach significantly reduces the financial cost of utilizing cloud resources compared to other popular graph processing frameworks such as Giraph [11] 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 this chapter is organized as follows: Section 5.2 describes graph applications and the proposed auto-scaling method that we use in this chapter. Section 5.3 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 5.4 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 provided in Section 5.5, while related works are studied in Section 5.6. Finally, we conclude the chapter and vision our future work in Section 5.7.

5.2 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.

5.2.1 Applications

According to iGiraph (Chapter 3), 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 [173], 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. 5.1 - c). Basically, PageRank

calculates the importance of a web page by counting the number of connections (links) from other pages to it. So, more connections mean higher rank for the page. For a specific web page P, the value will be measured by receiving the values of all the neighboring pages during a processing iteration. This generates almost the same number of messages every time and will continue until all the pages update their values. The graph never shrinks because vertices are actively sending and receiving messages. Therefore, a graph processing framework uses a constant number of machines to process a graph using PageRank for the entire operation.

On the other side, while processing a graph by a convergent algorithm, the number of messages that are being generated in the network will start decreasing at some point during the operation until the end of processing (Fig. 5.1 – a,b). This is because as the more iteration is completing, the more vertices become deactivated (processed) and do not need to exchange messages with their neighbors anymore. As a result, once in a while within an operation using convergent algorithms, deactivated vertices (processed vertices) can be kept outside the memory which means less number (or smaller type) of resources will be sufficient for the rest of the processing (remaining vertices). This continues until there are no vertices to be processed or the desirable result has been achieved. According to this, the processing can be dynamically scalable. Two important algorithms in this category are single source shortest path [190] and connected components [96] that have been used in many studies.

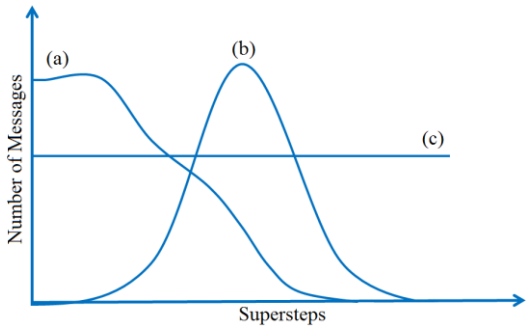


Figure 5-1 General patterns of the number of messages passing through the network during a typical processing show convergence by the end of the operation for (a) CC and (b) SSSP, but (c) PageRank is not converged

Single source shortest path (SSSP): single source shortest path is derived from 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 processed vertices during the operation means, in most cases, 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 specific 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 chapter as they are more suitable for scaling scenarios and will show how they can benefit from the heterogeneity of public cloud's resources using our proposed auto-scaling and repartitioning algorithms and framework.

5.2.2 Proposed Auto-Scaling System Architecture

A user can have access to different types of VMs on a public cloud according to his/her requirements. If the application is communication-intensive, VMs with larger memories and more bandwidth can be utilized and if the application is computation-

intensive, VMs with more CPU capacity will be more helpful to use. Most distributed graph processing frameworks rely only on homogenous implementation and try to reduce the cost by speeding up the computation and decreasing the execution time [80] [223] [32]. These frameworks consider dedicated clusters in various sizes whereas in real world it is not possible for all users to provide such infrastructures. Instead, from a user point of view, it is beneficial to use public clouds for large-scale graph processing [35]. However, monetary cost is a very important issue in choosing the right service from a provider. Issues such as what is the best scaling policy (horizontally or vertically) to reduce the cost?, what is the best partitioning method to take advantage of more cost-efficient VMs?, how these policies can be applied to a graph processing framework?, how to improve the system performance on public cloud?, etc. are very important problems that influence the final performance and cost of the processing. To enhance the performance of large-scale graph processing on public clouds, first, we implement an auto-scaling approach within our graph processing framework to utilize the heterogeneity of resources in this environment.

As shown in Figure 5.2, 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 the network 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 that has 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 and passes them 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 untouched. This module also determines the number of VMs that can be replaced and the types of new VMs based on the information from monitoring module. Eventually, partition distributor module will be notified of the new configuration and distributes new partitions accordingly.

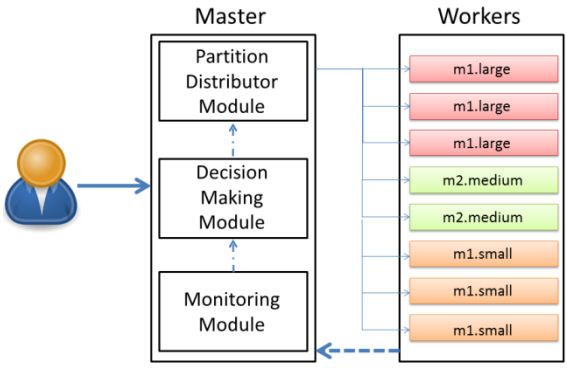


Figure 5-2 Proposed auto-scaling architecture

5.3 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 adding 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 added 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.

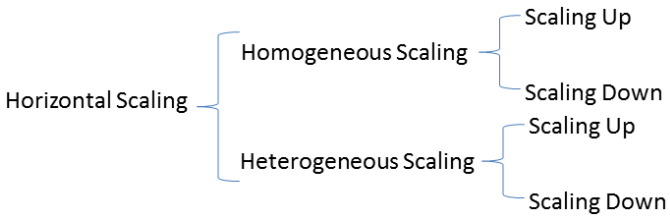
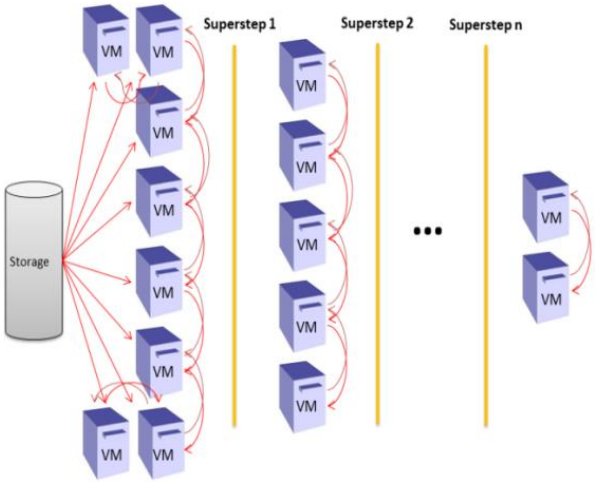
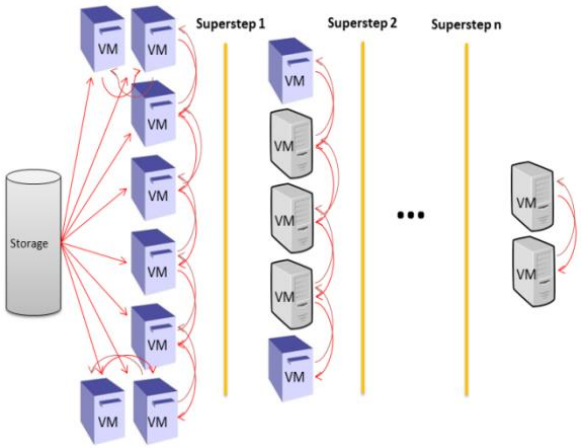


Figure 5-3 General horizontal scaling policies

iGiraph is an extension of Giraph [11] and the only Pregel-like framework that scales down homogeneously across public clouds while processing convergent algorithms (Fig. 54 - a). Basic iGiraph does not monitor network factors (except network traffic) or VM availability. Its decisions are made based only 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 only using one type of machines (e.g. large machines) during the entire operation.



(a) iGiraph homogeneous scaling policy



(b) Our proposed heterogeneous scaling policy

Figure 5-4Scaling 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

The alternative to make this method even more efficient is to use 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 larger than what is needed to complete the operation and it means that the user is paying for a big 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 and process the rest of the graph, and partitions it accordingly. The new framework can easily be used as a cost-efficient graph processing service. Fig. 5.4 compares the original iGiraph homogeneous scaling policy versus our proposed policy.

iGiraph-heterogeneity-aware, unlike basic iGiraph, 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, available VMs, etc. provides the new approach with a holistic view of the entire system and the environment that it is operating in it which in turn is required to estimate and optimize the cost of

processing. Since every processing operation consists of several iterations (supersteps), 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.

$$C(S_{i+1}) \leq C(S_i) \qquad (5.1)$$

According to Formula 5.1, 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 5.2.2). This approach successfully deals with price heterogeneity of the cloud resources too as price is one of the variables in the equation. This ensures that not only the smaller VMs are being used, but the monetary cost is being considered as well.

5.4 Dynamic Characteristic-Based Repartitioning

In this section, we discuss the smart VM monitoring and our proposed characteristic-based dynamic repartitioning approach.

5.4.1 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 lead communication messages to the channels with less traffic to reduce latency, or monitoring available memory and price of VMs enables to choose 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 which is usually dependent on the application itself. For example, if the application is communication-bound, 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 Chapter 4. In this chapter, 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 Chapter 3, 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 [79]. 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 Ganglia20 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 is 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 (Chapter 4). 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% 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

[20] http://ganglia.sourceforge.net/

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.

5.4.2 Dynamic Repartitioning

Algorithm 5.1: Characteristic-based Dynamic Re-partitioning

```
1:  Get the information about available VMs in the network
2:  Partition the graph randomly
3:  Set PP=0 for each partition and WP=0 for each worker
4:  For the rest of the computation do
5:      Calculate PP for each partition based on the number of messages that each
        partition receives
6:      Calculate WP for each worker using end-to-end mechanism
7:      If global synchronization happened then
8:          If Size(Partition P1)<=Size(MemoryOfSmallVM) and
              Size(Partition P2)<=Size(MemoryOfSmallVM) and Size(Partition
9:            P1+Partition P2)<=Size(MemoryOfSmallVM) then
10:           mergeIntoSmallVM(P1,P2)
11:           removeCurrentVM(P1,P2)
            If Size(P1+P2)>Size(MemoryOfSmallVM) and
12:           Size(P1+P2)<=Size(MemoryOfCurrentVM) then
13:           mergeIntoCurrentVM(P1,P2)
14:           removeCurrentVM( )
            If Size(Partition P1) is very small and there is enough space
15:           in its adjacent partitions then
16:           migrateIntoAdjacent(P1)
17: Merge the partitions or migrate vertices if needed
18: Set the priorities based on PP and WP
19: Add/Remove VMs if needed
20: Map partitions(based on PP) and workers(based on WP)
21: If VoteToHalt() then
      Break
```

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 5.4.1, 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 chapter, 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 chapter, at the start of the processing, the graph will be partitioned randomly (based on vertices identifiers). It has been shown that other well-managed partitioning methods such as METIS have better performance compared to the cheap random partitioning (in terms of time and cost). Therefore, if we are able to improve the system by using random approach, it will work even better when the processing starts with other well-managed methods in many cases. 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 5.4.1) before the next superstep to use them for repartitioning purpose.

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, like 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 (PP=0), the PP for the partition that has received the second largest number of messages through the network will be set to 1 (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 chapter (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 (WP=0), WP for the VM with the second highest bandwidth will be set to 1 (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.

As mentioned above, the system selects VMs of the same type from the resource pool based on the user's requirement. For example, if partition P1 has higher priority than partition P2, after merging these two partitions (P3=P1+P2), P3's priority will be set to the former priority of P1. Meanwhile, the priorities of the new partitions that are formed by merging other partitions will be set to the highest priority among existing partitions. Also in cases a big 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. The priorities of all partitions and VMs are set to 0 at the beginning of the operation (before superstep 0 starts).

As mentioned, the system starts with the number of VMs that is determined by the user. These VMs are part of available VMs in the network (resource pool). The operation also starts with the VMs of the same type. For example, if the user set the number of VMs to 16, and choose 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 (P1+P2) will be moved to the new small VM. If (P1+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. So, for example, if the CurrentVM is "Large" type, then SmallVM can be a "Medium" type and so on. As a result, this algorithm is working for any types of VMs.

5.5 Performance Evaluation

5.5.1 Experimental Setup

To evaluate our framework and effectiveness of the proposed algorithms, we utilized resources from Australian national cloud infrastructure (NECTAR) [163]. 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 utilized

VMs are shown in Table 5.1. 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 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 VMs from, our solution always reduces the monetary cost by the order of magnitude compared to other existing frameworks. We use iGiraph 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, and Pokec and Twitter [125] as shown in Table 3.1.

Table 5-1 VM characteristics

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

5.5.2 Evaluation and 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 [49]. We also compared the performance of basic iGiraph that scales out homogeneously with our proposed heterogeneous extension of iGiraph (iGiraph-Heterogeneity-aware). The size of the messages in all experiments is equal, hence the total cost of communication is independent of message size. Instead, the total number of messages that are transferring through the network is calculated for cost. In addition, one medium VM is almost equal to two small VMs in terms of capacity and the power, or we can say each small VM is equal to 0.5 medium VM. We use this when we are calculating the number of machines that are being used by the system at any moment. For example, 1.5 means that there is one medium VM (1) and one small VM (0.5) being used, or 1 can mean either one medium VM or two small VMs (0.5+0.5=1), etc. All experiments start with medium VMs as their workers.

