Distributed Heterogeneous Systems for Large-Scale Graph Processing

Yong Guo

Distributed Heterogeneous Systems for Large-Scale Graph Processing

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
vrijdag 13 mei 2016 om 15:00 uur

door

Yong GUO

Bachelor of Engineering in Computer Science and Technology,
National University of Defense Technology, China
geboren te Jingzhou, China

This work was carried out in the ASCI graduate school. ASCI dissertation series number 351.

This work was supported by the China Scholarship Council (CSC).

Part of this work has been done in collaboration with Oracle Labs, CA, USA.

Published and distributed by: Yong Guo
E-mail: guoyongnudt@gmail.com

ISBN: 978-94-6328-042-6

Keywords: Distributed Heterogeneous Systems, Graph Processing, Game Trace Archive, Performance Evaluation, Graph Partitioning.

Cover design: Lu Ding and Yong Guo. Some of the cover images are from http://www.58pic.com/.

Printed in the Netherlands by: CPI-Koninklijke Wöhrmann-Zutphen.

To my dear parents

Acknowledgements

I have collected more than twenty PhD dissertations during my PhD study. To be honest, I haven't read the contents of any of them, but I have taken a look at the acknowledgements for all of them because they are quite interesting. I have also dreamed one day I can write my own acknowledgements. Now, this dream has come true. It is my turn to thank the people I have met and to share my sweetest memories with you.

Alex, you are definitely the most important person for my PhD study. Without you, I cannot imagine how I could have conquered so many difficulties and challenges. I cannot forget how you inspired me why online match-based games were important in the coffee room, how you drew the format on the write board when I was stuck with designing the Game Trace Archive, how you spent the entire night to help me meet the deadline, how you tolerated me to conduct the research on my own opinion, how you were pleased when you heard I had a good idea on graph partitioning, and how you abstracted the critical concept of three families of distributed heterogeneous systems for me. You also taught me how to design attractive posters, to make nice slides, to give good presentations, and to sell our ideas. I also miss the days that we played basketball, badminton, and beach volleyball. It is my honour and pleasure to be your PhD student.

Dick, thank you very much for your guidance and supervision. Thanks for your suggestions about my thesis structure and research plans. Thanks for your patience on correcting and polishing my draft manuscripts. Your kind advices and words warmed me up when I did not progress well in my research. I also appreciate your great help on managing the process of my PhD defense.

Ana, I would say you are my third supervisor. Your name appears five times out of nine in my publication list. From our collaboration, I have learnt how to think thoroughly, how to design comprehensive experiments, and how to write good research papers. I am so lucky and pleased to work with you.

Henk, thank you for considering my PhD application and for accepting me as a member of the PDS group. You are a great professor to talk with. I wish you a happy life after your retirement.

I appreciate the supervision and advice from Prof. Xiangke Liao, Prof. Liquan Xiao, and Dr. Shanshan Li in China. You guided me to the research field and taught me basic research skills. After I came to the Netherlands, you were also concerned about my progress and always encouraged me.

I would like to thank my committee members for reviewing my thesis and for giving me valuable comments. I want to thank the China Scholarship Council for the four-year scholarships. I am thankful to my mentor in the Graduate School, Prof. Kofi Makinwa.

Many collaborators helped me during my PhD study. Hassan and Sungpack, thank you for accepting my application and for hosting me for a three-month internship at Oracle Labs. Based on our collaboration, I accomplished the first journal article of my research. I also like the way how you supervise and manage the team. Marcin, thank you for evaluating different graph-processing systems. Claudio, thank you for your support on Giraph and for your ideas on modeling the performance of graph-processing systems. Theodore, thank you for discussing with me many topics on graph processing.

Siqi, your help started before I came to the Netherlands. You told me about the PDS group and gave me important tips before my travel. You shared with me your ideas about how to do good research. You taught me how to use different tools to process massive data. You improved my debating skills because you have so many amazing opinions that I cannot agree with. You encouraged me when I was upset. Your support never ends even though you graduated. You are telling me how to prepare for the PhD defense. I am very glad to hear that you are married and I wish you a happy new life.

Bogdan, my friend and brother, there are so many great moments we have experienced. We took several PhD courses together. We exchanged information about our progress towards graduation. We shared with each other our experience on paper submission. We played tennis together. I appreciate this very much also because you helped me a lot. You gave me advices and encouraged me when I lost myself. You helped me understand Hadoop and its configuration. You read my articles, provided valuable comments, and even polished some parts of my writings. I believe you will finish your PhD with a big success and my best wishes to you and Elena.

Jianbin, you set a good example for me on doing research. You are also kind-hearted to give a helping hand when I have problems. Thanks a lot. It's great to hear that you are a father of a little boy.

Boxun and Adele, the “first and second senior fellows”, I cannot forget our wonderful new year parties, so much fun and so much wine. Although the PhD study is hard, you can always take it easy. I have learnt a lot from your attitudes on life and study.

Otto, thanks for helping me write my first research paper, answering my questions, maintaining the SVN, and allowing me to change my password so many times.

Nezih, Kefeng, Yunhua, Alex, Lipu, Wing, Sietse, and Stijn, you are very nice and friendly officemates. It is great to talk with you, no matter the scientific problems or other random topics. Stijn, thank you very much for translating the summary of my thesis into Dutch.

I am very thankful to my colleagues, Rahim, Mihai, Riccardo, Niels, Elric, Alexey, Boudewijn, Tamás, Arno, Nitin, Tim, Lucia, and Dimitra. You made the PDS group such a nice and interesting place to do research and study. I enjoy our monthly dinners, parties, movie nights, and sports.

Paulo, Munire, Stephen, Ilse, Shemara, Rina, Franca, and Monique, thank you for solving technical problems and administrative issues. Your support was so important and helped me concentrate on my research.

Mingxin, we took the same flight to the Netherlands to pursue our PhD degrees. Now, we are about to finish our PhD study almost at the same time. You are a great friend that I can always share my ideas and emotions with during the entire period of my PhD. Your fearlessness always encourages me. I wish you will have an excellent PhD defense.

Xiaoqin, I appreciate your patience and kindness when we complained about our life and study. You are so warm to listen and to cheer us up. I would also like to thank the spiciest dinner I have had in the Netherlands. I hope you can keep enjoying your PhD study and life.

Yichen, Lu, Shiqian, Ya, Zhiyu, Ye, Yikun, and Siqi, playing tennis with you is one of the best experience in the last year of my PhD. Thank you for beating me so hard in our matches, for teaching me how to improve my skills, and for so many dinners and activities. You are a group of people full of energy and enthusiasm. I believe you will succeed in your study, work, and business. Lu, you helped me design the cover of this thesis and I am very grateful for that.

I am so lucky to have many friends in the Netherlands. They are: Hao, Chao, Xian, Yao, Yongchang, Yanqing, Shuai, Wei, Tiantian, Xinchao, Shanshan, Yue, Jian, Xiang, Yibin, and Ye. I have so many wonderful memories with you, for dinners, trips, parties, games, and sports.

I also want to thank my friends in China. Yi and Peixin, thanks for dealing with my administrative issues at NUDT. Dan, Xinye, Haidong, Chengye, and Hao, I really enjoy our chats. Jian, Yilun, Xiaowei, and Cong, thanks for adding me in the "new daddy" chatting group and for sharing your happiness, although I am not a dad. I wish you all have a better life.

My special thanks to Jie, thank you for sharing success and happiness, and for overcoming challenges and difficulties with me. The future will be colorful and beautiful for both of us.

Finally, I would like to give my deepest thanks to my beloved parents. Thank you so much for your tolerance, patience, and support. The young boy in your mind has grown up. I love you.

Yong Guo
Delft, April 2016

Contents

Chapter 1

Introduction

Graph processing is of increasing interest for many business applications and scientific areas, such as online social networks [84], bioinformatics [64], and online gaming [54]. To answer to the growing diversity of graph datasets and graph-processing algorithms, developers and system integrators have created a large variety of graph-processing systems (platforms)—which we define as the combined hardware, software, and programming system that is being used to complete a graph processing task. Many system challenges need to be addressed before graph processing can be made available to the masses. Emerging fields of application for graph processing, such as online gaming, have new data characteristics. The diversity of graph-processing systems makes it difficult for users to select a system for their own applications, because the performance of such systems has not been evaluated and compared thoroughly. Although great benefits in performance could be achieved by using *both* CPUs and GPUs on *multiple* machines, currently there is no publicly available graph-processing system with such capability. Addressing these challenges is further complicated by the need to combine new models and other *fundamental research* methods with system implementation and other *applied research* methods. *In this thesis we address important challenges of graph processing by conducting fundamental and applied research, from collecting and sharing graph datasets, though gaining knowledge about the performance of previous graph-processing systems, to designing, implementing, and evaluating new methods, and even entire systems for graph-processing.*

Extracting and understanding information from large-scale graphs is essential to the operation of both public and private organizations, and by now follows the typical graph-processing workflow depicted in Figure 1.1. By applying algorithms to input datasets, analysts are able to predict the behavior of the customer, and tune and develop new applications and services. For example, in the area of online social networks, better and more accurate content recommendation can be made based on identifying and using the friendship between users [54]. In the area of online gaming, understanding the relationships and interactions among players can help gaming operators improve match-making