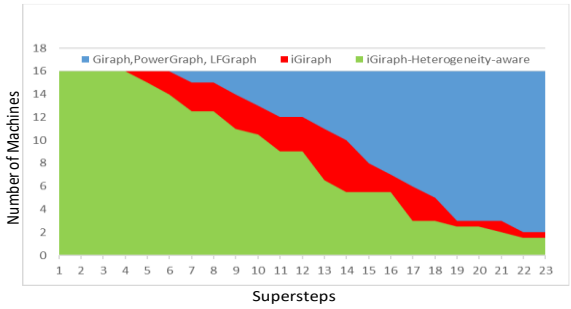


Figure 5-5 Number of machines during processing shortest path on Amazon

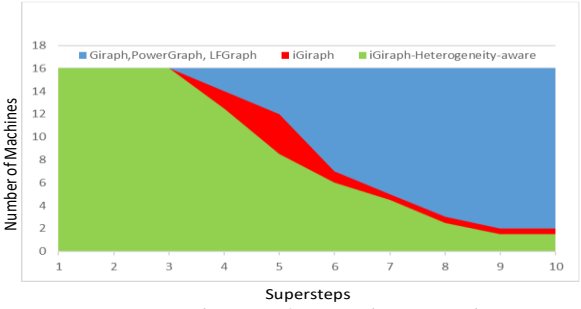


Figure 5-6 Number of machines during processing shortest path on YouTube

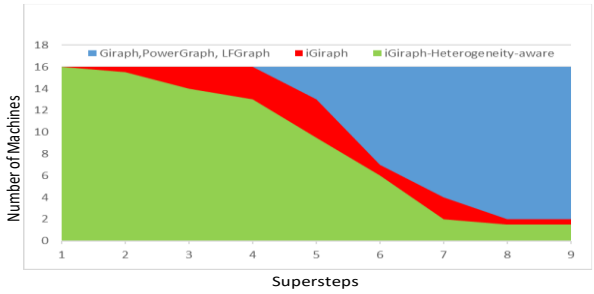


Figure 5-7 Number of machines during processing shortest path on Pokec

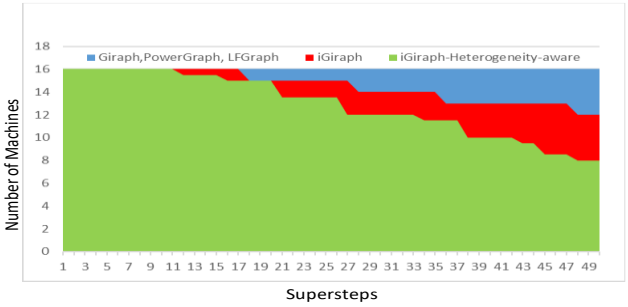


Figure 5-8 Number of machines during processing shortest path on Twitter for the first 50 supersteps

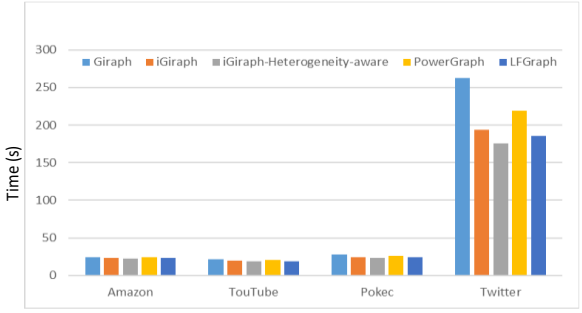


Figure 5-9 Total execution time for processing connected components algorithm on various datasets

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 wich is correlated with the cost of the operation. So, Giraph is the most costly solution among the three 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 and more vertices become processed, 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 the Giraph. However, basic iGiraph only removes the machines homogeneously. It means that if the processing is started by medium machines, it will end by medium machines as well despite the VM reduction. In this case, although smaller partitions tend to be merged to create a bigger partition that is suitable for the current using VMs to optimize VM utilization, there are always situations where a tiny partition cannot be merged or migrated but occupies a big VM and all its capacity. To address this, iGiraph-Heterogeneity-aware replaces current VMs by smaller one. It has been shown that iGiraph-Heterogeneity-aware provides more than 20% 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 VMs as a package of resources including computation resources, storage resources, etc. Hence, removing or downsizing VMs lead to huge cost savings. All VM types are correlated with particular prices as shown in Table 5.1. Therefore, as it can be seen in the diagrams (Fig. 5.5, 5.7, 5.9) for both basic iGiraph and 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 5.3). However, iGiraph-heterogeneity-aware achieves better results by taking advantage of resource heterogeneity. As shown in Figures 5.6, 5.8 and 5.10, iGiraph-Heterogeneity-aware even completes the processing faster than other two frameworks due to its new partitioning approach which distributes partitions based on their characteristics and the properties of available machines. Figures 5.11-5.13 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.

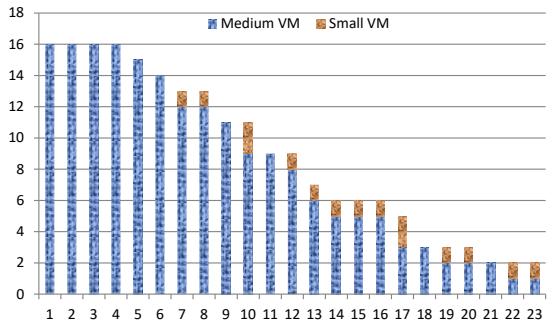


Figure 5-10 Resource modification during processing shortest path on Amazon

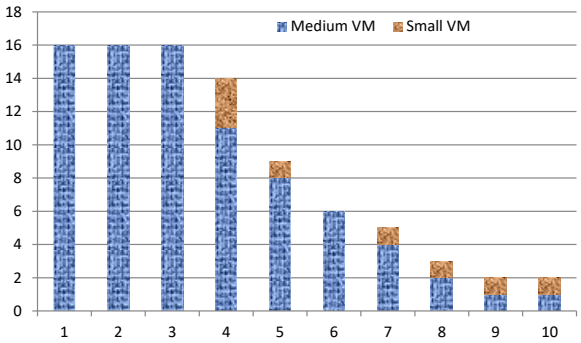


Figure 5-11 Resource modification during processing shortest path on YouTube

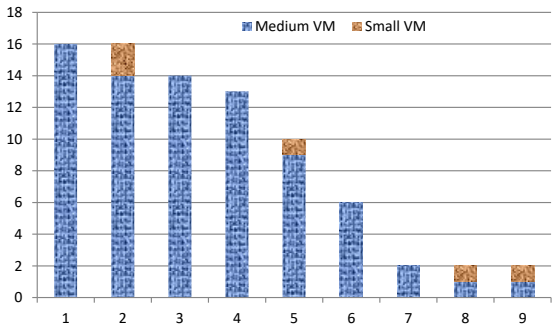


Figure 5-12 Resource modification during processing shortest path on Pokec

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 Formula 5.2, where $C(VM_i)$ is the price of the VM and $T(VM_i)$ is the time that

within the VM is used. The equation calculates the cost for all VMs (n) during the entire processing iterations (m).

$$\text{Cost}_{final} = \sum_{j=0}^{m}\sum_{i=1}^{n}(C(VM_i)\times T(VM_i)) \quad (5.2)$$

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 5.2 shows the cost comparison for different datasets for shortest path algorithm on each framework

Table 5-2 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

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 (Fig. 5.14-5.17). Also, our proposed partitioning method for iGiraph-Heterogeneity-aware makes it outperform basic iGiraph up to 20%. Table 5.3 shows the cost comparison for different datasets for connected components algorithm on each framework.

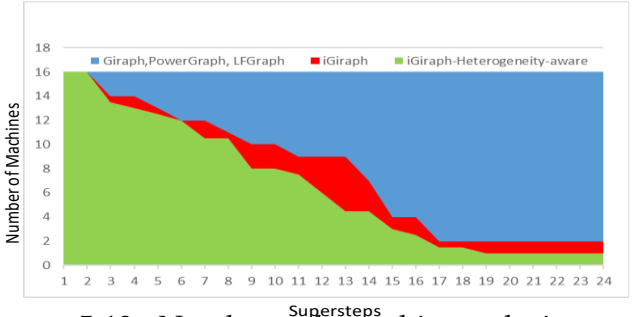


Figure 5-13 Number of machines during processing connected components on Amazon

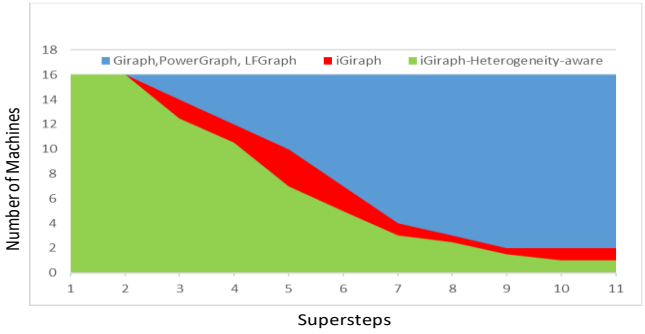


Figure 5-14 Number of machines during processing connected components on YouTube

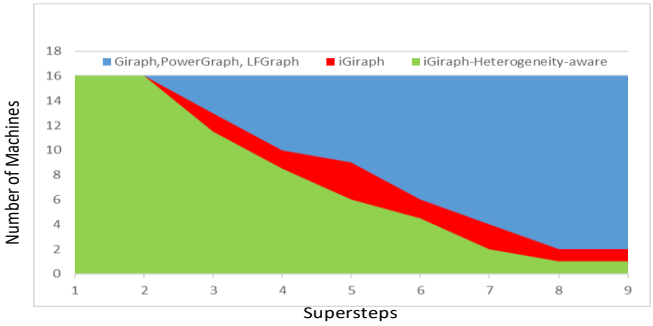


Figure 5-15 Number of machines during processing connected components on Pokec

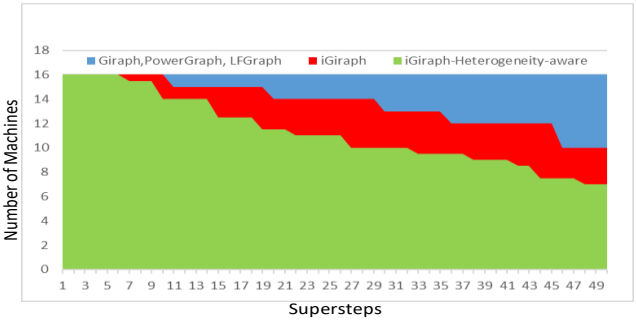


Figure 5-16 Number of machines during processing connected components on Twitter

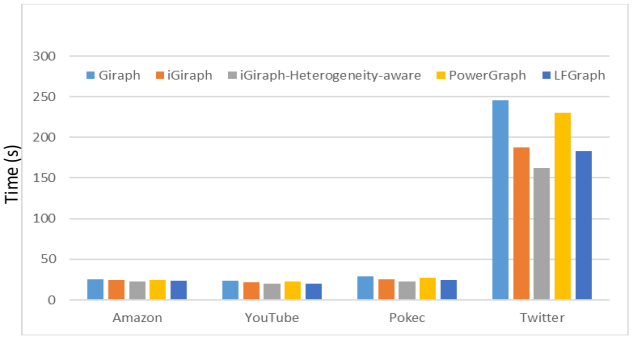


Figure 5-17 Total execution time for processing connected components algorithm on various datasets

Table 5-3 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

It should be noted that there is always overheads while migrating or merging partitions and vertices across the system. However, the overhead is not very large that can affect the total performance of the system. That is because migrating and merging usually lead to removing a VM from the list of active resources or replacing that with a smaller type. Therefore, this trade-off cannot influence the performance significantly. Table 5.4 demonstrates different characteristics of the three systems and the newly implemented features.

Table 5-4 Comparison of scheduling and resource provisioning algorithms

Property	Giraph	iGiraph	iGiraph-heterogeneity-aware
Dynamic repartitioning	×	√	√
Traffic-aware	×	√	√
Bandwidth, CPU, environment-aware	×	×	√
Heterogeneity-aware	×	×	√

5.6 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 [11], GPS [195] and GiraphX [220] along with some non-Pregel-like frameworks such as Trinity [202], Presto [233] and PowerSwitch [240] argue that a distributed architecture can provide better scalability. The system can access as many resources as needed to operate and increase the performance. However, these systems use other optimizations to deliver better performance as well. GPS [195], 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 like PowerSwitch [240], improves the performance of the system by effectively predicting the proper heuristic for each step and switching between synchronous and asynchronous execution states if required. iGiraph-heterogeneity-aware provides not only scalability over heterogeneous resources, but also is elastic as it provisions in an autonomic way such that at any given

time the current demand and resource consumption matches, according to [95].

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. The pay-as-you-go model provided by cloud service providers is defined based on this fact. In such environment, it is important to reduce the cost of utilizing resources as much as possible. Many graph frameworks tried to decrease the cost by executing faster to reduce the total runtime so that they can release the resources quicker to pay less. For example, Surfer [46] 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, which we used in this chapter 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 [195] and Mizan [118] implement dynamic repartitioning and vertex migration, they do not scale across VMs. It has also been shown that iGiraph outperforms frameworks such as popular Giraph while operating on non-convergent algorithms like 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 [91] [165] [204]. Since system memories and disks have unprecedentedly become large and available on single PCs, these frameworks implement mechanisms for processing large-scale graphs without the hassle of distributed computing. 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 [127] is one of the firsts in this category. It is a vertex-centric framework that offers a parallel sliding window (PSW) which is an asynchronous computing method to leverage external memory (disk) and is suitable for sparse graphs. Using PSW, GraphChi requires transmitting only a small number of

disk-blocks sequentially. PSW's input is a subset of the graph that is being 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 [262] are specifically designed to perform on SSDs. In FlashGraph, I/O requests will be merged cautiously to improve the throughput and decrease CPU overhead. Despite various optimizations and improvements in single-machine-based graph processing frameworks, they are not still efficient compared to their distributed counterparts in more sophisticated scenarios such as when there are multiple datasets that need to be processed at the same time.

Finally, dynamic partitioning and network factors are the last two aspects of our work in this chapter. 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 [195], 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 [195] repartitions the graph based on using high-degree vertices while LogGP [244] does so based on analyzing and reusing "the historical statistical information" to rectify the partitioning outcomes. Other systems such as Mizan [118], XPregel [222] and xDGP [232] 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[21] has investigated the impact of fast networks on graph analytics after an NSDI paper [172] 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. Chapter 3 has used network factors such as bandwidth, traffic, and CPU utilization to partition the graph dynamically. It has shown that using a suitable combination of factors will make the system to

[21]http://www.frankmcsherry.org/pagerank/distributed/performance/2015/07/08/pagerank.html

outperform frameworks such as Giraph. A detailed comparison of many existing graph processing frameworks has been discussed in Chapter 2.

5.7 Summary

The amount of data that is being made and stored every day in the form of graph is dramatically increasing. Social networks popularity, IoT birth, and mobile applications improvements are among the many reasons for this growth. To address the challenges in this area, where traditional data processing solutions could not be helpful, graph processing frameworks have been developed in the past few years. On the other side, cloud computing provides solutions to facilitate large-scale computing and storage. It also brings various features such as pay-as-you-go model, elasticity, and scalability to the computing. However, cloud-based graph processing and how these features can be utilized to streamline operating on large-scale graphs is less investigated. In this chapter, 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 is the first implementation of a graph processing framework that horizontally scales up using heterogeneous resources in a cloud environment.

Chapter 6
Graph Processing-as-a-Service

Large-scale graph data is being generated every day through applications and services such as social networks, Internet of Things (IoT), mobile applications, etc. To overcome challenges and shortcomings of traditional processing approaches such as MapReduce, 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 chapter, for the first time, we introduce the large-scale graph processing-as-a-service (GPaaS). The algorithm that we introduce for this service consider 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 and also reduces the execution time compared to other graph processing frameworks such as popular Giraph. We show that our service significantly improves the performance compared to Giraph or other frameworks such as PowerGraph.

This chapter is partially derived from:

- **Safiollah Heidari** and Rajkumar Buyya, "Quality of service (QoS)-driven Resource Provisioning for Large-scale Graph Processing in Cloud Computing Environments: Graph Processing-as-a-Service (GPaaS)", Future Generation Computer Systems (FGCS), 2018 (Second Review: Minor Revision)

6.1 Introduction

TODAY data is an asset and being able to collect, analyze, protect and use 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 [6], during each minute at 2017, 3.3 million posts were put on Facebook, 3.8 million queries were searched on Google search engine, 500 hours 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 of exclusive large-scale graph processing frameworks. Pregel [148] 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. 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 a 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 has more complications as mentioned. 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 [174]. 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) [175] 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.

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 (Chapter 3) are developed specifically to take advantage of cloud computing features, there is no mechanism to certify the quality of service that is provided by these systems on cloud. Current frameworks typically receive a 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. *Therefore, in this chapter 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. 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 built-in 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), 3) discovering various patterns in graphs and extracting knowledge using pattern matching algorithms, and so on.

The rest of the chapter is organized as follow: 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 finally, Section 6 concludes the chapter.

6.2 Related Work

Since 2010, when Google introduced its graph processing framework called Pregel [148], many research works have been conducted to exclusively improve processing of graph data structures. Some graph processing systems such as GraphChi [127], TurboGraph [91], X-Stream [192] and Grace [236] were developed to enable operating 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 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 [118], PowerGraph [80], GiraphX [220], Trinity [202], 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. Many of these challenges have been investigated in various research works and different solutions have been proposed to address them.

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 [35]. 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 [46] and Pregel.Net [187] 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 Prgel.Net is evaluating the impact of Bulk Synchronous Parallel (BSP) model [231] 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 (Chapter 3) 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[22]. iGiraph is also employing BSP model while it is implemented on top of Hadoop[23] 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.

Scalability and monetary costs have been investigated separately in few other research works. For example, Pundir et al. [185] have developed a dynamic repartitioning technique based on LFGraph framework [98] 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. [136] 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 [63], where authors are improving the performance of the system by distributing the computation among GPUs to boost the computation speed while others such as [246] are evaluating the performance of single-node frameworks on cloud environments. Table 6.1 shows the comparison of the most related works.

[22] https://giraph.apache.org/
[23] https://hadoop.apache.org/

Table 6-1 Comparison of the most related works in the literature

System	Architecture	Implemented Environment	Partitioning Method	Resource-aware	Scalability	QoS-aware
Pregel	Distributed	HPC	Static	No	No	No
Giraph	Distributed	HPC	Static	No	No	No
PowerGraph	Distributed	HPC	Static	No	No	No
GPS	Distributed	HPC	Dynamic	No	No	No
Pregel.Net	Distributed	Cloud	Dynamic	No	No	No
Surfer	Distributed	Cloud	Dynamic	No	No	No
iGiraph	Distributed	Cloud	Dynamic	Yes	Only Scale-in	No
Our work - GPaaS	Distributed	Cloud	Dynamic	Yes	Scale-in/out	Yes

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 [242]. 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. [18], "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 [149] [157] [130]. However, in order to addressing QoS challenges in the context of large-scale graph processing, every solution needs to meet specific requirements due to the inherent characteristics of highly connected graph data. in this chapter, we are providing a graph processing as a service framework based on our latest version of iGiraph that discussed in Chapter 3. 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.

6.3 Overview of the Proposed Solution

Figure 6.1 and 6.2 show the workflow and architecture of our proposed solution, respectively. As can be seen in the picture, the system contains seven different functionality modules that are depicted by seven different colors. These modules include: 1) Users, 2) Repositories, 3) Priority queue, 4) Monitoring, 5) Management, 6) Partitioning, and 7) Computation. Each functionality 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 zone and its components in detail.

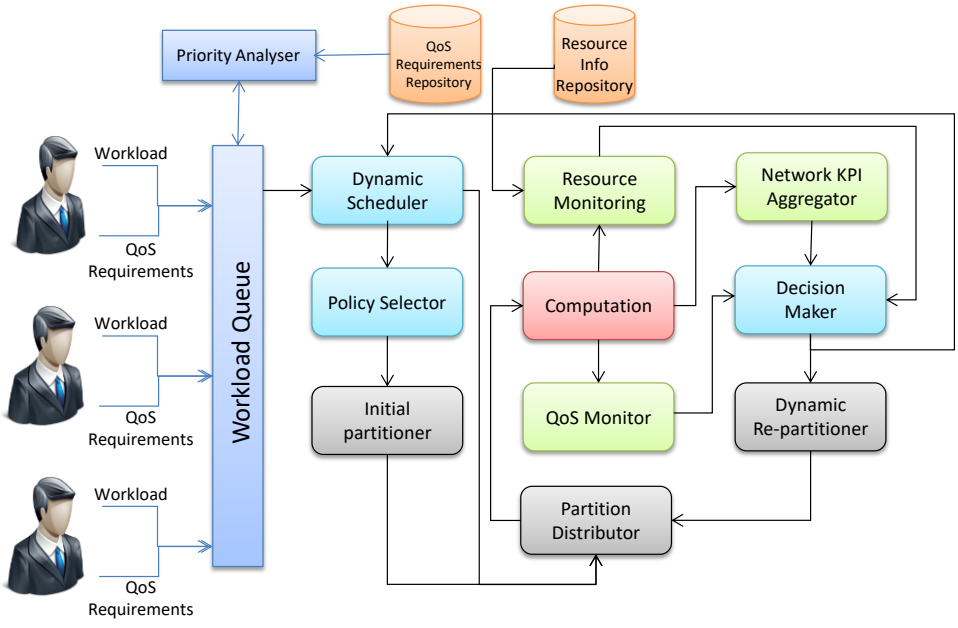


Figure 6-1 The workflow of our proposed solution (GPaaS)

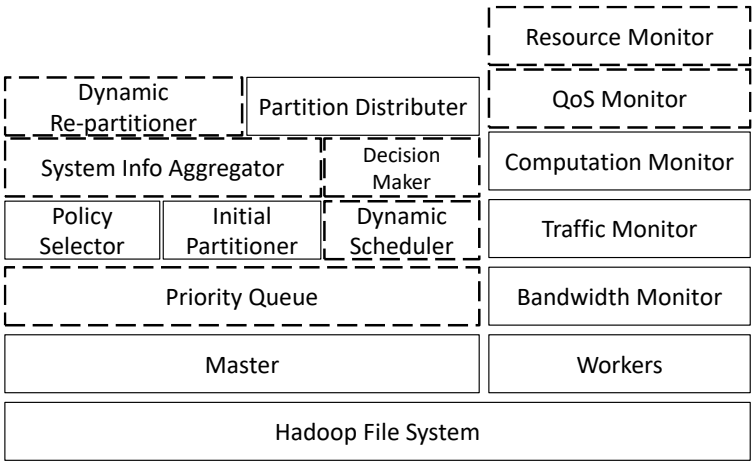


Figure 6-2 The components that we added to our work in Chapter 5 are shown in dotted rectangles

6.3.1 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 chapter, we discuss two 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.

6.3.2 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 chapter, 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.

6.3.3 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.

Algorithm 6.1: 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
```

6.3.4 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 KPI Aggregator*: This component monitors network factors such as network traffic, bandwidth, latency, topology, etc. In this chapter, we are using two major factors including traffic and bandwidth in our dynamic repartitioning algorithm. We are using the method that is introduced in Chapter 4. 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 chapter 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.

6.3.5 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 6.2: 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(DesisionMakerVMList)
7:      matchVMWith(NewInfo)
```

- *Policy Selector*: Original iGiraph (Chapter 3) and its extended network-aware version (Chapter 4) provided a general categorization for various processing environments on clouds and different graph algorithms. This is shown in Figure 5.3. 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 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*.

6.3.6 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 in 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 (Chapter 2). 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 chapter 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 [116] 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 (Chapter 5). This allows us to achieve better performance with less monetary cost compared to other frameworks such as popular 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.

6.3.7 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.

6.4 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 first 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.

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 chapter 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* 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 usable 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.

As can be seen 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 chapter 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.