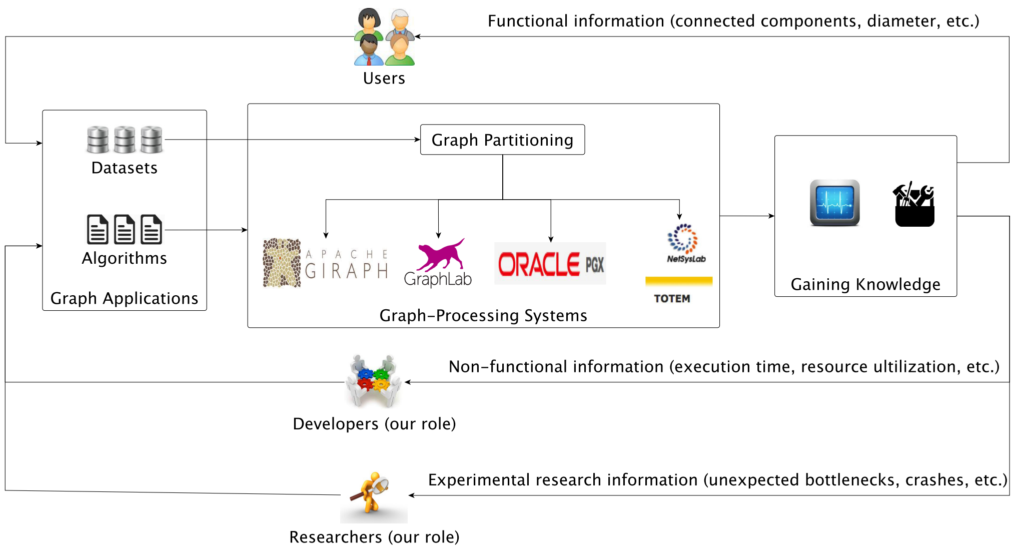

Users
Functional information (connected components, diameter, etc.)
Datasets
Algorithms
Graph Applications
Graph Partitioning
APACHE GIRAPH
GraphLab
ORACLE PGX
NetSysLab
TOTEM
Graph−Processing Systems
Gaining Knowledge
Non−functional information (execution time, resource ultilization, etc.)
Developers (our role)
Experimental research information (unexpected bottlenecks, crashes, etc.)
Researchers (our role)

Figure 1.1: A typical graph-processing workflow.

systems [37] and build reputation systems [69], which are crucial to the Quality of Experience for players. The scale and other characteristics of graphs vary for different domains. In general, graphs grow over time because more entities are included and more relationships are created. This makes graph processing too large for single-machine infrastructures and leads to the development of distributed systems, such as Apache Giraph [43] in Figure 1.1. Due to the scale of typical input datasets, distributed graph-processing systems involve a crucial data partitioning step.

Many graph-processing systems already exist, and more are newly developed. As the graph size and structure become larger and more complex, and also as the graph-processing analysis aims to obtain more useful and complex information, it is increasingly more difficult to handle and analyze modern graphs. Generic distributed data-processing systems, such as Hadoop, have been used to process large-scale graphs. However, the performance of Hadoop on processing graphs is poor (quantified for the first time during the course of this thesis)—this has become common knowledge for the graph-processing community. In recent years, many graph-processing systems have been designed and developed. For example, Apache Giraph [43], the open source version of Google Pregel [92], is one of the most popular distributed graph-processing systems that can handle large-scale graphs by using multiple machines. Other distributed graph-processing systems, such as GraphLab [87], GraphX [46], and PGX.D [61], are also becoming popular because of their high performance and expressive programming models. To further improve performance, another branch of graph-processing systems is emerging: systems using accelerators such as GPUs. GPU-enabled systems, such as TOTEM [41], Medusa [152], and Gunrock [137], can under specific circumstances achieve much higher performance than CPU-only graph-processing systems [59].

Understanding the new data characteristics in the emerging field of graph processing is non-trivial. Among the many business applications and scientific fields of graph processing, we envision that graph analytics in for example *online gaming* can deliver business-critical information (e.g., predicting the number of online players and identifying key social players) for companies such as Blizzard (World of Warcraft [82]) and Zynga (CityVille [155]), and for other leaders in this multi-billion industry. In work leading to this thesis, we found that in many game traces the relationships (e.g., play with, send message to, member of) between many kinds of game entities (e.g., player, group, etc) are common and significant for the operation of these games. Online gaming data covers millions of players and various relationships between them, and is usually large-scale and rich with information. Online gaming data is a very interesting area for big data analysis and it is also an important data source for graph processing. However, the complexity of data from diverse games makes it difficult to *share gaming graphs*. A unified format for gaming graphs and other challenges, such as fast format converting and data anonymizing, should also be addressed.

The diversity of graph-processing systems provides more choices for users, but it also increases the challenge of *system selection*. It is difficult for users to make a selection of graph-processing systems, according to their specific graph-processing applications. Performance evaluation on graph-processing systems can address this problem. However, conducting a thorough and fair performance evaluation has many methodological and practical challenges, such as defining a fair evaluation process, selecting and designing performance metrics, selecting representative graph-processing applications (datasets and algorithms), ensuring the scalability and portability of the evaluation, and reporting results.

Related to the challenge of designing new methods and even entire systems for graph-processing, for distributed CPU-based systems, *graph partitioning* is a mandatory process that represents new *methods* for distributing workload to multiple working machines. The quality of graph partitioning can significantly influence the performance of graph-processing systems. With GPUs becoming increasingly more powerful and affordable, even Small and Medium Enterprises (SMEs) who could previously invest only in CPU-based commodity clusters, can now afford to buy a heterogeneous environment. However, current graph-processing systems cannot operate on both distributed and heterogeneous settings. Designing *distributed heterogeneous graph-processing systems* can not only help SMEs fully use the hardware resources they invested in, but can also help them improve the performance of graph processing in their specific context. To achieve high performance in such systems, the heterogeneity of CPUs and GPUs should be considered and addressed, and the workload partitioned between CPUs and GPUs should be carefully balanced. Thus, graph partitioning becomes very important.

Graph processing includes a wide range of research directions, such as graph representation and pre-processing, graph partitioning, algorithm tuning, graph-processing system design, software engineering and prototyping of graph-processing systems, performance evaluation of graph-processing systems, etc. In this thesis, we identify at least three categories of research in graph-processing systems: understanding graph applications (*application*), gaining knowledge about the performance of previous graph-processing systems (*knowledge*), and designing methods and systems for graph-processing (*design*). We conduct fundamental and applied research on each of these three directions. For application research, we aim to understand the characteristics of gaming graphs and to design a unified format for data sharing (fundamental), and also we use this format to store and share multiple game traces (applied). For knowledge research, we design an empirical method to define which performance aspects we should measure and to understand possible bottlenecks of existing graph-processing systems (fundamental), and also we use this method to evaluate the performance of multiple systems (applied). For design research, we identify different types of graph-partitioning policies and different families of graph-processing systems (fundamental), and also we design and implement new partitioning policies and distributed heterogeneous graph-processing systems (applied). Our study combines application-knowledge-design research into a virtuous research cycle.

In the remainder of this chapter, we discuss the diversity of graph-processing applications in Section 1.1. We show examples of popular graph-processing systems, describe their programming models and features, and introduce the status of understanding their performance in Section 1.2. We discuss various graph-partitioning policies for graph-processing system in Section 1.3. We formulate the five main research questions of this thesis in Section 1.4. We summarize our contributions and introduce the outline of this thesis in Section 1.5.

1.1 Graph Processing Applications

To extract useful information from graphs, many graph-processing applications are executed in various fields and areas. A graph-processing application includes two core components, the *dataset* that is the input graph for processing and the graph-processing *algorithm* that expresses the analytical operations on the dataset. The former component is of particular interest for this thesis, as datasets are emerging in many fields that involve millions of participants, and can generate revenue both after being processed and raw.

We are experiencing a data deluge of large-scale graphs that are generated from hundreds of areas, from genomics to consumer profiles, from social networks to business decision support, with periodic updates and different data structures. For example, in the area of online gaming, there are thousands of successful games in the world, some of which involve tens of millions of active players [82, 99]. A huge amount of online gaming data is rapidly generated by players and recorded by gaming operators. Among these data, relationships between gaming entities, such as players and items, are a major part. These relationships can be presented as graphs, which consist of millions of vertices (gaming entities) and billions of edges (in-game relationship). The graphs contain important information for gaming operators to provide better service and create more revenue. Processing such graphs can also be important to other third-parties (non-game operators), with many different goals.

Due to the many different goals of processing graphs, a variety of graph algorithms have been developed. From the perspective of functionality, the algorithms can be categorized into several groups such as for calculating basic graph metrics [139], for traversing graphs [3, 144], for detecting communities [31, 85, 105], for searching for important vertices [104, 106], for sampling graphs [83], for predicting graph evolution [9, 84], etc. From the perspective of the behavior of graph-processing algorithms, most of the algorithms consist of a number of iterations, while only a few of the algorithms can be completed within a single step. The iterative algorithms can be further categorized into stationary and non-stationary by the size and variation of the sets of active vertices at each iteration [75]. In every iteration of stationary algorithms, all or a rather static set of vertices are active and they receive and generate about the same amount of messages. Typical stationary algorithms are PageRank [104], Connected Components [144], and Semi-clustering [92]. In contrast, only a (frequently changing) part of vertices are active in any given iteration of a non-stationary algorithm, and the number of messages varies across different iterations. This is true, for example, for Breadth First Search [18], Single Source Shortest Path [92], and Random Walk [125],

1.2 Graph Processing Systems

Tens of graph processing systems have been developed in the past decade, each one designed with specific requirements in mind. Among these requirements, support for using a single machine or multiple machines, and for using CPUs and/or GPUs, are often the most important. For example, Pregel [92], Giraph [43], and PGX.D [61] are distributed CPU-based systems that offer a simple, high-level programming model and focus on processing very large graphs with reasonable performance and very good scalability. Single-machine CPU-based systems, such as GraphChi [80] and GridGraph [153], load and process only a part of a large-scale graph at a time, and repeat this until the entire graph has been processed, to break the memory constraints of a single machine. However, for complex graph applications, they may have poor performance, due to slow disk access, but also due to the lack of computation capability. Graph databases, such as OrientDB [103], Sparksee [124], and Neo4j [101], have also been designed and developed to process graphs, but their ability to handle large-scale graphs is limited. Other systems, such as TOTEM [41], Medusa [152], Gunrock [137], focus on offering users efficient ways to accelerate their graph processing using *GPUs* on a *single* machine. MapGraph [39] can use only GPUs, but from *multiple* machines. Despite their ability to achieve high performance, these systems cannot handle large-scale graphs efficiently; for example, they cannot simultaneously use multiple CPU+GPU machines.