Algorithm 6.3: 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)
```

6.5 Performance Evaluation

6.5.1 Experimental Setup

To evaluate our framework and effectiveness of the proposed algorithms, we utilized resources from Australian national cloud infrastructure (NECTAR) [163]. 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 utilized VMs are shown in Table 5.1. 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 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 (Chapter 3) 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) and PageRank (PR) algorithms as representatives of convergent and non-convergent graph algorithms respectively 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, and Pokec [125] as shown in Table 3.1.

6.5.2 Evaluation 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 [49]. To evaluate different scenarios by our service, we

have provided various workloads and jobs by combining the datasets from Table 2 with different characteristics. Table 3 demonstrates input jobs and the order of inputs along with their properties.

Table 6-2 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

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 Figure 6.3, 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 experiences. In our future research work, 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.

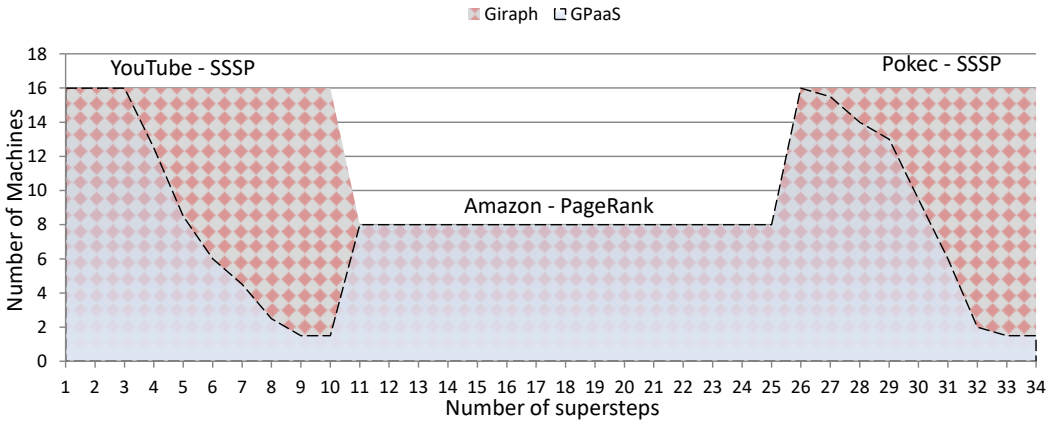


Figure 6-3 Scenario1

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. Figure 6.4 shows how this scenario works and Figure 6.5 demonstrates the scenario in which Giraph follows the job order and depicts what is happening in reality.

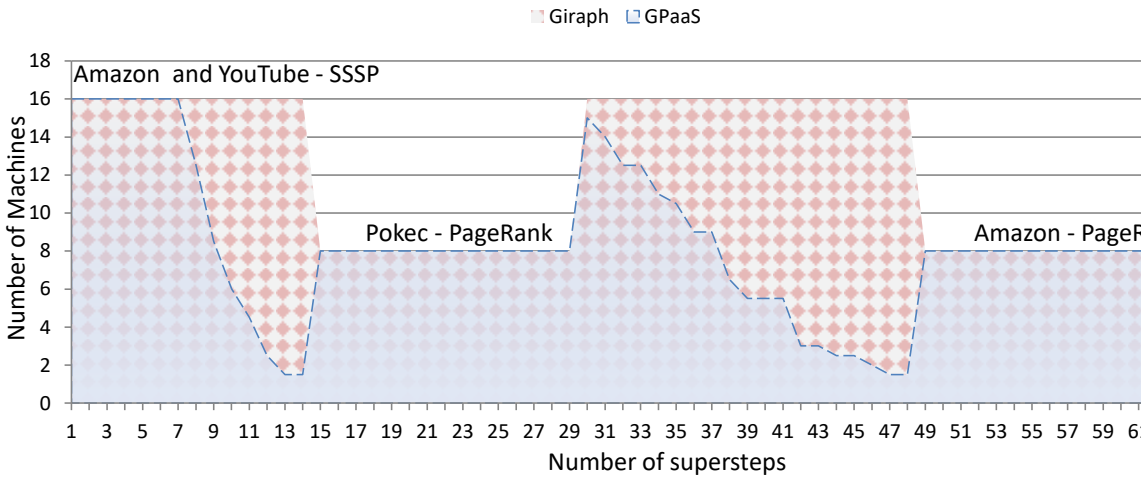


Figure 6-4 Scenario 2 – Price(#VMs) Comparison

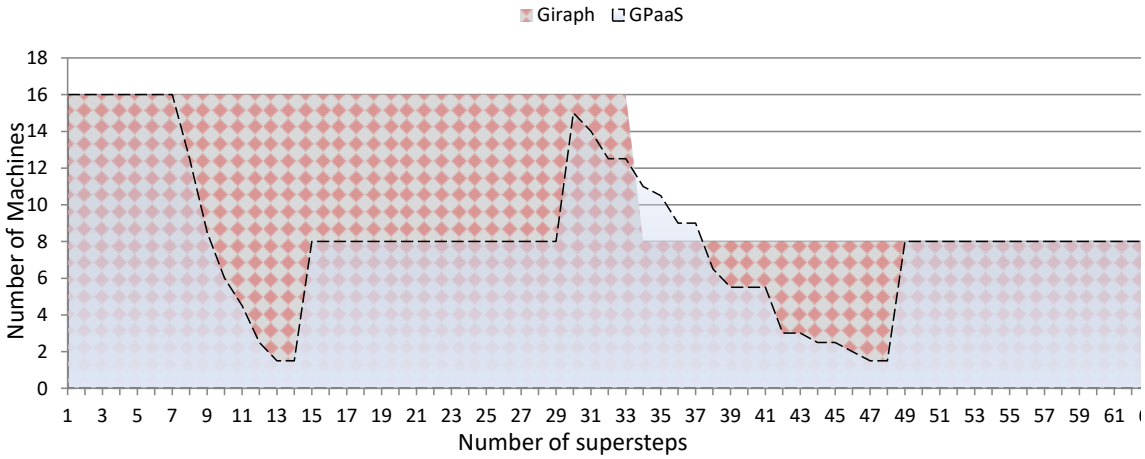


Figure 6-5 Scenario 2 – If Giraph follows the job order

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. Figure 6.6 shows the processing order in this scenario and compares that with Giraph.

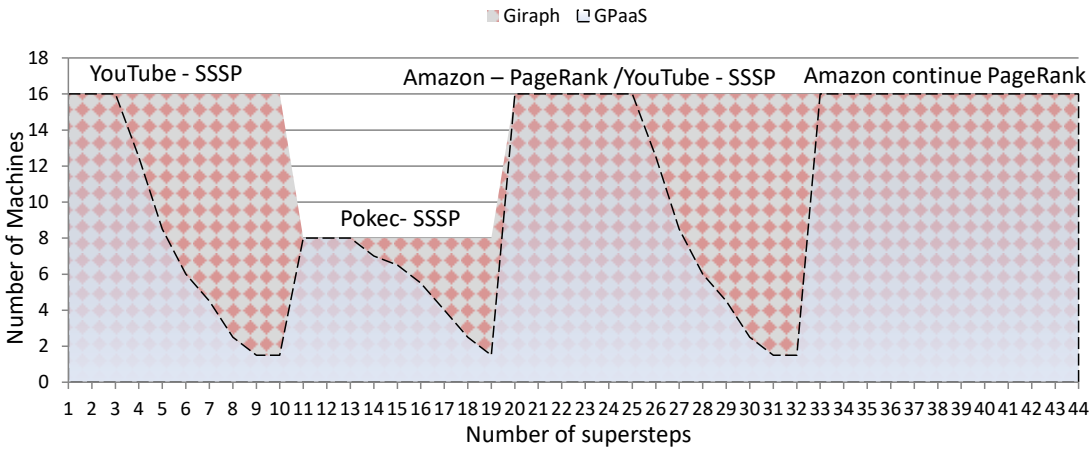


Figure 6-6 Scenario 3

We conducted the same experiments on PowerGraph, 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". However, its processing pattern is the same as Giraph as shown in Figures 6.3 to 6.6 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.

Figure 6.7 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.

Table 6-3 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

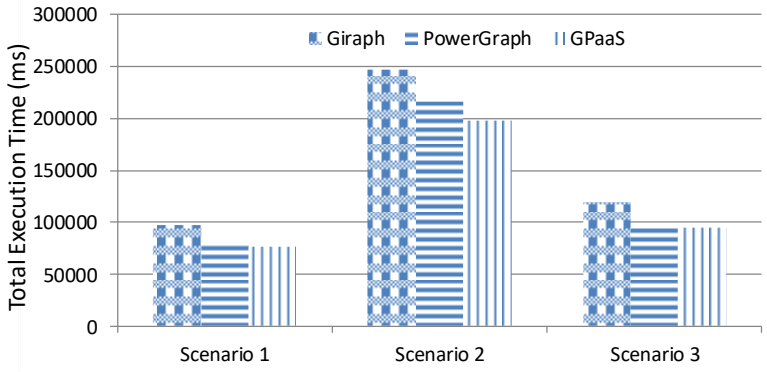


Figure 6-7 Total execution time per scenario

6.6 Sumary

Many applications such as social networks, mobile applications, IoT devices and applications, etc. are generating huge amount of data which a considerable fraction of it is graph data. It has been proven that traditional processing solutions such as MapReduce does not work efficiently with graph data due to their inherent properties. So, many graph processing frameworks are developed to address the challenges of large-scale graph processing. However, many of these frameworks are designed to operate on HPC environments rather than clouds. Since HPC infrastructure is not

available to everyone cloud computing is a suitable candidate for implementing the frameworks on as it can be accessible easier. Cloud computing is offering unprecedented features such as elasticity and pay-as-you-go billing model that can be utilized to improve processing operations' performance even more. Although, few research works developed graph processing frameworks exclusively to be used on cloud environments, they have many limitations and cannot guarantee the quality of services as it is expected in negotiated SLA between cloud provider and clients. In this chapter, we have proposed the first large-scale graph processing service on cloud. 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. Our evaluation results showed that our service can handle graph processing requests successfully to a high extent. We showed 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. This means that customers can save a lot of money and time while the quality of service is being maintained. GPaaS can be used for a wide range of applications such as finding shortest path in GPS systems, recommendation systems, pattern recognition, knowledge extraction, etc.

Chapter 7
Conclusions and Future Directions

This chapter concludes the presented research works' contributions about cost-efficient resource provisioning in distributed graph processing systems in cloud environments. This clearly shows the additive approach of the thesis in developing its proposed solutions for the identified problem in the first chapter. Summarizing the findings also results in identifying promising research directions to be explored In the future.

7.1 Conclusions and Discussion

LARGE-SCALE graph processing has attracted a lot of attention since the appearance of its new exclusive frameworks in 2010. Although the age of big data began many years before 2010, traditional processing solutions such as MapReduce were shown not to be working efficiently on highly connected data structures such as graphs. This was due to the inherent characteristics of graphs and the infrastructures

---

This chapter is partially derived from:

- **Safiollah Heidari** Yogesh Simmhan, Rodrigo N. Calheiros and Rajkumar Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges", ACM Computing Surveys, vol. 51, Issue 3, No. 60, ACM Press, New York, USA, 2018

where operations were conducting on. Meanwhile, cloud computing and its comprehensive features for addressing various processing challenges became very popular. Cloud computing is committed to supply seemingly endless storage space and computing resources via subscription-based services. This enables the users (clients) to scale up and down the amount of resources they are willing to utilize while the pay-as-you-go billing model allows the cloud providers to charge them on a usage basis similar to other utilities namely as electricity, gas and water. However, despite all the benefits that cloud computing provides, most distributed graph processing systems are designed and implemented on high performance computing (HPC) clusters where the complexity of usage billing, elasticity, etc. of cloud environments are not applied.

Unlike HPC environment where several research works have extensively investigated graph processing frameworks on, the impact of implementing such systems in cloud environments and how they can take advantage of various features there have been less studied. In this thesis, we conducted a comprehensive survey to find the gaps in the literature and then proposed several algorithms and techniques to overcome the issues related to cost-efficient resource provisioning and scheduling of graph processing systems in the cloud. We also implemented all the proposed solutions within our extended version of Giraph framework and called it iGiraph. To clarify the challenges in this area, Chapter 1 discusses some research problems and highlights five major issues which are targeted by this thesis in order to provide cost-efficient resource scheduling algorithms – scalability, partitioning, network factors, heterogeneous resources, and quality of service. Then a summary of the thesis objectives demonstrated what is required to tackle these challenges and how they are going to be evaluated throughout the thesis. Afterwards, the contributions of the conducted thesis were presented following the illustration of the thesis organization at the end of the chapter.

Chapter 2 provided a comprehensive literature review and conducted a taxonomy to map the existing research works on that. It started by discussing the inefficiency of traditional data analytic methods such as MapReduce for processing large-scale graphs and depicted the necessity of exclusive processing approaches by providing several

examples in which the system is representing a graph-based environment. It then explains the overall scheme of a typical graph processing system which is a basic for the rest of the chapter where each subsection can be mapped into this scheme. Chapter 2 conducted a detailed investigation on various components of graph processing frameworks and discussed many examples about how these components are being used in existing frameworks in each section. This chapter also analysed the gap in the literature and concludes with a tabular review of several important works.

Chapter 3 introduces iGiraph, our extended version of the popular graph processing framework Giraph, that is a distributed framework designed to operate on cloud environments by which large-scale graph datasets are processed in a cost-efficient way. This chapter is also presenting a new classification of graph algorithms that have not been considered in any graph processing system implementation before. Using this classification that divides algorithms into convergent and non-convergent, the system can adjust the number of required resources for the operation. Another critical part of the system is the dynamic repartitioning mechanism that has utilized the aforementioned classification with taking advantage of high-degree vertices' characteristics in real-world graphs and the network traffic to repartition and distribute nodes across the resources. This resulted in reducing the number of cross-edges in the network as well as the number of messages passing through that. The monetary cost is also declined because of the faster execution and using optimal number of machines.

Chapter 4 emphasises on the importance of network factors in a highly distributed environment as clouds. The graph algorithm classification from Chapter 3 is expanded in this chapter and a second dimension is added to it. Two network-aware dynamic repartitioning-based scheduling algorithms were proposed in this chapter too. The bandwidth-and-traffic-aware algorithm which is suitable for communication-intensive applications measures the bandwidth and the traffic in the network and distributes the partitions accordingly. It is shown that the algorithm can effectively reduce the monetary cost of the operation and since most graph applications fall into this category, the evaluation results seem promising. The computation-aware algorithm works well when the processing speed is important for the user/application. Using

this algorithm, the system always utilizes the machines with less CPU burden to speed up the operation. It is also shown that this approach reduces the dollar cost remarkably.

In Chapter 5, the importance of benefiting from resource heterogeneity in cloud environments and its impact on large-scale graph processing is investigated. This chapter introduces a smart monitoring mechanism to enable the system to obtain information about various metrics during each superstep. Metrics such as network bandwidth, CPU utilization, network traffic, available machines, prices of machines, available memory, partition size, etc. are being measured and monitored to be used for further calculation in our characteristic-based dynamic repartitioning algorithm. According to the measured metrics, our proposed characteristic-based dynamic repartitioning algorithm migrate vertices or merges partitions and distributes them across the provisioned VMs. However, the type and the number of resources in each iteration might change based on the decisions that is made in decision-making module to minimize the dollar cost of the processing. Then, the auto-scaling algorithm will adjust the heterogeneous resources with the actual requirements. This significantly reduces the cost.

Chapter 6 puts all the developed techniques from previous chapters together and combines them with its own new mechanisms to provide the large-scale graph processing as a service in cloud environments. Unlike existing graph processing frameworks that can only handle one dataset, this chapter proposes a multi-handling approach in which different users can submit their jobs (that includes the dataset and SLA requirements) at the same time. Then the prioritization algorithm will place them in the system based on their priorities. A new dynamic scheduling and resource provisioning approach has also employed in the framework that comprehensively watches the changes in the system and makes decisions based on various measured metrics.

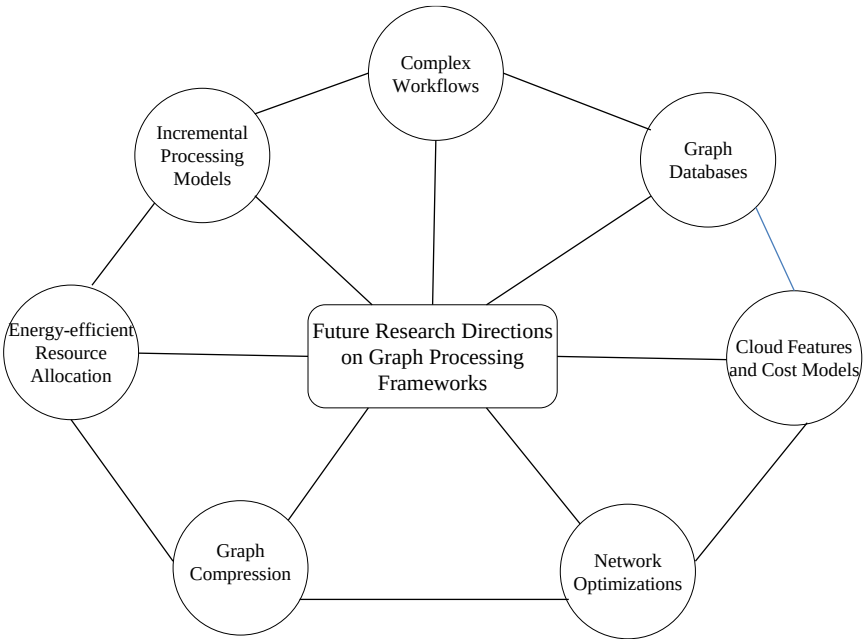


Figure 7-1 Future directions

7.2 Future Directions

Although several works have looked into improving graph processing systems, there are still a number of issues that are open. For example not many graph processing frameworks use dynamic repartitioning which performs better than simple static partitioning in many cases. Most the frameworks use checkpointing for error handling, which can be costly, and other approaches to fault recovery are not well studied. While many researchers have studied classic graph algorithms such as PageRank and shortest paths, it is not clear whether these frameworks can still perform as well for more sophisticated and real-world applications such as machine learning algorithms.

Besides these, there are several other advances to the programming and data model of large graphs, and the runtime execution of the graph platforms that need to be examined. These are discussed below.

7.2.1 Incremental Processing Models

Regardless of the type of framework or algorithms used for processing big graphs, or how large the graph is, data can be processed in three different ways as shown in Figure 7.2.

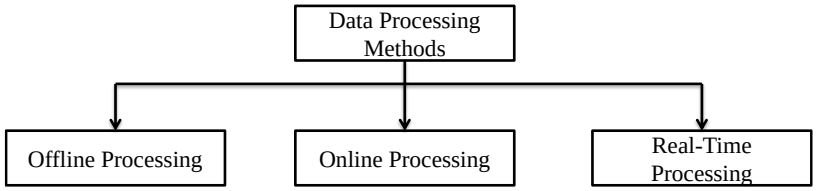


Figure 7-2 Data processing approaches

According to the problem domain that a framework wants to present solutions for, each data processing approach shown above can be considered in the framework. Offline processing (batch processing) is done where a number of analogous tasks are gathered together to be processed by a computing system all at once instead of individually. In this method, which is used in many graph processing frameworks, the whole graph dataset is loaded into the system, processed for an application, and the results will be return to the user. The original graph is not changed externally, other than through modifications by the running application, and this leads to predictable partitioning and scheduling strategies which make their design easy.

In online processing, user can communicate with the system and can make changes to the graph data stored in the system. Thus, the system will be updated automatically and re-process the data with new values periodically or based on user-defined events, which is not necessarily real-time and immediate.

Real-time processing allows the graph to change over time based on incremental updates that it receives to the graph topology or properties. Processing such dynamic graphs is more like an event-driven system where sensors may generate a stream of updates about a vertex or edge that the sensor represents (e.g., road network with traffic cameras or sensors). Real-time processing requires that the computation should be done immediately after the changes happen to the data, and the updated results should be returned with very short delays. Sometimes, the operations may be performed on the delta events themselves before they are actually applied to the graph

data. Such requirements are increasingly important in competitive businesses such as social networks, and in IoT domains. There is limited work in the area of real-time processing of large dynamic graphs. For example, Twitter uses a graph-based content recommendation engine called GraphJet [203] which is an in-memory framework that supplements real-time with batch processing by keeping real-time bipartite interplay graph between tweets and users.

The temporal dimension can also come through the notion of time-series graphs where different states of a graph are available, and the application has to operate over both the spatial and temporal dimension. However, this data base be collected a priori and available offline, and distinct from the changing states of the graph arriving in real-time. For example, GoFFish operates over time-series graphs for algorithms such as time-dependent shortest path and tracking meme propagation [208].

7.2.2 Complex Workflows

A workflow is a dependency graph of different tasks that should be done in a specific order to complete a bigger job. Current graph processing frameworks are based on very simple workflows, typically singleton workflows with one operation executed in a data-parallel manner. They pick a dataset and an algorithm and execute the algorithm on the data. They usually try to solve very simple problems such as finding shortest path or PageRank problem. But, many real world problems are not as easy as this. For example, in a social network a typical scenario can be like this: an algorithm finds all friends and followers of somebody, then finds the common interests between them using another algorithm, draws a map of his/her communication history, combines all these information with the information from other people in that city to find the whole trends and so on. Such a complicated series of processes cannot be modeled seamlessly based on existing graph processing systems without manually creating multiple jobs and passing data explicitly between them through the file system. While there is limited work on Master-Compute model that allows a master task to change the phase of computation on the workers, this can be used to model only simple sequential

operations on a single graph rather than more complex operations that may even span different graphs. So, new frameworks are needed to allow the users to perform more complicated operations.

On the other hand, such complex scenarios also require efficient resource provisioning. That is, proper scheduling is critical to minimize the monetary costs and execution time on one side, and improve resource utilization and performance of the system on the other side. Graph tasks in the workflow may have different processing needs, and may arrive at different intervals, and with different priorities and profitability metrics. Managing these graph workloads offer novel challenges as well. Some research issues on this include:

- How to schedule complex graph workflows to gain minimum cost and maximum resource utilization?
- What factors influence workflow management in graph processing systems considering graph algorithms characteristics and features of graph datasets?
- How does workflow management in graph processing frameworks–specially for complex scenarios- affect the energy consumption of resources?

7.2.3 Graph Databases

Relational databases have existed since the 1980s, and have grown mature. While they deal well with structured data tuples stored in tables, their use for storing and querying graph datasets is limited. As discussed in Chapter 2, graph database, while not a novel concept, are still in their early stages when considering large property and semantic graphs. It is because relationships in a graph database are much stronger than those hypostatized at runtime in a relational database since they are being treated as high priority entities [188]. Relational databases are much slower than graph databases for connected data, hence using graph databases is recommended for highly connected environment and applications such as social networks, IoT, business transactions, and Web searching.

Despite the usefulness of graph databases in the aforementioned environments, they are not as mature as relational databases particularly in terms of tools for data mining

purposes on massive graph data on distributed systems. Therefore, future directions for research include:

- What are the canonical query models for static and dynamic graphs? What is the equivalent of a relational algebra for graph databases?
- Supporting graph queries in combination with graph kernel algorithms, e.g., find all websites hosted in Australia (property) whose PageRank (algorithm output) is greater than X.
- What are the cost models to be developed for efficient execution of graph queries on distributed environments?
- Improve the ability to sustain low-latency processing of large numbers of transactional graph queries on distributed and elastic systems like Cloud.
- How can analysis be performed across data stored in traditional relational databases and graph databases seamlessly and effectively?
- How do we manage distributed data and indexes in graph databases that have data constantly changing or streaming in?

7.2.4 Cloud Features and Cost Models

The Cloud computing paradigm has modified hardware, software and datacenters implementation and design. It offers new economical and technological solutions such as utilizing distributed computing, pay-as-you-go pricing models and resource elasticity. Cloud computing offers computing as a utility in which users can have accesses to different services according to their needs without heed to where the services are hosted or how they are delivered.

Computing as a service is the infrastructure service most relevant to graph processing. While the scalability offered by VMs has been used for graph processing, these are treated as yet another distributed resource by graph processing frameworks rather than consider their ability to elastically scale, or consider their costs. There is limited work on this regard. As mentioned in this thesis, iGiraph has started to consider cost optimization on clouds. It provides various algorithm classifications and

utilizes dynamic repartitioning mechanisms by which it reduces the number of VMs for the graphs that are shrinking during the operation to decline the price. It also performs better on non-convergent applications compared to other frameworks such as the popular Giraph.

Dindokar et al. [59] have proposed an approach to model the computational behavior of non-stationary graph algorithms using a meta-graph model for subgraph-centric programming model. The meta-graph model is able to offer predictions on the subgraphs that will be active in different supersteps, and this is used to schedule subgraphs to VMs in different supersteps, including elastically scaling the VMs in and out [59]. Their strategies show a pricing reduction by half for large graphs like Orkut for costly graph algorithms like Betweenness centrality, with minimal increase in the runtime relative to static over-provisioning of VMs. Elasticity has also be examined in [167] where it uses two partitioning mechanisms called Contiguous Vertex Repartitioning (CVR) and Ring-based Vertex Repartitioning (RVR) to 1) scale in/out without interfering graph computation, 2) decrease the network overhead after scaling, 3) keep the load balanced by reducing stragglers across servers.

Cloud providers have different cost models for their VM resources, typically, on-demand VMs that you pay for based on the minutes or hours used, and spot VMs that have dynamic pricing based on demand-supply, and are pre-emptible when the demand out-strips supply. Spot VMs are much cheaper than on-demand VMs and their use can also be explored for large graph applications, while addressing the faults that can occur due to out of bid event when prices spike. While this has been examined for applications like MapReduce, there is no work in this regard yet for graph processing.

Service level agreement (SLA) [175] is a contract between a service provider and a service user to define the service features, the time for delivering the service, the steps that should be taken in the case of service crashes, service domain, prices, etc. Using SLAs, both user and provider can ensure that the service is delivered exactly based on what had been agreed upon and penalties can be applied in case of commitment violation. Quality of a service (QoS) [18] provides a level of performance, availability and reliability offered by software, platform and infrastructure that the service is

hosted on them. If we consider graph processing as a service then the quality of this service should be in an acceptable level from both provider and customer points of views. According to the aforementioned SLA and QoS definitions, and taking graph processing characteristics into consideration, some research directions can be defined as follow:

- Which parameters have the most impact on the performance of a graph processing service and quality of that?
- What factors should be considered for selecting appropriate graph processing service among other analogous services?
- How SLA-based resource provisioning and scheduling mechanisms for managing graph processing systems and services can be?

7.2.5 Network Optimizations

The network communication and messaging aspects are less studied in current graph processing frameworks. The factors such as network latency, network bandwidth, network traffic and topology can affect the runtime performance of the system. The problem also becomes more complicated when it comes to the Cloud environments. Most existing distributed graph frameworks have been developed for integrated clusters in which resource management and communication is more predictable. But in a Cloud-based framework, the network performance can be variable and VM placement not in the control of users. Hence it becomes essential to consider network factors. Unlike earlier works that considered the role of the network as trivial in graph processing [172], particularly for the graphs that can fit into the memory of a single machine, most recent experiments showed that the network plays a major role in the performance of a graph processing system whether the graph can fit in the memory of a single machine or it is processed on a distributed system [155]. For example, as we demonstrated in Chapter2, allocating larger or denser partitions to the machines with higher bandwidth on one side and reducing the network traffic by decreasing the number of messages transferred between machines on the other hand can enhance the

efficiency of the system. Nevertheless, more research is needed to study other parameters such as topology, latency, etc. and how they affect the processing.

7.2.6 Graph Compression

According to [204], processing large graphs on share memory can be remarkably quicker than processing in a distributed memory environment. Although the amount of data created in the form of graph is growing every day, the capacity of available memory is also increasing which enables very large graph datasets to be fit into memory of a single machine. However, improving the space utilization and execution time of graph algorithms has become crucial. This leads towards compressing graphs to use the memory efficiently.

Graph compression is a technique that has been investigated in the past in frameworks such as WebGraph that could store Web-based graphs using graph compression algorithms such as referentiation, intervalisation, etc. in a limited memory. By the emergence of graph processing frameworks, some systems started proposing similar mechanism to process large-scale graphs. Ligra+ [Shun et al 2015], for example, is a shared memory graph processing system that is developed based on Ligra to reduce memory usage. Ligra+ combines encoding (compression) and decoding techniques utilizing byte codes and nibble codes to represent data. Vertices are being parallelized in encoding where edge list of each vertex will be compressed by coding the differences between source and target vertices of sequential edges. This framework uses two separate methods for decoding out-edge and in-edge lists. Compression on single machines has been implemented on a number of frameworks [145] [194].

Compression techniques have been used for both single machine and multi-computing frameworks. Some works proposed a compression mechanism to optimize memory usage on distributed frameworks. Their solution which has been developed based on Pregel paradigm includes: 1) considering out-edges of each vertex as a row in the graph neighboring matrix for the compression to efficiently represent the space, 2) quick mining the graph without decompression, and 3) considering memory limitation to operate on graph algorithms. GBASE is another distributed framework that utilizes

graph compression. To store the graph efficiently, GBASE uses block compression by creating multiple regions that contain adjacency vertices.

Apart from various compression techniques that have been implemented on graph processing frameworks, there are some issues that make this topic promising for further research.

- Although compression helps in reducing memory utilization, encoding and decoding data are time-consuming and current solutions have made limited improvement in execution time of the operations.
- Some compression techniques might positively influence particular graph algorithms but not the others. Is there any mechanisms that can be useful for different types of graph algorithms? How about switching between different techniques based on the graph application that is being used?
- How do compression mechanisms affect the bandwidth and other computing resources in a single server or distributed environment?

7.2.7 Energy-efficient Resource Allocation

Requests for cloud computing resources are increasing constantly since many years ago as more applications are being migrated to clouds every day. This encourages cloud providers to raise the capacity of datacenters to appropriately serve the growing demands of new clients. One of the major issues of increasing the capacity of datacenters is that more energy is needed to power this big infrastructure. However, a big fraction of the provided energy is being wasted due to various reasons such as inefficient resource provisioning, costly communications, I/O inefficiency, ineffective program architecture, old servers, etc. in the past few years, many works have been done on replacing green power generated energy such as wind, sun and water to supply the needed energy for datacenters and reduce carbon emission. Scheduling and provisioning of resources for various applications on cloud infrastructure in a way that increases usage of green nodes or optimizes the number of required machines will help to decrease the negative impacts of energy wastage. More advanced techniques can be proposed for graph processing systems to achieve this goal.