Graph-processing systems can also be distinguished by their programming models. Many graph-processing systems adopt the vertex-centric paradigm, in which graph-processing algorithms are implemented from the perspective of the operations conducted by each individual vertex. The Bulk Synchronous Parallel (BSP) computing model has been used by many graph-processing systems, such as Pregel [92] and Hama [114], mainly because the BSP model simplifies the design and implementation of iterative graph-processing algorithms. A BSP computation of a graph-processing algorithm consists of a series of global iterations (or supersteps). In each iteration, active vertices execute the same user-defined function, generate messages, and transfer them to neighbors that are not located in the same machine. Synchronization is needed between consecutive iterations to ensure that all vertices have been processed and all messages have been delivered. The synchronization in BSP systems may lead to significantly degraded performance, especially when the workload is not balanced across all the working machines. To improve performance, graph-processing systems such as GraphLab [87] and GraphHP [17] have used *asynchronous* models to avoid using barriers for synchronization and to reduce the performance degradation caused by imbalanced workload. The use of asynchronous models increases the complexity of graph-processing systems and, in some cases, creates redundant messages [151] when executing graph algorithms.

Graph-processing systems can also be categorized according to their use of computation phases, such as the *gather* phase of combining all relevant data [95]: *one-phase* [43, 92], *two-phase* [61, 109, 128], and *three-phase* [39, 45]. Four or more phase systems are theoretically possible, but we have not encountered them among the tens of graph-processing systems we have studied. In each of the three kinds of systems, the main computation tasks (processing incoming messages, applying vertex updates, and preparing outgoing messages) are placed and executed in different computation phases. For example, in *Scatter-Gather* systems, the scatter phase prepares outgoing messages, and the gather phase collects incoming messages and applies updates to vertex values. We will further analyze and discuss these three systems in Chapter 5.

Thoroughly understanding the performance of various graph-processing systems can help users select appropriate graph-processing systems for their applications, and can help developers design systems with better performance. However, few studies have been conducted on comprehensively evaluating the performance of graph-processing systems. Most performance studies were proposed by the system designers themselves to prove the superiority of their own systems, and may lack the method for performance study.

Previous performance studies lack representative workloads, performance metrics, and comparative systems [41, 70, 76, 112, 136].

1.3 Graph Partitioning Policies

Graph partitioning is essential to the performance of distributed graph-processing systems. Graph partitioning has been explored and studied for a long time in many research areas [73, 95], from scientific workflow scheduling [44] to recent work on large-scale graph processing [46]. Traditional heuristics, such as METIS [72] and its family of partitioning policies [32], aim to minimize the communication between partitions while balancing the number of vertices in each partition. METIS and its family are commonly used by the community because of their high-quality partitions. However, traditional heuristics are difficult to be applied to distributed graph-processing systems and time consuming for partitioning large-scale graphs.

Streaming graph-partitioning policies, which treat vertices as a stream and assign them one-by-one, have been proposed for distributed graph-processing systems. *Hash partitioning* is a typical type of streaming graph partitioning. Hash partitioning determines the partition of each vertex by using a hash of the vertex ID. Due to its simplicity and short partitioning time, hash partitioning is used by many graph-processing systems, such as Pregel-like systems [43, 92]. Hash partitioning has obvious drawbacks on partitioning large-scale graphs. For example, hash partitioning does not consider any locality of vertices and edges, which may incur a massive number of messages and intensive network traffic. Many complex streaming graph-partitioning policies have been designed to improve the performance of graph processing. For example, Stanton and Kliot [126] design more than ten streaming policies by considering many factors, such as the relationship between the vertex to be assigned and the current vertices in the partition and streaming orders.

Many traditional heuristics and streaming partitioning policies are edge-cut partitioning policies, by which vertices are assigned to different partitions. Thus, the edge of each pair of vertices in two different partitions is cut. In contrast, *vertex-cut* partitioning policies place edges, instead of vertices, in different partitions. Vertex-cut partitioning policies have been used in some graph-processing systems [45, 46]. By using vertex-cut partitioning, the intensive workload of high-degree vertices can be split to multiple working machines and balance the communication among partitions. However, vertex-cut partitioning cannot always achieve good performance, for example, too many pieces of vertex replicas can still generate high communication. Also, vertex-cut partitioning increases the complexity of graph-processing systems, which need to allow a single vertex's computation to span multiple machines.

1.4 Problem Statement

In this thesis, we address each of the three categories of research in graph-processing systems: application, knowledge, and design. Specifically, we address the following research questions.

RQ-1 (application): How to build a virtual meeting space for sharing, exchanging, and analyzing graphs? Currently, few graph repositories can be accessed publicly. Although tens of graph datasets from diverse sources are provided by repositories, they originate from relatively few application domains. One of the important emerging application domains for graph data is online gaming. Designing and building a data repository for sharing graphs collected from online games can provide an important data source of large-scale graphs, and can facilitate the comparability of studies.

RQ-2 (knowledge): How well do CPU-based graph-processing systems perform? Graph-processing systems are increasingly used in a variety of domains (Section 1.1). Although both industry and academia are developing and tuning graph-processing algorithms and systems, the performance of graph-processing systems has never been explored or compared in-depth. Currently, tens of graph-processing systems of different kinds have been designed and developed (Section 1.2). Thus, users face the daunting challenge of selecting an appropriate system for their specific applications. Most of the existing graph-processing systems are CPU-based, they can only use CPU(s) as the computation resource. Understanding the performance of CPU-based graph-processing systems can help developers design and tune graph-processing systems, and can help users make their selections.

RQ-3 (knowledge): How well do GPU-enabled graph-processing systems perform? Using the capability of GPUs to process graph applications is a new promising branch of graph-processing research (Section 1.2). Previous performance evaluation studies have been conducted for CPU-based graph processing systems. Unlike them, the performance of GPU-enabled systems is still not thoroughly evaluated and compared. The different programming models and various optimization strategies designed in GPU-enabled graph-processing systems raise the challenging question of understanding their performance. Evaluating the performance of GPU-enabled systems complements the performance study of CPU-based graph-processing systems (RQ-2) and can further help the design of graph-processing systems for system developers and the selection for system users.

RQ-4 (design): How to design low-overhead graph-partitioning policies for distributed graph-processing systems? Many distributed graph-processing systems have been designed and developed to analyze large-scale graphs (Section 1.2). For all distributed graph-processing systems, partitioning graphs is a key part of processing and an important aspect to achieve good processing performance (Section 1.3). To improve the performance of partitioning graphs, even when processing the ever-increasing modern graphs, many previous studies use lightweight streaming graph-partitioning policies. Although many such policies exist, currently there is no comprehensive study of their impact on load balancing and communication overheads, and on the overall performance of graph-processing systems. This relative lack of understanding hampers the development and tuning of new streaming policies, and could limit the entire research community to the existing classes of policies. A method to design graph-partitioning policies for real graph-processing systems is crucial.

RQ-5 (design): How to design a distributed *and* heterogeneous graph-processing system? To process graphs efficiently, GPU-enabled graph-processing systems such as TOTEM and Medusa exploit the GPU or the combined CPU+GPU capabilities of a single machine. Unlike scalable distributed CPU-based systems such as Pregel and GraphX, existing GPU-enabled systems are restricted to the resources of a single machine and to the limited amount of GPU memory, and thus cannot analyze the increasingly large-scale graphs we see in practice. A distributed and heterogeneous graph-processing system is needed to bridge the gap between existing distributed CPU-based systems and GPU-enabled systems for large-scale graph processing.

1.5 Main Contributions and Thesis Outline

This thesis consists of 7 chapters. The 5 research questions introduced in the previous section are addressed in Chapters 2 through 6, respectively. The structure of the thesis is depicted in Figure 1.2. The contributions of this thesis are listed as follows:

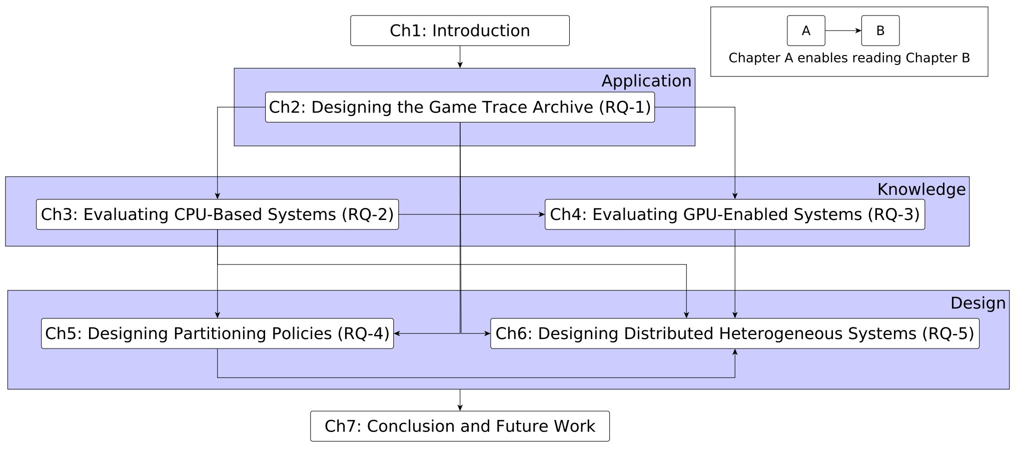

Ch1: Introduction
A
B
Chapter A enables reading Chapter B
Application
Ch2: Designing the Game Trace Archive (RQ-1)
Knowledge
Ch3: Evaluating CPU-Based Systems (RQ-2)
Ch4: Evaluating GPU-Enabled Systems (RQ-3)
Design
Ch5: Designing Partitioning Policies (RQ-4)
Ch6: Designing Distributed Heterogeneous Systems (RQ-5)
Ch7: Conclusion and Future Work

Figure 1.2: Structure of the thesis.

The Game Trace Archive (Chapter 2) To address research question RQ-1 about how to share graphs in the online gaming area, we build and maintain the Game Trace Archive

(GTA), which is a virtual meeting space for gaming-graph exchange and analysis. This chapter is based on the following work:

- **Yong Guo** and Alexandru Iosup, "The Game Trace Archive", *Annual Workshop on Network and Systems Support for Games (NetGames)*, 2012.
- **Yong Guo**, Siqi Shen, Otto Visser, and Alexandru Iosup, "An Analysis of Online Match-Based Games", *International Workshop on Massively Multiuser Virtual Environments (MMVE)*, 2012.

Evaluating the Performance of CPU-Based Graph-Processing Systems (Chapter 3) To address research question RQ-2 about how well CPU-based graph-processing systems perform, we propose an empirical method to address the challenges of benchmarking graph-processing systems, and we use this method to evaluate six CPU-based graph-processing systems. This work has been extended to Graphalytics, which is a developing project for benchmarking graph-processing systems in which researchers from both academia (TU Delft, VU Amsterdam, UvA Amsterdam, UPC Barcelona, and Georgia Tech) and industry (Oracle Labs, Intel Labs, IBM, and Neo4j) are involved. This chapter is based on the following work:

- **Yong Guo**, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke, "Benchmarking Graph-Processing Platforms: A Vision", *ACM/SPEC international conference on Performance engineering (ICPE)*, 2014.
- **Yong Guo**, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke, "How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis", *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2014.

Evaluating the Performance of GPU-Enabled Graph-Processing Systems (Chapter 4) To address research question RQ-3 about how well GPU-enabled graph-processing systems perform, we adapt and extend our empirical method of Chapter 3, and we conduct a comparative performance study of three GPU-enabled graph-processing systems. This chapter is based on the following work:

- **Yong Guo**, Ana Lucia Varbanescu, Alexandru Iosup, and Dick Epema, "An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

Designing Streaming Graph-Partitioning Policies (Chapter 5) To address research question RQ-4 about how to design low-overhead graph-partitioning policies for distributed graph-processing systems, we model the run time of different types of graph-processing systems, propose a method to identify the run-time-influencing graph characteristics, and design new streaming partitioning policies to minimize the run time of real world graph-processing systems. This chapter is based on the following work:

- **Yong Guo**, Sungpack Hong, Hassan Chafi, Alexandru Iosup, and Dick Epema, "Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies", *Journal of Parallel and Distributed Computing (JPDC)*, http://dx.doi.org/10.1016/j.jpdc.2016.02.003, 2016.

Designing Distributed Heterogeneous Graph-Processing Systems (Chapter 6) To address research question RQ-5 about how to design a distributed and heterogeneous graph processing system, we explore the design space of distributed heterogeneous graph-processing systems and implement three families of such systems that can use both the CPUs and GPUs of multiple machines. This chapter is based on the following work:

- **Yong Guo**, Ana Lucia Varbanescu, Dick Epema, and Alexandru Iosup "Design and Experimental Evaluation of Distributed Heterogeneous Graph-Processing Systems", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

In Chapter 7, we summarize the key findings of this thesis and discuss future research directions.

Chapter 2

Designing the Game Trace Archive: More Graph Datasets

In this chapter, we design the Game Trace Archive. We propose the Game Trace Format to facilitate the exchange and ease the usage of game traces. It is a unified format to cover many different types of game traces and to include complex and detailed gaming information in each trace. We observe that the relationships between game entities (such as player, group, item, etc.) are a major part of the online-gaming data. We model the relationships and game entities as the edges and vertices of graphs, respectively. In the Game Trace Format, we design a specific part to store different types of relationships in various online-games. GTA currently offers about 10 traces in a unified format and is much used by the gaming community and graph analysts. By using traces from GTA, we have also conducted an analysis of online match-based games in several aspects, including the influence of the friendship of players in gaming experience.

2.1 Overview

Over the last decade, video and computer gaming have become increasingly popular branches of the entertainment industry. Understanding the characteristics of games and OMGNs such as player behavior [129], [26], traffic analysis [16], [77], resource management [13], [23], etc., is essential for the design, operation, and tuning of game systems. However, only a small number of game traces exist; even fewer can be accessed publicly. As a consequence, previous studies used at most a few traces and no comprehensive comparative analysis exists. Moreover, the few available game traces have diverse formats, which makes it difficult to exchange and use these game traces among game researchers. To address this situation, in this chapter we propose the Game Trace Archive.

There are thousands of successful games in the world, which attract a large number of players. For example, World of Warcraft, one of the most popular Massively Multiplayer Online Games (MMOGs), and CityVille, a widely spread Facebook game, have each tens of millions of players. A number of online communities have been constructed by game operators and third-parties around single games or even entire collections of games. These communities through the relationships between entities such as players and games, form Online Meta-Gaming Networks [119]. Tens of millions of players currently participate in OMGNs, such as Valve's Steam, XFire, and Sony's PlayStation Network, to obtain game-related information (game tutorials, statistic information, etc.) and use non-gaming functionalities (voice chat, user product sharing, etc.).

We design in this chapter the GTA as a virtual meeting space for the game community, in particular to exchange and use game traces. Game traces, which can be collected at one timepoint, several or series of timepoints, or through a continuous period of observation, contain many types of game-related data about both games and OMGNs. With the GTA, we propose a unified Game Trace Format (GTF) to store game traces, and provide a number of tools to analyze and process game traces in GTF format. Our goal is to make both the game traces and the tools in the GTA publicly available. Mainly because of the diversity and big size of game traces, there are three main challenges in building the GTA. Firstly, game traces can be collected from many sources. They can focus on any of the multiple levels of the operation of gaming systems, from OMGNs to single game traces, from players to player relationships, and to packets transferred between the players and the game servers. The trace content at different levels is significantly different; moreover, even at the same level the traces may be very different. Secondly, the content of each trace can be complex. Traces may include many kinds of relationships between the game entities (players, guilds, etc.) and detailed information of entities (player name, player date of birth, in-game and meta-game information, etc.). Thirdly, it is difficult to process the large-scale and complex game traces, and to choose from tens of or hundreds of game traces depending on specific scenarios.

This chapter is further motivated by our ongoing project @larGe, which aims at designing new systems that support gaming at large-scale. Trace archives support several other computing areas including the Parallel Workloads Archive (PWA) [132] for the parallel systems, the Grid Workloads Archive (GWA) [67] for the grid systems, etc. However, non of these previous archives can include the complex game traces we target with the GTA. **Our research is the first work in establishing a comprehensive Game Trace Archive to benefit gaming researchers and practitioners.** The main content of this chapter is structured as follows:

1. We synthesize the requirements for building an archive in the gaming domain (Section 2.2).

2. We design the Game Trace Archive, including a unified format and a toolbox to collect and share game and OMGN traces (Section 2.3).

3. We conduct a comparative analysis using many real game traces (Section 2.4).

2.2 Requirements for a Game Trace Archive

Starting from the three main challenges we mentioned in the introduction, in this section we identify five main requirements to build an archive for game traces. Although these requirements are similar to those of archives in other areas, our ambitious goal of building a single format for *all* game data makes it challenging to achieve these requirements.

Requirement 1: Trace collecting. To improve game trace readability and facilitate game trace exchange, a unified game trace format should be provided to collect game traces from diverse sources. Firstly, the game trace format must be able to include many types of game information. Secondly, formats already exist for specific game information; for example, packets sent between game servers and clients. The unified game trace format needs to use these other formats. Finally, due to the rapid evolution of the gaming industry, many new games and game traces may emerge. The format should be extensible to collect traces of future games, while not affecting the usage of old game traces that have already been stored in the archive.

Requirement 2: Trace converting and anonymizing. The raw content of game traces is complex. However, a small part of the content may not be game-related, because of where the game traces are collected. For example, when crawling OMGN traces from their websites, advertisement information in each website may also be collected into the trace. Thus, during the procedure of converting game traces to the unified game trace format, this useless content should be filtered out. Another important issue in converting traces is to provide a privacy guarantee. Many studies [100], [111] have shown that just anonymizing user names is not sufficient to ensure privacy.

Requirement 3: Trace processing. The archive should provide a toolbox to process the converted game traces and generate reports including commonly used characteristics of game traces (e.g., trace size, the number of information items). Furthermore, the toolbox could also be used by archive users to build comprehensive analysis tools.

Requirement 4: Trace sharing. The game traces and the trace processing toolbox must be shared publicly. The archive users may face the problem of selecting proper traces according to their own requirements, especially when the archive expands to store tens of or hundreds of traces. A trace-selection mechanism is needed to address this problem. Allowing archive users to rank and comment on the traces would be a good component in the selection mechanism.

Requirement 5: Community building. The main goal of building a game trace

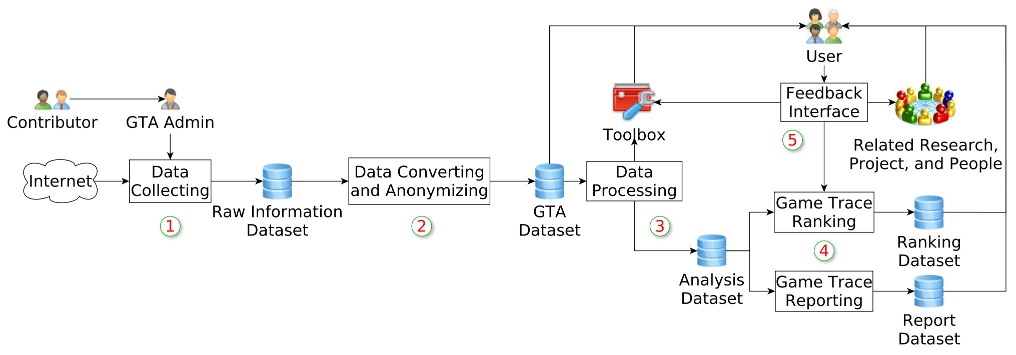


Figure 2.1: The design of the Game Trace Archive.

archive is to establish a virtual meeting space for game researchers. Besides the exchange of game traces, a list of research, projects, people, and applications of the archive should be maintained to facilitate further communication of game community.