7.2.8 Other Improvements

Since scalable graph processing is still in its infancy, there are many open issues to improve the performance and features of each component discussed in Chapter 2, and the overall performance of the system. For example, read and write from/to disk is costly in these systems and usually acts as a bottleneck. In many researches such as [262] and [167] SSD (solid state drive) is used as a faster storage device compared to traditional HDDs (hard disk drive) and cheaper compared to main memory. Further, efficient storage models for graph datasets on disks also need to be explored. For e.g., when processing large graphs, the time to load data from disk to memory can outstrip the time to perform the analysis. Compact and compressed graph data representation on disk, loading necessary subsets of the graph on-demand, and support for efficient storage of property graph are some novel topics to explore. Literature has also examined processing of large graphs on single machines and tries to keep the whole graph and computation results in memory. They rely on memory costs dropping and capacities increasing with new technologies like 3D stacked RAM, when single machines will become viable even for billion-vertex graphs. So there are several aspects of storage and memory management that can be explored.

In addition to these, other parts of graph processing system pipeline can be improved as well. These include:

- What initial partitioning or pre-processing techniques can improve the performance of the system and speed up the computation process? How can repartitioning improve the efficiency of the system and if it can speed up the computation process?
- How can we better model and predict the behavior of different graph algorithms for graphs with different characteristics, such as power law, small world, planar, etc.? How are these affected by the different programming models? Can we use these to determine the ideal graph processing technique or strategy to be chosen, e.g., synchronous vs. asynchronous algorithms, computation-bound algorithms vs. memory-bound algorithms, denser datasets vs. less dense datasets and so on.

- Are there any computational mechanisms that use less memory size or can reduce network traffic by reducing the number of messages between machines?
- What fault-tolerance techniques can be used other than check-pointing to improve system reliability and performance?
- What resource provisioning and scheduling algorithms can be used to optimize the processing framework particularly in a competitive environment such as cloud spot-markets?

7.3 Final Remarks

The introduction and provisioning of new features such as unprecedented level of scalability, elasticity and pay-as-you-go models by the cloud computing paradigm on one side and the expanding popularity of large-scale graph processing on the other side have provided opportunities for both technologies to benefit from each other's traits and serve other areas. In this thesis, we used the several characteristics of cloud computing environments to propose solutions in order to empower distributed processing of large-scale graphs. We investigated and implemented various techniques such as different approaches for dynamic repartitioning of the graph during the processing, network-aware scheduling algorithms, methods for taking advantage of heterogeneous resources that are provided in the cloud, and monitoring and maintaining quality of service (QoS) while focusing on minimizing monetary cost as a major factor for service selection. The proposed solutions in this thesis provide direction towards a comprehensive graph processing-as-a-service (GPaaS) paradigm through which analytics in other computing environments such as Internet of Things (IoT) and Edge/Fog computing can be conducted and managed more effective.

BIBLIOGRAPHY

[1] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-Law Graphs," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS'6)*, Rhodes Island, Greece, 2006, pp. 124-124.

[2] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman, "Vision Paper: Towards an Understanding of the Limits of Map-Reduce Computation," in *Proceedings of Cloud Futures 2012 Workshop*, Berkeley, California, USA, 2012.

[3] F. Akbari, A. Hooshang Tajfar, and A. Farhoodi Nejad, "Graph-Based Friend Recommendation in Social Networks Using Artificial Bee Colony," in *Proceedings of IEEE 11th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, Chengdu, China, 2013, pp. 464-468.

[4] A.r Alexandrov et al., "The Stratosphere platform for big data analytics," *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 23, no. 6, pp. 939-964, 2014.

[5] (2015) AllegroGraph. [Online]. http://allegrograph.com/

[6] R. Allen. (2017, Feb.) What happens online in 60 seconds? [Online]. https://www.smartinsights.com/internet-marketing-statistics/happens-online-60-seconds/

[7] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of 9th ACM conference on Computer and*

communications security (CCS '02), Washington, DC, USA, 2002, pp. 217-224.

[8] K. Andreev and H. Racke, "Balanced Graph Partitioning," in *Proceedings of 16th annual ACM symposium on Parallelism in algorithms and architectures (SPAA'04)*, Barcelona, Spain, 2004, pp. 120-124.

[9] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1-39, 2008.

[10] (2015) Apache Cassandra. [Online]. http://cassandra.apache.org/

[11] Apache Giraph. [Online]. http://giraph.apache.org/

[12] Apache Hadoop. [Online]. http://hadoop.apache.org/

[13] Apache Hama. [Online]. https://hama.apache.org/

[14] (2015, July) Apache Hbase. [Online]. http://hbase.apache.org/

[15] Apache Hive TM. [Online]. https://hive.apache.org/

[16] Apache Pig. [Online]. https://pig.apache.org/

[17] (2015) ArangoDB. [Online]. https://www.arangodb.com/

[18] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-Service in Cloud Computing: Modeling Techniques and Their Applications," *Journal of Internet Services and Applications*, vol. 5, no. 11, pp. 1-17, 2014.

[19] D. A. Bader and K. Madduri, "SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Systems (IPDPS'08)*, Miami, FL, USA, 2009.

[20] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Graph Partitioning and Graph Clustering*. Atlanta, GA, US: American Mathematical Society, 2013.

[21] H. Bagci and P. Karagoz, "Context-aware location recommendation by using a random walk-based approach," *Knowledge and Information Systems*, 2015.

[22] M. J. Bannister and D. Eppstein, "Randomized Speedup of the Bellman-Ford Algorithm," In *Proeedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO'12,* Kyoto, Japan, 2012, pp. 41-47.

[23] C. Battaglino, P. Pienta, and R. Vuduc, "GraSP: Distributed Streaming Graph Partitioning," in *Proceedings of the 1st High Performance Graph Mining workshop (HPGM '15)*, Sydney, Australia, 2015.

[24] S. Beamer, K. Asanovic, and D. Patterson, "Direction-Optimizing Breadth-First Search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, Utah, 2012.

[25] 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 *Proceedings of Advances in Service-Oriented and Cloud Computing (ESOCC 2014)*, Manchester, UK, 2014.

[26] T. Beseri Sevim, H. Kutucu, and M. Ersen Berberler, "New Mathematical Model for Finding Minimum Vertex Cut Set," in *Proceedings of International Conference on Problems of Cybernetics and Informatics (PCI'2012)*, BAKU, AZERBAIJAN, 2012, pp. 143-144.

[27] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*, Washington, DC, USA, 2017.

[28] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing," in *Proceedings of IEEE 27th International Conference on Data Engineering (ICDE '11)*, Hannover, Germany, 2011, pp. 1151-1162.

[29] F. Bourse, M. Lelarge, and M. Vojnovi, "Balanced Graph Edge Partition," in *Proceedings of 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '14)*, New York, NY, US, 2014, pp. 1456-1465.

[30] Y. Boykov and V. Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision," *IEEE Transaction on*

Pattern Analysis and Machine Intelligence, vol. 26, no. 9, pp. 1124-1137, 2004.

[31] T. Britton, M. Deijfen, and A. Martin-Lof, "Generating Simple Random Graphs with Prescribed Degree Distribution," *Journal of Statistical Physics*, vol. 124, no. 6, pp. 1377-1397, 2006.

[32] 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.

[33] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," *The VLDB Endowment*, vol. 3, no. 1-2, pp. 285-296, 2010.

[34] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent Advances in Graph Partitioning," *arXiv preprint*, *arXiv:1311.3144*, 2013.

[35] 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 Computer Systems*, vol. 25, no. 6, pp. 599-616, 2009.

[36] L. Cao. (2011) GoldenOrb. [Online]. https://github.com/jzachr/goldenorb

[37] U. Catalyurek and C. Aykanat, "Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication," in *Proceedings of third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR '96)*, Santa Barbara, CA, USA, 1996, pp. 75-86.

[38] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system," in *Proceedings of 7th ACM SIGCOMM conference on Internet measurement (IMC '07)*, San Diego, CA, 2007, pp. 1-14.

[39] C. Chambers et al., "FlumeJava: Easy, Efficient Data-Parallel Pipelines," in *Proceedings of 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, Toronto, Ontario, Canada, 2010, pp. 363-375.

[40] K. Mani Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63-75, 1985.

[41] S. Che, "GasCL: A Vertex-Centric Graph Model for GPUs," in *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2014, pp. 1-6.

[42] Q. Chen et al., "GraphHP: A Hybrid Platform for Iterative Graph Processing.," *arXiv preprint*, *arXiv:1706.07221*, Technical Report 2014.

[43] G. Chen, Z. Fan, and X. Li, "Modelling the Complex Internet Topology," *Complex Dynamics in Communication Networ*ks, Springer, Berlin, Heidelberg, pp. 213-234, 2005.

[44] R. Cheng et al., "Kineograph: Taking the Pulse of a Fast-Changing and Connected World," in *Proceedings of 7th ACM european conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, 2012, pp. 85-98.

[45] Y. Chen, Y. H Lee, W. E. Wong, and D Guo, "A Race Condition Graph for Concurrent Program Behavior," in *Proceedings of 3rd International Conference on Intelligent System and Knowledge Engineering (ISKE'08)*, Xiamen, China, 2008, pp. 662-667.

[46] R. Chen, X. Weng, B. He, and M. Yang, "Large Graph Processing in the Cloud," in *Proceedings of ACM SIGMOD International Conference on Management of data (SIGMOD '10)*, Indianapolis, Indiana, USA, 2010, pp. 1123-1126.

[47] R. Chen et al., "Improving Large Graph Processing on Partitioned Graphs in the Cloud," in *Proceedings of third ACM Symposium on Cloud Computing (SoCC '12)*, San Jose, CA, 2012.

[48] A. Ching. Scaling Apache Giraph to a Trillion Edges. [Online]. https://www.facebook.com/notes/facebookengineering/scaling-apache-giraph-to-a-trillionedges/10151617006153920

[49] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale," *The VLDB Endowment*,

vol. 8, no. 12, pp. 1804-1815, 2015.

[50] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic, "Reliable Distributed Storage," *Computer*, vol. 42, no. 4, pp. 60-67, 2009.

[51] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29-41, 2009.

[52] The European Commission, "Social Networks Overview: Current Trends and Research Challenges," Information Society and Media, Luxembourg, 2010.

[53] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems Concepts and Design*, 5th ed. Boston, Massachusetts, US: Addison-Wesley, 2012.

[54] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*, Monterey, California, USA, 2016.

[55] Committee on the Analysis of Massive Data, Theoretical Statistics Committee on Applied, Their ApplicationsApplications Board on Mathematical Sciences, and Physical Sciences Division on Engineering, and National Research Council, *Frontiers in Massive Data Analysis*.: The National Academies Press, 2013.

[56] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters:," in *Proceedings of Sixth Symposium on Operating Systems Design and Implementation(OSDI '04)*, San Francisco, California, USA , 2004.

[57] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel Hypergraph Partitioning for Scientific Computing," in *Proceedings of 20th international conference on Parallel and distributed processing (IPDPS'06)*, Rhodes Island, Greece, 2006.

[58] M. D. de Assuncao, R. N. Calheiros, S. Bianchi, M. A.S. Netto, and R. Buyya, "Big Data computing and clouds: Trends and future directions," *Journal of Parallel and Distributed Systems*, vol. 79-80, pp. 3-15, 2015.

[59] R. Dindokar and Y. Simmhan, "Elastic Partition Placement for Non-stationary Graph Algorithms," in *Proceedings of the 16th IEEE/ACM International*

Symposium on Cluster, Cloud and Grid Computing (CCGrid'16), Cartagena, Colombia, 2016.

[60] N. Doekemeijer and A. Lucia Varbanescu, "A Survey of Parallel Graph Processing Frameworks," Delft, The Netherlands, Technical report 2014.

[61] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. Lluis Larriba-Pey, "A Discussion on the Design of Graph Database Benchmarks," in *Proceedings of the Second TPC Technology Conference (TPCTC'10)*, Singapore, Springer, 2010.

[62] D. Ediger and D. A. Bader, "Investigating Graph Algorithms in the BSP Model on the Cray XMT," in *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, Boston, Massachusetts, USA, 2013, pp. 1638-1645.

[63] T. Egan, 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, pp. 12-21.

[64] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302-1326, 2013.

[65] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, and S. Bae, "Twister: a runtime for iterative MapReduce," in *Proceedings of 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, Chicago, IL, USA, 2010, pp. 810-818.

[66] D Elena, *Fault-Tolerant Design*. New York, NY, US: Springer-Verlag, 2013.

[67] E. N. Elnozahy, L. Alvist, Y. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375-408, 2002.

[68] H. El-Rewini and M. Abd-El-Barr, "Message Passing Interface (MPI)," *Advanced Computer Architecture and Parallel Processing*, pp. 205-234, 2005.

[69] U. Elsner, *Static and Dynamic Graph Partitioning. A Comparative Study of Existing Algorithms*. Berlin, Germany: Logos Verlag Berlin, 2002.

[70] N. Engelhardt and H. K. So, "Vertex-centric Graph Processing on FPGA," in *Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2016)*, Washington DC, USA, 2016.

[71] U. Feige, M. Hajiaghayi, and J. R. Lee, "Improved Approximation Algorithms for Minimum Weight Vertex Separators," *SIAM Journal on Computing*, vol. 38, no. 2, pp. 629-657, 2008.

[72] F. Fouss, A. Pirotte, J. M. Renders, and M. Saerens, "Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 3, pp. 355-369, 2007.

[73] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *Proceedings of Workshop on GRAph Data management Experiences and Systems (GRADES'14)*, Snowbird, Utah, USA, 2014, pp. 1-6.