2.3 The Game Trace Archive

In this section, we introduce the Game Trace Archive. We first discuss the design of the GTA and then describe the design of the Game Trace Format (GTF).

2.3.1 The Design of the Game Trace Archive

Figure 2.1 illustrates the overall design of the Game Trace Archive, including the GTA members and how the game traces been processed in the GTA. The circles in Figure 2.1 represent the five requirements we formulated in Section 2.2.

We envision three main roles for the GTA members. The *contributor*, who is the legal owner of game traces, offers their traces to the GTA and allows public access to the traces. The *GTA Admin* helps contributors to add and convert game traces to the GTA, and manage traces processing and sharing. Our game research team may act as the GTA Admin. The *user* accesses the archived game traces, uses the processing toolbox, and obtains the relevant research information through the GTA. Most users may be game researchers and practitioners, but we believe that people in other areas (e.g., biologists, economists, social network researchers) can also benefit from the GTA.

In the design of the GTA, the *Data Collecting* module is for collecting game traces from multiple sources: contributors, publicly shared game data repositories, game websites, etc. These game traces have their own formats and some of them may include sensitive information. Firstly, we store these traces in a raw information dataset without additional processing. Then, we map the raw content to the unified GTF (for requirement

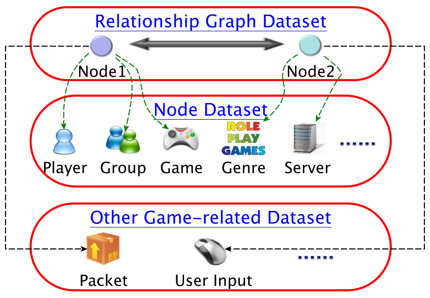


Figure 2.2: The structure of the Game Trace Format.

1).

The *Data Converting and Anonymizing* module is responsible for converting the game traces from their own formats to the unified GTF, while anonymizing the sensitive information (for requirement 2). The anonymization process, which is a topic of research in itself [100], [111], is outside the scope of this work. The map from the original information and the anonymized information will not be distributed, only the corresponding trace contributor has the authority to read it.

In the *Data Processing* module, a toolbox is provided for comprehensive trace analysis: overview information, such as trace size, period, the number of relationships, and the number of players; in-game characteristics, such as active playing time, average session duration, and the number of played games; relationship graph metrics, such as diameter, link diversity, clustering coefficient. The basic tools in the toolbox can be used to build other processing tools and can also be used to process large scale graphs in other areas, such as social network and viral marketing (for requirement 3).

Two modules are designed for trace sharing (for requirement 4). The *Game Trace Reporting* module receives the trace analysis results from Data Processing module and formulate them into more visible reports. The *Game Trace Ranking* module considers both the overview information from Data Processing module and feedback from trace users to rank the game traces. The Game Trace Reporting and Ranking modules help the GTA users to select a game trace based on a quick knowledge of the basic game trace characteristics.

The *Feedback Interface* is for supporting trace sharing and community building (for requirements 4 and 5). It is the interface for users to submit their feedback after using game traces in the archive. There are four types of feedback: rank of traces, comment on traces, updated or newly designed tools for traces, and research information (e.g., research direction, project, applications for traces). For each game trace in the GTA, we maintain a list of research information derived from feedback. Through these lists, users can know the game community better and users in similar research directions may establish further communication.

Table 2.1: Format for basic edge information.

ID	Column	Description
1	RowID	*Int*. A count field. The lines in this file should be sorted by ascending RowID.
2	SrcType	*Int*. The type of *source node*, including player, community, team, faction, guild, etc. Each node type has a unique integer number assigned to it, the correspondence of type and number can be found in meta-information *Format for node type*. Source node: the starting node of directed edge. For undirected edge, it is the node appearing first in the raw data of the edge.
3	SrcID	*Int*. ID of source node. To protect the source node privacy, their names are anonymized by assigning each unique source node a unique ID.
4	DstType	*Int*. The type of *destination node*, including player, community, team, faction, guild, etc. Each node type has a unique integer number assigned to it, the correspondence of type and number can be found in meta-information *Format for node type*. Destination node: the ending node of directed edge. For undirected edge, it is the node appearing last in the raw data of the edge.
5	DstID	*Int*. ID of destionation node. To protect the destination node privacy, their names are anonymized by assigning each unique destination node a unique ID.
6	EdgeType	*Int*. The type of edge between source node and destination node, indicating the relationship between source node and destination node. Each type is represented by a unique integer number, the correspondence of type and number can be found in meta-information *Format for edge type*.

2.3.2 The Design of the Game Trace Format

We propose the Game Trace Format to facilitate the exchange and ease the usage of game traces. It is a unified format to cover many different types of game traces and to include complex and detailed gaming information in each trace. In this subsection, we introduce the main elements of the design of the GTF.

Figure 2.2 shows the structure of the Game Trace Format. Briefly, the GTF consists of three datasets: the *Relationship Graph Dataset*, the *Node Dataset*, and the *Other Game-related Dataset*. These datasets are responsible for storing different kinds of content in game traces respectively.

Relationship Graph Dataset

From our observation, in many game traces the relationships (e.g., play with, send message to, member of) between many kinds of game entities (e.g., player, group, game in OMGN, genre in OMGN) are significant for the operation of these games and OMGNs. Thus, we model these relationships as *edges* in a *graph* where *nodes* are game entities. We include in the GTF the Relationship Graph Dataset to include the relationships presented in game traces. The Relationship Graph Dataset has three sub-datasets: with *basic edge*

Table 2.2: Format for the fixed part in detailed edge information.

ID	Column	Description
1	RowID	*Int*. Same to RowID in the *basic edge information*.
2	TimeStamp	*UnixTimeStamp*, in millisecond. The beginning time of the recorded event. Can be Null.
3	EdgeLifetime	*Float*, in millisecond. Duration of the edge exsiting. Can be Null.
4	SrcScore	*String*. Source node payoff. The meaning of SrcScore can be found in meta-information *Format for edge type*. Can be Null.
5	DstScore	*String*. Destination node payoff. The meaning of DstScore can be found in meta-information *Format for edge type*. Can be Null.
6	ExtEdgePath	*String*. The path set to access the *extended part* of detailed edge information.

Table 2.3: Format for edge type in meta-information.

ID	Column	Description
1	EdgeType	*Int*. The integer number assigned to edge type. The lines in this file should be sorted by ascending EdgeType.
2	EdgeDirectivity	*String*. Directed or Undirected edge.
3	EdgeTypeTerm	*String*. A short term describing edge type.
4	EdgeTypeDef	*String*. A detailed definition or description of edge type.
5	SrcScoreDef	*String*. The meaning of *SrcScore*.
6	DstScoreDef	*String*. The meaning of *DstScore*.

information, with *detailed edge information*, and with *meta-information*.

The basic edge information (Table 2.1) includes the essential or must-have elements for relationships (e.g., edge/node type, edge/node identifier). By using different edge types and node types, various relationships in game traces can be presented in our format.

To store other diverse edge-related information, we use the detailed edge information sub-dataset, which includes two parts, the *fixed part* and the *extended part*. The fixed part (Table 2.2) stores typical attributes of edges (e.g., timestamp, edgelifetime). The extended part stores extended edge attributes that are not common for all relationships. Since these extended edge attributes differ per trace, there is no exact format for the extended part. For each attribute, we use one column to store its value. The design of extended part makes it possible to cover new types of edge-related information.

The meta-information includes the overview of the Relationship Graph Dataset and the definitions of all the edge (Table 2.3) and node types (Table 2.4).

Table 2.4: Format for node type in meta-information.

ID	Column	Description
1	NodeType	*Int*. The integer number assigned to node type. The lines in this file should be sorted by ascending NodeType.
2	NodeTypeTerm	*String*. A short term describing node type.
3	NodeTypeDef	*String*. A detailed definition or description of node type.

Table 2.5: Format for the fixed static part in Node Dataset.

ID	Column	Description
1	RowID	*Int*. A count field. The lines in this file should be sorted by ascending RowID.
2	NodeID	*Int*. The integer number assigned to node.
3	TypStatic1	*String*. Typical static attribute of node.
4	TypStatic2	*String*. Another typical static attribute of node.
...	...	*String*. Another typical static attribute of node.
N+2	TypStaticN	*String*. The Nth typical static attribute of node.
N+3	ExtStaPath	*String*. The path set to access the *extended static part*.
N+4	TypDynPath	*String*. The path set to access the *fixed dynamic part*.
N+5	ExtDynPath	*String*. The path set to access the *extended dynamic part*.

Node Dataset

The Node Dataset is designed to include detailed node information. Since the information of different types of nodes can be diverse, to store this information in a unified format, we categorize the information first.

We divide the complex node information into *static* (keep constant with time, e.g., player gender, date of birth) and *dynamic information* (change with time, e.g., player rank, level).Each type of node has its own *node sub-dataset*. For each type of node, we further divide their static and dynamic information into *typical static information*, *typical dynamic information*, *extended static information*, and *extended dynamic information*.

These four kinds of information are stored in *fixed static part* (Table 2.5), *fixed dynamic part* (Table 2.6), *extended static part*, and *extended dynamic part*, respectively in the node sub-dataset. We use the same method (as we do for the extended part in detailed edge information, Section 2.3.2) to store extended static and dynamic attributes. Through its design, specifically through its extended static and extended dynamic parts, our Node Dataset is the first to store the information of many kinds of nodes in a unified format.

Table 2.6: Format for the fixed dynamic part in Node Dataset.

ID	Column	Description
1	RowID	*Int*. A count field. The lines in this file should be sorted by ascending RowID.
2	NodeID	*Int*. The integer number assigned to node.
3	TimeStamp	*UnixTimeStamp*, in millisecond. The timestamp for dynamic attribute value.
4	TypDyn1	*String*. Typical dynamic attribute of node.
5	TypDyn2	*String*. Another typical dynamic attribute of node.
...	...	*String*. Another typical dynamic attribute of node.
N+3	TypDynN	*String*. The Nth typical dynamic attribute of node.

Other Game-Related Dataset

For more traditional gaming data, such as packets between game clients and servers, match replays, player click streams, etc., we use when possible the de-facto standard formats and store the data in the Other Game-related Dataset. For example, for match replays, we keep their own formats derived from games. The formatted replays, such as StarCraft replays, are used by Weber et al. [140] and Hsieh et al. [63] to study player in-game behavior.

Moreover, we provide detailed introduction files to guide the archive users how to process the formats. Meta-information is also provided, to link the game-related information to its corresponding nodes; for example, the links between players and the packets they have sent.

2.4 Analysis of Traces from the Game Trace Archive

In this section, we present and analyze the game traces collected by the GTA. Using our GTA toolbox, we conduct an analysis that focus on four main aspects.

Table 2.7 summarizes the nine game traces currently formatted in the GTF. These traces have been collected from five types of games or OMGNs, including board games, card games, RTS games, MMORPG games, and OMGNs. Our GTA can cover game traces collected by ourselves. The KGS and FICS traces include a large number of matches in two popular board games, Go, and chess. The BBO trace is collected from

[1]http://www.gokgs.com/
[2]http://www.freechess.org/
[3]http://www.bridgebase.com/
[4]http://beta.xfire.com/
[5]http://www.dota-league.com/
[6]http://www.dotalicious-gaming.com/
[7]http://replay.garena.com

Table 2.7: Summary of game traces in the GTA.

Trace	**Period**	**Size (GB)**	**# Nodes (K)**	**# Links (M)**	**Genre**
KGS[1]	2000/02-2009/03	2	832	27.4	board
FICS[2]	1997/11-2011/09	62	362	142.6	board
BBO[3]	2009/11-2009/12	2	206	13.9	card
XFire[4]	2008/05-2011/12	58	7,734	34.7	OMGN
Dota-League[5]	2006/07-2011/03	23	61	3.7	RTS
DotAlicious[6]	2010/04-2012/02	1	64	0.6	RTS
Dota Garena[7]	2009/09-2010/05	5.2	310	0.1	RTS
WoWAH [82]	2006/01-2009/10	1	91	N/A	MMORPG
RS [99]	2007/08-2008/02	1	0.1	N/A	MMORPG

one of the largest bridge sites in the world, where people can play bridge online for free. The dataset of the OMGN XFire contains the information of thousands of games and millions of players, as well as complex relationships between those games and players. Defense of the Ancients (DotA) is mod build on the RTS game Warcraft III. A match of DotA is played by two competing teams, allowing at most 5 players in one team. The main content in the Dota-League, DotAlicious, and DotA Garena traces is plenty of DotA matches played on three different DotA playing platforms. We are also able to include in the GTA existing game traces. WoWAH is a public dataset consists of online session records and attributes of World of Warcraft avatars collected from game servers in Taiwan. RS collects the online player counts of more than 100 servers of the MMORPG game RuneScape. The size in Table 2.7 is the raw data size of each trace. The node may be player, avatar, or game server, corresponding to each trace. In XFire, we use the link as friendship. For the other traces, we define the link as playing in a same match. With these game traces, we can conduct a comparative analysis of games, inter- and intra-genre.

2.4.1 Analysis of Workload Characteristics

Provisioning resources is an essential task for online gaming operators. Using the least possible resources to support the workloads generated by players brings maximum profits. However, inadequate provisioning may result in idle resources or the departure of players. In this subsection, we discuss five characteristics of the workloads of online games.

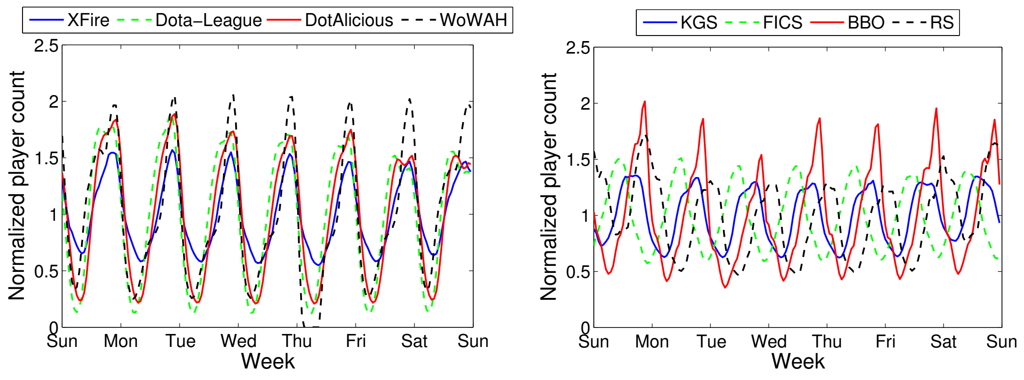


Figure 2.3: Weekly pattern of normalized online player count.

Weekly Pattern of Online Player Count

The online player count is an important metric of the usage and workload of game severs. In this subsection, we study how the online player count fluctuates during the week.

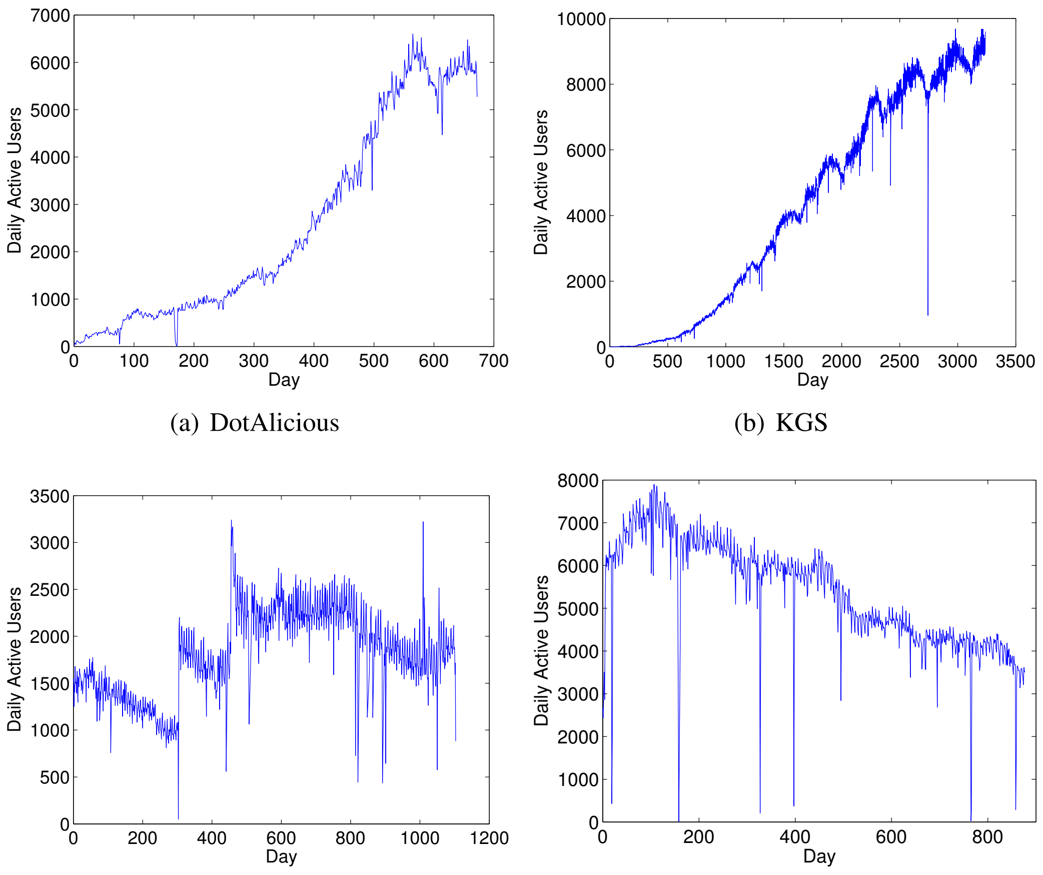
We define the *normalized online player count* at any moment of time, as the ratio between the player count at that moment and the trace-long average player count. Figure 2.3 illustrates the normalized online player count for eight game traces. The ninth trace (Dota Garena) currently included in the GTA does not have player count information over time. The figure shows that, for each game trace, the online player counts have obvious diurnal patterns; the peak and bottom counts occur at nearly the same time for each day. However, the occurring time of both peak and bottom differ per game. For example, for chess the peak occurs during the day, whereas midnight is the busiest period for all DotA platforms. Un-expectedly and unlike the workloads of web servers, for our traces the player counts do not differ significantly between week days and week ends. Moreover, for DotAlicious the player presence is even lower during week ends. Due to the scheduled weekly maintenance, there is an outage period on Thursday morning in WoWAH; the normalized player count drop to 0, see Figure 2.3 (left).

Lee et al. [82] investigate the daily, weekly, and monthly patterns of avatar counts in World of Warcraft. Chambers et al. [13] study the online player count over a 4-week period in FPS, casual, and MMORPG games. Both of their results show a similar weekly pattern as our result, but for fewer games.