[74] N. H. Gehani, "Message passing in Concurrent C: Synchronous versus asynchronous," *Software: Practice and Experience*, vol. 20, no. 6, pp. 571-592, 1990.

[75] F. Geier, "The Differences Between SSD and HDD Technology Regarding Forensic Investigations," Småland, Sweden, Thesis 2015.

[76] A. Geist et al., *PVM: A Parallel Virtual Machine*. Cambridge, MA, USA: MIT Press, 1994.

[77] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu , "On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest," in *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*, Boston, Massachusetts USA, 2013, pp. 851-862.

[78] A. Gharaibeh et al., "Efficient Large-Scale Graph Processing on Hybrid CPU and

GPU Systems," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1563-1586, 2013.

[79] Y. Gong, B. He, and J. Zhong, "Network Performance Aware MPI Collective Communication Operations in the Cloud," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 11, pp. 3079-3089, 2015.

[80] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *Proceedings of 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, 2012, pp. 17-30.

[81] J. E. Gonzalez et al., "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

[82] M. Grabowski, J. Hidders, and J. Sroka, "Representing MapReduce Optimisations in the Nested Relational Calculus," in *Proceedings of 29th British National Conference on Databases*, Oxford, United Kingdom, 2013.

[83] (2015) GraphBase. [Online]. http://graphbase.net/

[84] Chloe Green. (2013, August) Information Age. [Online]. http://www.information-age.com/technology/information-management/123457275/an-introduction-to-graph-databases

[85] D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," in *Proceedings of Parallel Object-Oriented Scientific Computing (POOSC)*, Glasgow, UK, 2005.

[86] Y. Gu, L. Lu, R. Grossman , and A. Yoo, "Processing Massive Sized Graphs Using Sector/Sphere," in *Proceedings of 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, New Orleans, LA, 2010, pp. 1-10.

[87] S. Gunelius. (2014, July) ACI. [Online]. http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/

[88] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema, "An Empirical Performance

Evaluation of GPU-Enabled Graph-Processing Systems," in *Proceedings of 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*, Shenzhen, China, 2015, pp. 423-432.

[89] P. Gupta et al., "WTF: The Who to Follow Service at Twitter," in *Proceedings of 22nd international conference on World Wide Web (WWW '13)*, Rio de Jaeiro, Brazil, 2013, pp. 505-514.

[90] M. Han and K. Daudjee, "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems," *The VLDB Endowment*, vol. 8, no. 9, pp. 950-961, 2015.

[91] W. S. Han et al., "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, Illinois, USA, 2013, pp. 77-85.

[92] W. Han et al., "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, Amsterdam, Netherland, 2014.

[93] Harshvardhan, A. Fidel, N. M. Amato , and L. Rauchwerger, "The STAPL Parallel Graph Library," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC'12)*, Tokyo, Japan, 2013.

[94] B. Hendrickson and J. W. Berry, "Graph Analysis with High-Performance Computing," *Computing in Science and Engineering*, vol. 10, no. 2, pp. 14-19, 2008.

[95] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *the Proceeding of the 10th International Conference on Autonomic Computing (ICAC'13)*, San Jose, CA, 2013.

[96] D. S. Hirschberg , A. K. Chandra , and D. V. Sarwate , "Computing connected components on parallel computers," *Communications of the ACM*, vol. 22, no. 8, pp. 461-464, 1979.

[97] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for Easy and

Efficient Graph Analysis," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, UK, 2012.

[98] I. Hoque and 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.

[99] B. A. Huberman, *The Laws of the Web: Patterns in the Ecology of Information*.: MIT Press Cambridge, 2001.

[100] P. Hudak, ""Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359-411, 1989.

[101] (2015) HyperGraphDb. [Online]. http://www.hypergraphdb.org/index

[102] (2015) InfiniteGraph. [Online]. http://www.objectivity.com/products/infinitegraph/

[103] (2015) InfoGrid. [Online]. http://infogrid.org/trac/

[104] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisboa, Portugal, 2007, pp. 59-72.

[105] Joab Jackson. (2013, Aug.) PCWorld. [Online]. http://www.pcworld.com/article/2046680/facebooks-graph-search-puts-apache-giraph-on-the-map.html

[106] A. K. Jain, *Data Clustering: 50 Years Beyond K-Means*. Berlin, Heidelberg, Germany: Springer, 2008.

[107] N. Jain , G. Liao , and T. L. Willke , "GraphBuilder: Scalable Graph ETL Framework," in *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*, New York, NY, US, 2013.

[108] N. Jamadagni and Y. Simmhan, "GoDB: From Batch Processing to Distributed Querying over Property Graphs," in *Proceedings of the 16th IEEE/ACM*

International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16), Cartagena, Colombia, 2016.

[109] Java Universal Network/Graph Framework. [Online]. http://jung.sourceforge.net/

[110] (2015) JCoreDB. [Online]. https://sites.google.com/site/jcoredb/

[111] S. Jouili and V. Vansteenberghe, "An Empirical Comparison of Graph Databases," in *Proceedings of the International Conference on Social Computing (SocialCom'13)*, Alexandria, VA, 2013, pp. 708 - 715.

[112] U. Kang, H. Tong, J. Sun, C. Y. Lin, and C. Faloutsos, "GBASE: A Scalable and General Graph Management System," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '11)*, San Diego, California, 2011, pp. 1091-1099.

[113] U. Kang, H. Tong, J. Sun, C. Y. Lin, and C. Faloutsos, "GBASE: an efficient analysis platform for large graphs," *The VLDB Journal*, vol. 21, no. 5, pp. 637-650, 2012.

[114] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM '09)*, Miami, FL , 2009, pp. 229-238.

[115] Z. Kaoudi and I. Manolescu, "RDF in the Cloud: A Survey," *The VLDB Journal*, vol. 24, no. 1, pp. 67-91, 2015.

[116] G. Karypis and V. Kumar, "Multilevel Graph Partitioning Schemes," in *Proceedings of the International Conference on Parallel Processing(ICPP'95)*, Raleigh, NC, USA, 1995, pp. 113–122.

[117] S. Deepthi K. et al., "A Survey on Fault Management Techniques in Distributed Computing," in *Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA)*. Berlin,Germany: Springer Berlin Heidelberg, 2013, vol. 199, pp. 593-602.

[118] Z. Khayyat et al., "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, pp. 169-182.

[119] F. Khorasani, K. Vora, R. Gupta , and L. N. Bhuyan, "CuSha: vertex-centric graph processing on GPUs," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC '14)*, Vancouver, BC, Canada, 2014, pp. 239-252.

[120] M. Kim and K. Selçuk Candan, "SBV-Cut: Vertex-Cut Based Graph Partitioning Using Structural Balance Vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285-303, 2012.

[121] H. Kim et al., "DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, Francisco, California, USA, 2016.

[122] S. Koo, S. J. Kwon, S. Kim, and T. Chung , "Dual RAID technique for ensuring high reliability and performance in SSD," in *Proceedings of the IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, Las Vegas, NV, USA, 2015, pp. 399 - 404.

[123] D. C. Kozen, "Depth-First and Breadth-First Search," in *The Design and Analysis of Algorithms*. New York, NY. US: Springer New York, 1992, pp. 19-24.

[124] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "HipG: Parallel Processing of Large-Scale Graphs," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 2, pp. 3-13, 2011.

[125] J. Kunegis, "KONECT - The Koblenz Network Collection," in *Proceedings of International.Web Observatory Workshop*, 2013.

[126] A. Kyrola and C. Guestrin, "GraphChi-DB: Simple Design for a Scalable Graph Database System - on Just a PC," *arXiv preprint*, *arXiv:1403.0701*, 2014.

[127] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, Hallywood, USA, 2012, pp. 31-46.

[128] H. K. Lau, "Error Detection in Swarm Robotics: A Focus on Adaptivity to Dynamic Environments," York, UK, PhD Thesis 2012.

[129] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308-323, 1979.

[130] J. Y. Lee, J. W. Lee, D. W. Cheun, and 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, pp. 261-266.

[131] K. Lee, Y. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11-20, 2011.

[132] K. Lee et al., "Scaling Iterative Graph Computations with GraphMap," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis(SC'15)*, Austin, Texas, USA, 2015.

[133] T. Leimbach et al., "Potential and Impacts of Cloud Computing Services and Social Network Websites," European Parliamentary Research Service, Brussels, 2014.

[134] C. E. Leiserson and T. B. Schardl, "A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Non-determinism of Reducers)," in *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, Thira, Greece, 2010.

[135] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," Pittsburgh, PA, 2008.

[136] Z. Li, T. N. Hung, S. Lu, and R. S. Mong 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, pp. 9-16.

[137] X. Liu, L. Xiao, A. Kreling, and Y. Liu, "Optimizing Overlay Topology by Reducing Cut Vertices," in *Proceedings of the International workshop on Network and operating systems support for digital audio and video (NOSSDAV '06)*, Newport, Rhode Island, 2006.

[138] C. Lochert, M. Mauve, H. Füßler, and H. Hartenstein, "Geographic routing in city scenarios," *ACM SIGMOBILE- Mobile Computing and Communications Review*, vol. 9, no. 1, pp. 69-72, 2005.

[139] B. Lorica, "One year later: some single server systems that can tackle Big Data," in *Proceedings of the Big Data Now*. Sebastopol, CA: O'reilly Media Inc., 2014, pp. 29-30.

[140] B. Lorica. (2013, April) O'Reilly Radar. [Online]. http://radar.oreilly.com/2013/04/single-server-systems-can-tackle-big-data.html

[141] Y. Low et al., "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716-727, 2012.

[142] Y. Low et al., "GraphLab: A New Framework For Parallel Machine Learning," in *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, Catalina Island, USA, 2010.

[143] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation," *The VLDB Endowment*, vol. 8, no. 3, pp. 281-292, 2014.

[144] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5-20, 2007.

[145] S. Maass et al., "Mosaic: Processing a Trillion-Edge Graph on a Single Machine," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, Belgrade, Serbia, 2017, pp. 527-543.

[146] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarría- Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *Proceedings of the*

IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09), Rome, Italy, 2009.

[147] S. Maleki et al., "DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, Istanbul, Turkey, 2016.

[148] G. Malewicz et al., "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-145.

[149] P. Manuel, "A Trust Model of Cloud Computing Based on Quality of Service," *Annals of Operations Research*, vol. 233, no. 1, pp. 281-292, 2013.

[150] (2015) MapGraph. [Online]. http://mapgraph.io/

[151] A. Marburger and B. Westfechtel, "Graph-Based Structural Analysis for Telecommunication Systems," in *Proceedings of the Graph Transformations and Model-Driven Engineering*.: Springer Berlin Heidelberg, 2010, pp. 363-392.

[152] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable Graph Partitioning in the Cloud," in *Proceedings of the IEEE 33rd International Conference on Data Engineering (ICDE'17)*, San Diego, CA, USA, 2015.

[153] R. R. McCune, T. Weninger, and G. Madey, "Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1-39, 2015.

[154] F. McSherry. The impact of fast networks on graph analytics, part 1. [Online]. https://github.com/frankmcsherry/blog/blob/master/posts/2015-07-08.md

[155] F. McSherry and M. Schwarzkopf. (2015, July) The Impact of Fast Networks on Graph Analytics. [Online]. http://www.frankmcsherry.org/pagerank/distributed/performance/2015/07/08/pagerank.html#fn0

[156] K. Mehlhorn and S. Näher, "The LEDA Platform of Combinatorial and Geometric Computing," *Communications of the ACM*, vol. 33, no. 1, pp. 96-102, 1995.

[157] J. Mei, A. Ouyang, and K. Li, "A Profit Maximization Scheme with Guaranteed Quality of Service in Cloud Computing," *IEEE Transactions on Computers*, vol. 64, no. 11, pp. 3064 - 3078, 2015.

[158] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1-14, 2015.

[159] R. C. Murphy and P. M. Kogge, "On The Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications," *IEEE Transactions on Computers*, vol. 56, no. 7, pp. 937-945, 2007.

[160] D. G. Murray et al., "Naiad: A Timely Dataflow System," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, USA, 2013.

[161] K. Najeebullah, K. U. Khan, W. Nawaz , and Y. Lee, "BiShard Parallel Processor: A Disk-Based Processing Engine for Billion-Scale Graphs," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 9, no. 2, pp. 199-212, 2014.

[162] K. Najeebullah, K. U. Khan, Muhammad Waqas Nawaz, and Young-Koo Lee, "BPP: Large Graph Storage for Efficient Disk Based," *arXiv preprint, arXiv:1401.2327*, 2014.

[163] NECTAR Cloud. [Online]. http://nectar.org.au/research-cloud/

[164] (2015, August) Neo4j. [Online]. http://neo4j.com/

[165] D. Nguyen, A. Lenharth, and K.Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, USA, 2013, pp. 456-471.

[166] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases," Waterloo, Ontario, Canada, Technical Report 2015.

[167] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal," in *Proceedings of the International Conference on Systems and Storage (SYSTOR'14)*, Haifa, Israel, 2014, pp. 1-12.

[168] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52-60, 1991.

[169] E. Nurvitadhi et al., "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in *Proceedings of the IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines (FCCM '14)*, Boston, Massachusetts, 2014, pp. 25-28.

[170] (2015) OrientDB. [Online]. http://orientdb.com/orientdb-vs-neo4j/

[171] J. Ousterhout et al., "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92-105, 2010.

[172] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, USA, 2015, pp. 293-307.

[173] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Info Lab, 1998.

[174] G. Pallis, "Cloud Computing: The New Frontier of Internet Computing," *IEEE Internet Computing*, vol. 14, no. 5, pp. 70-73, 2010.

[175] P. Patel, A. Ranabahu, and A. Sheth, "Service Level Agreement in Cloud Computing," Write State University, Dayton, OH, USA, 2009.

[176] A. Paul , "Graph based M2M optimization in an IoT environment," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems(RACS '13)*, Montreal, Canada, 2013, pp. 45-46.

[177] F. Pellegrini , "Current challenges in parallel graph partitioning," *Comptes Rendus Mécanique*, vol. 339, no. 2-3, pp. 90-95, 2011.

[178] C. Pettey. (2011, June) Gartner. [Online]. http://www.gartner.com/newsroom/id/1731916

[179] R. Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, "Interpreting the