Daily Active Users

Daily Active Users (DAU) is another important metric of the workloads of online games. On understanding the evolution of DAU, game operators can deploy game resources better in long term.

Figure 2.4 shows the evolution of DAU for four game traces: DotAlicious, KGS,



(a) DotAlicious (b) KGS

(c) WoWAH (d) Dota-League

Figure 2.4: Daily Active Users.

WoWAH, and Dota-League. We collected the datasets of DotAlicious and KGS from their launch, but the WoWAH and Dota-League datasets start at the date later than their setup. Both DotAlicious and KGS attracted more and more player significantly from their establishment. By contrast, the DAU of Dota-League dropped gradually until the end of the trace, which might be one of the reasons why the Dota-League platform shut down in November, 2011. In WoWAH, the release of new game contents or expansions can result in surges of the DAU. For example, on the day of around 450, the beginning of April, 2007, when an important expansion of WoW - The Burning Crusade was released(in Taiwan). However, the previous study [33] on another MMORPG EVE Online shows that updates slightly impact player growth.

Match Count per Player

Figure 2.5 depicts the match count (total number of matches played) per player of Dota-League, KGS, and FICS. The Cumulative Distribution Function (CDF) is depicted against

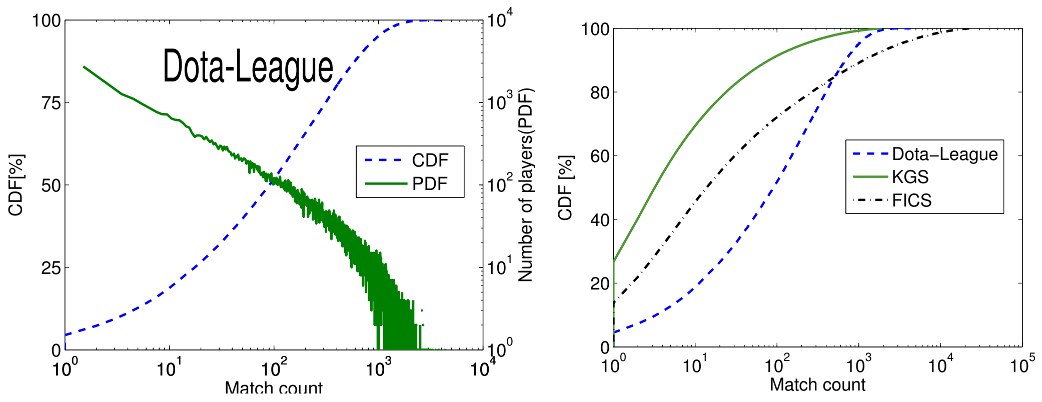


Figure 2.5: Match count per player for Dota-League, KGS, and FICS.

the left vertical axis. The Probability Distribution Function (PDF) is depicted against the right axis and only for the Dota-League dataset in Figure 2.5 (left). The right vertical axis and the horizontal axis are log-scale. Since there is a number of computer players (bots) existing in KGS and FICS servers, we filter out the top 0.5% "players" in terms of their match count, assuming these are all bots.

Although the number of board game players is larger, a significant portion of them play only a few matches: about 25% of KGS players and about 15% of FICS players participated in only one match. However, this value is much lower for Dota-League (3%). The match count per player of match-based games is heterogeneous. The median values of the match count in Dota-League, KGS, and FICS are 91, 4, and 15, while the 99.5% quartiles are 1,945, 1,908, and 23,396, respectively. Nearly half of the online board game players participate in less than 15 matches.

The match count per player follows a long tail distribution, where the maximum value can be over a hundred times larger than the median value. We fit the match count per player against the power-law, log-normal, weibull, exponential, normal, and gamma distributions using the maximum likelihood estimation technique. The best-fitting distribution has the smallest Akaike information criterion with correction (AICc) [8]. The match count per player can be best fitted, for KGS and FICS, using the power-law distribution. For Dota-League, the log-normal distribution is the best fit.

Inter-Arrival Time Distribution

We define the (match) inter-arrival time as the duration between the start times of two consecutive matches of a player. The inter-arrival times of a player represent his frequency of playing matches. Figure 2.6 shows the CDF of inter-arrival times across all the players of DotAlicious, KGS, and FICS. Over 80% of the inter-arrival times are less than one day and over 95% of the inter-arrival times are less than one week in these datasets. This

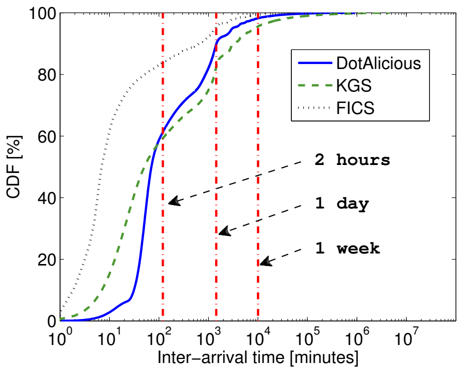
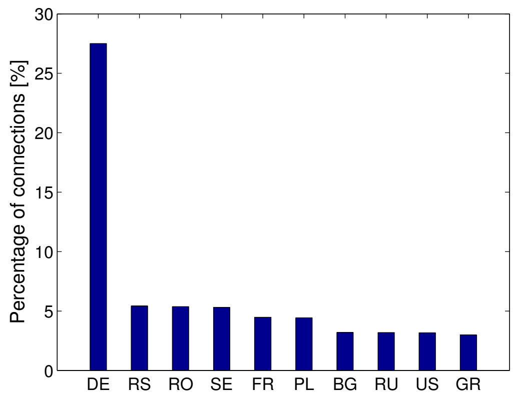
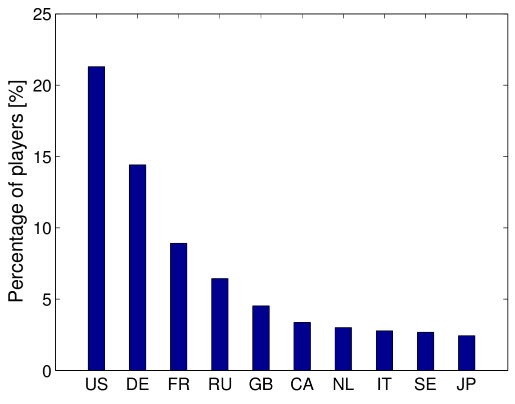


Figure 2.6: CDF of inter-arrival time.



(a) DotAlicious



(b) DGS

Figure 2.7: Geographic distribution of connections and players.

indicates that a large percentage of players comes back to play shortly after their last match; this is similar to MMORPG EVE Online [13]. The distribution of the inter-arrival times peaks at 47 minutes. Given the average duration of DotA matches (41 minutes), it means players tend to play two consecutive DotA matches. The inter-arrival time of DotA matches is longer than for World of Warcraft (median 20 minutes) [154]. As players are very likely to be continuously playing in successive matches, it could be beneficial to build a highly-efficient P2P-based MMVE using the DotA players' own computers as servers.

Geographic Distribution of Connections and Players

International use can be a metric to measure the success of online games. Many of the popular games, such as World of Warcraft, Starcraft, etc., are catering to subscribers from all over the world. One of the problems in serving players from different countries is how to deploy geographically distributed game servers while keeping a reasonable quality of experience for all players (see [99] and references within). Investigating the geographic distribution of game workloads can support addressing this problem.

For each match in DotAlicious, the countries where the players connect from are recorded. We count connections from each country, over all matches. Figure 2.7 shows the geographic distribution of connections in DotAlicious and that of the players of DGS[1]. The workloads of games are not equally distributed, since the top 10 countries account for a majority of the connections or the players. For DotAlicious, probably because nearly half of the European game servers are located in Germany, there are significantly more connections from Germany than from other countries. For both of these games, the top 10 countries are located in North America and Europe, which should therefore be key areas in resource deployment. For comparison, Feng et al. [34] analyzed the distribution of online FPS players, and also found that most players are located in North America, Europe, and Asia.

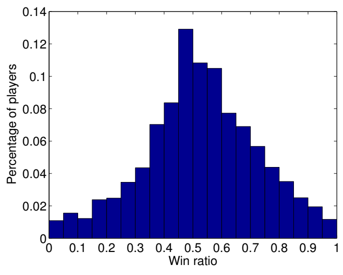
2.4.2 Analysis of Win Ratio

A big skill gap between players in a match can result in a disappointing experience. The more skillful players may lack challenge; the less skillful players may give up. Most of the match-based games have implemented a rating system to help players recognize their skill level and find proper opponents. The quality of rating system affects an elementary metric for match-based game players, the win ratio, defined as the percentage of wins of the total of wins and losses. In this subsection, we study the distribution of win ratios and the correlations between winning and several other characteristics, including the amount of played matches, friendship, and current rating systems.

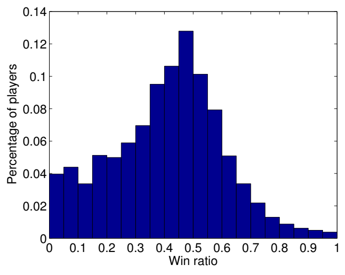
The Distribution of Win Ratios

First of all, we investigate the overview of win ratios of players. Figure 2.8 depicts the distributions of the win ratios for two board game traces and three DotA traces. To improve the accuracy of the analysis, players who play less than 10 matches are filtered out. The win ratio distributions of KGS, DotAlicious, and Dota-League almost follow the Gaussian distribution, with the average win ratios shift around 0.5. There are more players whose win ratios are less than 0.5 in FICS, which indicates chess beginners may find proper opponents easier. For Dota Garena, larger fraction of players own higher win ratios (more than 0.5) compared with that of the other two DotA platforms. The reason might be that matches collected from Dota Garena are uploaded by players, who may tend to show their victories.