Data: Parallel Analysis with Sawzall," *Scientific Programming - Dynamic Grids and Worldwide Computing*, vol. 13, no. 4, pp. 277-298, 2005.

[180] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 4th ed. Springer-Verlag New York: New York, US, 2012.

[181] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale Graph Algorithms.," *Parallel Computing*, vol. 37, no. 9, pp. 610-632, 2011.

[182] R. Power and J. Li, "Piccolo: Building Fast, Distributed Programs with Partitioned Tables," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, Vancouver, Canada, 2010.

[183] K. Prakasam and M. Chandrasekhar. (2010) JPregel. [Online]. http://kowshik.github.io/JPregel/

[184] M. Prigg. Facebook hits one billion users in a single day: Mark Zuckerberg reveals one in seven people on earth used the social network on Monday. [Online]. http://www.dailymail.co.uk/sciencetech/article-3213456/Facebook-s-billion-user-day-Mark-Zuckerberg-reveals-one-seven-people-EARTH-used-social-network-Monday.html

[185] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell, "Supporting On-demand Elasticity in Distributed Graph Processing," in *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E'16)*, Berlin, Germany, 2016, pp. 12-21.

[186] F. Rahimian, A.H. Payberah, S. Girdzijauskas, and S. Haridi, "Distributed Vertex-Cut Partitioning," in *Proceedings of the Distributed Applications and Interoperable Systems*. Berlin, Germany: Springer Berlin Heidelberg, 2014, pp. 186-200.

[187] M. Redekopp, Y. Simmhan, and V. K. Prasan, "Optimizations and Analysis of BSP Graph Processing Models on Public Clouds," in *Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)*, Boston, MA, 2013, pp. 203-214.

[188] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, 2nd ed. Sebastopol, CA:

O'Reilly, 2015.

[189] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using Hadoop on a cluster," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP 2012)*, Bern, Switzerland, 2012.

[190] P. Roy, "A new memetic algorithm with GA crossover technique to solve Single Source Shortest Path (SSSP) problem," in *Proceedings of the 2014 Annual IEEE India Conference (INDICON)*, Pune, India, 2014.

[191] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out Graph Processing from Secondary Storage," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, USA, 2015, pp. 410-424.

[192] A. Roy, I. Mihailovic, and 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, pp. 472-488.

[193] M. Sagharichian, H. Naderi, and M. Haghjoo, "ExPregel: a new computational model for large-scale graph processing," *Concurrency and Computation Practice and Experience*, vol. 27, no. 17, pp. 4954–4969, 2015.

[194] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. Widom, "Graft: A Debugging Tool For Apache Giraph," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, Melbourne, VIC, Australia, 2015, pp. 1403-1408.

[195] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *Proceedings ot the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*, Baltimore, Maryland, 2013.

[196] S. Salihoglu and J. Widom, "Optimizing Graph Algorithms on Pregel-like Systems," *VLDB Endowment*, vol. 7, no. 7, pp. 577-588, 2014.

[197] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel, "Horton+: a distributed system

for processing declarative reachability queries over partitioned graphs," *The VLDB Endowment*, vol. 14, no. 5, pp. 1918-1929, 2013.

[198] K. Schloegel, G. Karypis, and V. Kumar, "Graph Partitioning for High Performance Scientific Simulations," in *CRPC Parallel Computing Handbook*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2001, pp. 491-541.

[199] R. Sedgewick and K. Wayne, *Algorithms (4th Edition)*. Upper Saddle River, NJ: Addison-Wesley Professional, 2011.

[200] D. Sengupta et al., "GraphIn: An Online High Performance Incremental Graph Processing Framework," in *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing*, Grenoble, France, 2016.

[201] Y. Shao, B. Cui, L. Ma, and J. Yao, "PAGE: A Partition Aware Engine for Parallel Graph Computation," in *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM '13)*, Burlingame, CA, USA, 2013.

[202] B. Shao, H. Wang, and 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, pp. 505-516.

[203] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "GraphJet: Real-Time Content Recommendations at Twitter," *The VLDB Endowment*, vol. 9, no. 13, pp. 1281-1292, 2016.

[204] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, Shenzhen, China, 2013, pp. 135-146.

[205] J. Siek, L. Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Upper Saddle River, NJ, USA: Addison-Wesley, 2002.

[206] Y. Simmhan and A. Kumbhare, "Floe: A dynamic, continuous dataflow framework for elastic clouds," *arXiv preprint*, *arXiv:1406.5977*, 2013.

[207] Y. Simmhan, N. Choudhury, C. Wickramaarachchi, and A. Kumbhare,

"Distributed Programming over Time-Series Graphs," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Systems (IPDPS'15)*, Hyderabad, India, 2015.

[208] Y. Simmhan et al., "GoFFish: A Sub-Graph Centric Framework forLarge-Scale Graph Analytics," in *Proceedings of the Euro-Par 2014 Parallel Processing*, Porto, Portugal, 2014, pp. 451-462.

[209] Y. Simmhan et al., "Scalable analytics over distributed time-series graphs using goffish," *arXiv preprint arXiv:1406.5975*, 2014.

[210] C. Snijders, U. Matzat, and U. D. Reips, "Big Data: Big gaps of knowledge in the field of Internet," *International Journal of Internet*, vol. 7, no. 1, pp. 1-5, 2012.

[211] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*, Vienna, Austria, 2018.

[212] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '12)*, Beijing, China, 2012, pp. 1222-1230.

[213] M. A. Stelzner, "2015 Social Media Marketing Industry Report," 2015, [Online]. https://www.socialmediaexaminer.com/social-media-marketing-industry-report-2015/

[214] P. Strandmark and F. Kahl, "Parallel and distributed graph cuts by dual decomposition," *Computer Vision and Image Understanding*, vol. 115, no. 12, pp. 1721-1732, 2011.

[215] P. Stutz, A. Bernstein, and W. Cohen, "Signal/Collect: Graph Algorithms for the (Semantic) Web," in *Proceedings of the 9th international semantic web conference on The semantic web (ISWC'10)*, Shanghai, China, 2010, pp. 764-780.

[216] Z. Sun, H. Wang, H. Wang, B. Shao, and L. Jianzhong, "Efficient Subgraph Matching on Billion Node Graphs," *The VLDB Endowment*, vol. 5, no. 9, pp. 788-799, 2012.

[217] P. Sun, Y. Wen, T. Nguyen Binh Doung, and X. Xiao, "GraphMP: An Efficient Semi-External-Memory Big Graph Processing System on a Single Machine," *arXiv preprint, arXiv:1707.02557*, 2017.

[218] S. Suri and S. Vassilvitskii, "Triangles and the Curse of the Last Reducer," in *Proceedings of the 20th international conference on World wide web (WWW '11)*, Hyderabad, India, 2011, pp. 607-614.

[219] A. S. Szalay, "Extreme Data-Intensive Scientific Computing," *Computing in Science and Engineering*, vol. 13, no. 6, pp. 34-41, 2011.

[220] S. Tasci and 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, pp. 458-469.

[221] D. Gregor, A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," in *Proceedings of the Parallel Object-Oriented Scientific Computing (POOSC'05)*, Glasgow, UK, 2005.

[222] N. T. Bao and T. Suzumura, "Towards Highly Scalable Pregel-Based Graph Processing Platform With x10," in *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*, Rio de Janeiro, Brazil, 2013, pp. 501-508.

[223] Y. Tian, A. Balmin, S. A. Corsten, . Tatikond, and J. McPherson, "From "Think Like a Vertex" to "Think Like a Graph"," *The Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193-204, 2013.

[224] J. Tian, J. Hahner, C. Becker, I. Stepanov, and K. Rothermel, "Graph-based mobility model for mobile ad hoc network simulation," in *Proceedings of the 35th Annual Simulation Symposium*, San Diego, California, 2002, pp. 337-344.

[225] (2015) TITAN Distributed Graph Database. [Online]. http://thinkaurelius.github.io/titan/

[226] Top500. [Online]. https://www.top500.org/

[227] M. Treaster, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems," *ACM Computing Research Repository (CoRR)*, pp. 1-11, 2005.

[228] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining (WSDM '14)*, New York, NY, US, 2014, pp. 333-342.

[229] Twitter. (2012, March) Twitter Blog. [Online]. https://blog.twitter.com/2012/cassovary-big-graph-processing-library

[230] Twitter. (2012, April) Twitter/FlockDB. [Online]. https://github.com/twitter/flockdb#readme

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

[232] L3 M. Vaquero, F3 Cuadrado, D3 Logothetis, and C3 Martella, "xDGP: A Dynamic Graph Processing System with Adaptive Partitioning," in *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13)*, Santa Clara, CA, 2013.

[233] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung , and R. S. Schreiber, "Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, Prague, Czech Republic, 2013, pp. 197-210.

[234] C. Vicknair et al., "A Comparison of a Graph Database and a Relational Database," in *Proceedings of the 48th Annual Southeast Regional Conference(ACM SE '10)*, Oxford, Mississippi, 2010.

[235] Y. Wang et al., "Gunrock: A High-Performance Graph Processing Library on the GPU," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*, San Francisco, CA, USA, 2016.

[236] G. Wang, W. Xie, A. Demers, and 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.

[237] K. Wang and G. Xu, "GraphQ: Graph Query Processing with Abstraction Refinement – Scalable and Programmable Analytics over Very Large Graphs on

a Single PC," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, Santa Clara, CA, USA, 2015.

[238] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based Fault-tolerance for Large-scale Graph Processing," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependabale Systems and Networks (DSN'14),* Atlanta, GA, USA, 2014.

[239] X. Wu, X. Ying, K. Liu, and L. Chen, "A Survey of Algorithms For Privacy-Preservation of Graphs and Social Networks," in *Proceedings of the Advanced in Database Systems*. Boston, MA, USA: Springer, 2010, pp. 421-453.

[240] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*, San Francisco, CA, 2015, pp. 194-204.

[241] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: a resilient distributed graph system on Spark," in *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*, New York, NY, USA, 2013.

[242] K. Xiong and H. Perros, "Service Performance and Analysis in Cloud Computing," in *Proceedings of the World Conference on Services*, Los Angeles, CA, USA, 2009, pp. 1-8.

[243] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande, "GraphGen: Adaptive Graph Processing using Relational Databases," in *Proceedings of the 5th International Workshop on Graph Data Management Experiences and Systems (GRADES'17)*, Chicago, IL, USA, 2017.

[244] N. Xu, L. Chen, and B. Cui, "LogGP: A Log-based Dynamic Graph Partitioning Method," *The VLDB Endowment*, vol. 7, no. 14, pp. 1917-1928, 2014.

[245] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an Efficient, Low-cost System for Concurrent Graph Processing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC*

'14), Vancouver, Canada, 2014.

[246] C. Xu et al., "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, pp. 420-427.

[247] Y. Yamato, "Use case study of HDD-SSD hybrid storage, distributed storage and HDD storage on OpenStack," in *Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15)*, Yokohoma, Japan, 2015, pp. 228-229.

[248] D. Yan, J. Cheng, Y. Lu, and W. Ng, "A Block-Centric Framework for Distributed Computation on Real-World Graphs," *The VLDB Endowment*, vol. 9, no. 7, pp. 1981-1992, 2014.

[249] D. Yan et al., "Quegel: A General-Purpose Query-Centric Framework for Querying Big Graphs," *The VLDB Endowment*, vol. 9, no. 7, pp. 564-575, 2016.

[250] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards Effective Partition Management for Large Graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, USA, 2012.

[251] J. Yan, G. Tan, and N. Sun, "GRE: A Graph Runtime Engine for Large-Scale Distributed Graph-Parallel Applications.," *CoRR abs/1310.5603*, 2013.

[252] E. Yoneki, K. Nilakant, V. Dalibard , and A. Roy , "PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal," in *Proceedings of the International Conference on Systems and Storage (SYSTOR 2014)*, Haifa, Israel, 2014.

[253] P. Yuan et al., "Fast Iterative Graph Computation: A Path Centric Approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, New Orleans, LA, 2014, pp. 401-412.

[254] Y. Yu et al., "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, San Diego, CA, 2008, pp. 1-14.

[255] M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, 2012.

[256] Z. Zha, T. Mei, J. Wang, Z. Wang, and X. S. Hua, "Graph-based semi-supervised learning with multiple labels," *Journal of Visual Communication and Image Representation*, vol. 20, no. 2, pp. 97-103, 2009.

[257] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate large-scale iterative computation through asynchronous accumulative updates," in *Proceedings of the 3rd workshop on Scientific Cloud Computing Date (ScienceCloud '12)*, Delft, the Netherlands, 2012, pp. 13-22.

[258] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation," *IEEE Transaction on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091-2100, 2014.

[259] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "PrIter: A Distributed Framework for Prioritized Iterative Computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1884-1893, 2012.

[260] T. Zhang, J. Zhang, W. Shu, M. Wu, and X. Liang, "Efficient graph computation on hybrid CPU and GPU systems," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1563-1586, 2015.

[261] M. Zhang, Y. Zhuo, C. Wang, M. Gao, and Y. Wu, "GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*, Vienna, Austria, 2018.

[262] D. Zheng et al., "FlashGraph: processing billion-node graphs on an array of commodity SSDs," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, 2015, pp. 45-58.

[263] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE*

Transactions on Parallel and Distributed Systems (TPDS), vol. 25, no. 6, pp. 1543 - 1552, 2013.

[264] J. Zhong and B. He, "Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud," in *Proceedings of the 5th International Conference on Cloud Computing Technology and Science (CloudCom'13)*, Bristol, UK, 2013, pp. 9-16.

[265] J. Zhong, B. He, and Y. Gong, "Network Performance Aware MPI Collective Communication Operations in the Cloud," *IEEE Transactions on Parallel & Distributed Systems*, vol. 26, no. 11, pp. 3079-3089, 2015.

[266] J. Zhou et al., "SCOPE: parallel databases meet MapReduce," *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 21, no. 5, pp. 611-636, 2012.

[267] X. Zhu, W. Chen, W. Zhang, and X. Ma, "Gemini: A Computation-Centric Distributed Graph Processing System," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, 2016.

[268] X. Zhu, W. Han, and W. Chen, "GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC '15)*, Santa Clara, CA, USA, 2015.