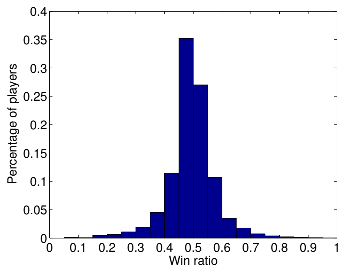
[1]Data from http://www.dragongoserver.net/statistics.php?stats=1, 2012-06-27



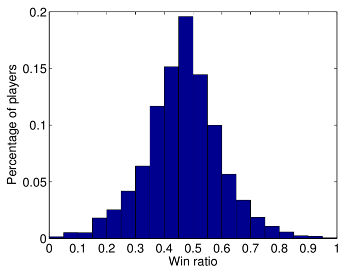
(a) KGS



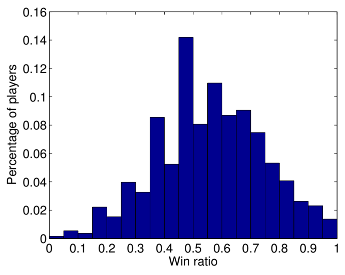
(b) FICS



(c) Dota-League



(d) DotAlicious



(e) DotA Garena

Figure 2.8: The distribution of player win ratios.

Win Ratio vs. Match Count

Intuitively, playing more matches should lead to better gaming skills and thus higher win ratios. Figure 2.9 shows the average win ratio versus the match count for Dota-League, KGS, and FICS. We normalize the match count based on the maximum values observed for each game. We then slice the normalized match count range into 100 bins and calculate the average win ratio for each bin.

For beginning players (range [0.0-0.1] in the horizontal axis), the evolution trends of the win ratios of KGS and FICS are opposite. The reason may be that, when registering in these games, players are suggested to fill in their skill levels. However, the default value of KGS is low, while that of FICS is a median value. Thus, beginners in KGS whose actual skill level may be higher, will play with less skillful players and gain a higher win ratio, and vice-versa for FICS. Beyond this starting zone, the win ratio fluctuates around 0.5. Thus, there is no direct correlation between win ratio and match count (with the coefficient of determination $R^2 = 0.1108$, p-value $P = 0.3342$). For advanced players (range [0.7-1.0]), the fluctuation is larger, which might be caused by the different types of players. People with longer player lifetime may be professional players or hardcore players with varying skill levels.

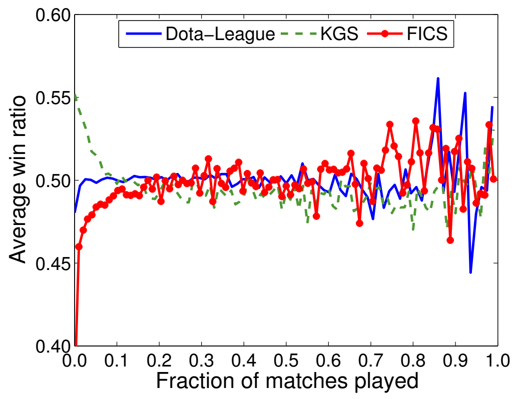


Figure 2.9: Average win ratio over fraction of matches played.

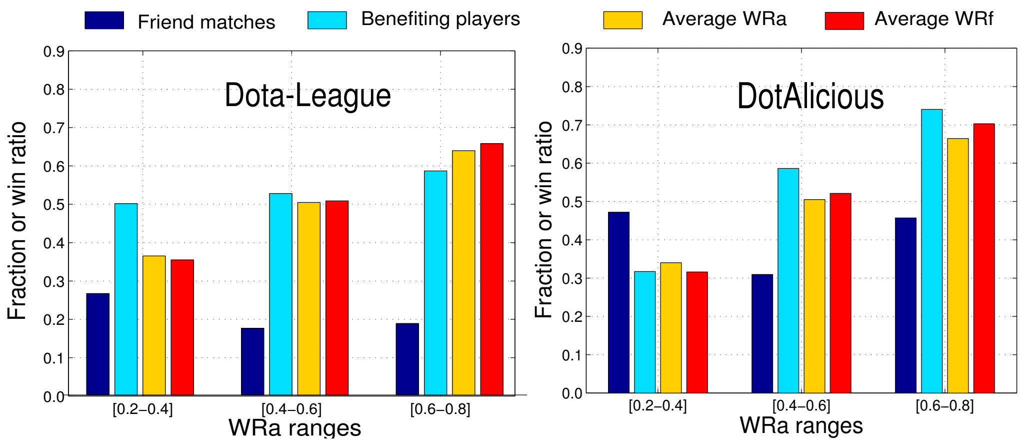


Figure 2.10: Performance of players for different win ratio ranges.

Win Ratio vs. Friendship

In many types of competitions, team-spirit and cooperation with friends have great effect on the success. In the gaming field, we can also find friendship and cooperation between players in many different types of games, for example in online bridge [5]. In this subsection, we analyze the impact of friendship on win ratios in DotA.

We find two kinds of relationships between players in Dota-League and DotAlicious. In Dota-League, players have a friend list called "buddy link", whereas in DotAlicious players can be a member of a clan with maximum 8 members. We consider both the buddy link and the clan membership as friendship. Figure 2.10 illustrates the performance of players, for 3 equally sized win ratio ranges. In this figure, friend matches refer to the matches players play with their friends in the same team. WRa is the win ratio of all matches; WRf is the win ratio of matches played with a friend. Benefiting players are players whose WRf is higher than their WRa. Since the fraction of players whose WRa is less than 0.2 or more than 0.8 is very small, we eliminate these players as outliers and focus on the WRa range from 0.2 to 0.8. The vertical axis is used to represent the fraction of friend matches, the fraction of benefiting players, average WRa and average WRf.

According to the "Friend matches" bars, more than half of the players play individually; also, players in Dota-League play less games with friends than players in DotA-licious. The reason is probably that the Dota-League matchmaking assigns players randomly, without guarantee to be in the same team with a friend. With the increase of win ratio, more players perform better in matches with their friends and improve their win ratios ("Benefiting players" bars). On average (bars "Average WRa" and "Average WRf"), the players with higher WRa (ranges [0.4-0.6] and [0.6-0.8]) win more matches when they play with friends in a team. However, the increase of the average win ratio is small (under 0.05). Surprisingly, the players with lower WRa (range [0.2-0.4]) lose more matches when they cooperate with friends. To some extent, it implies that DotA is not a beginner-friendly game: beginners can't help each other to higher win ratios. For comparison, Ducheneaut et al. [27] found that cooperation helps players to level up in World of Warcraft; Mason and Clauset [94] found that in Halo: Reach, teams composed of friends, on average, win more games than teams composed of strangers.

Winning Prediction of Current Rating Systems

As we mentioned at the beginning of this section, game operators may design or implement their own rating systems. In this subsection, we discuss the existing rating systems in DGS and FICS, and analyze the winning probability (WP) of a player in a match according to the skill level given by the rating systems. The DGS and FICS servers implement the EGF[1] and Glicko[2] systems, which are both based on the Elo[3] rating system, to measure the skill of players, respectively.

We define the winning probability for a specific skill gap as the fraction, from the matches between players whose skill rating differs by the gap, of the matches where the winner is the player with higher skill rating, based on the rating before the match. In general, if the skill gap is less than 100 in DGS or less than 200 in FICS, the skill levels of the two players are very similar. We logarithmically assign the skill gap value into 10 bins based on the maximum skill gaps of DGS and FICS (3,299 and 2,487, respectively). Table 2.8 presents the change of winning probability from the lowest to the highest skill gap. After filtering out abnormal matches (such as draws, matches with handicap[4] in DGS, etc.), we obtain 40,369 cleaned DGS matches and 136,104,236 FICS matches.

When the skill gap is small (under 100), the winning probability of the higher skill player is around the expected value of 0.5. As the skill gap increases, the higher skill

[1]http://senseis.xmp.net/?FIDETitlesAndEGFGoRatings
[2]http://senseis.xmp.net/?GlickoRating
[3]http://senseis.xmp.net/?EloRating
[4]http://senseis.xmp.net/?Handicap

Table 2.8: The winning probability by skill gaps.

DGS				FICS			
Gap	# Matches	%	WP	Gap	# Matches	%	WP
2	389	1.0	0.499	2	1,675,560	1.2	0.502
5	588	1.5	0.524	4	1,821,258	1.3	0.505
11	1,300	3.2	0.518	10	5,094,868	3.7	0.509
25	2,970	7.4	0.484	22	10,019,375	7.4	0.521
57	6,283	15.6	0.534	49	20,980,110	15.4	0.545
129	11,950	29.6	0.550	108	35,523,732	26.1	0.597
290	9,130	22.6	0.617	238	38,386,954	28.2	0.696
652	5,349	13.3	0.710	520	19,009,291	14.0	0.838
1,467	1,962	4.9	0.787	1,137	3,365,828	2.5	0.932
3,299	448	1.1	0.821	2,487	227,260	0.2	0.994
Total	40,369	100.0	0.591	Total	136,104,236	100.0	0.648

players have higher probability to win matches. When the skill gap is high enough, there is still a probability for the lower skill player to win the match, especially in DGS. The reason may be that a number of players play casually in online board games, and the amount of DGS matches is not as large as that of FICS. The percentage of matches with large skill gaps (over 500) is over 16%, which indicates that players in those matches may not have a good gaming experience and that the board game operators should improve the quality of their matchmaking systems.

2.4.3 Analysis of Player Behavior and Evolution

Due to the large variety of available games, players have many choices. The selection can be influenced by both the content of games and the quality of the offered service. Retaining players with longer player lifetime (from the first time till the last time a player has been seen) can yield more revenue for game companies. In this subsection, we study how many matches played by departure players over their player lifetimes and analyze how the player lifetime interacts with the number of in-game friends, and in-game play strategy.

Normalized Match Count of Departure Players

In this subsection, we analyze the player departure behavior in Go, chess, and DotA. We assume that a departure player is a people who did not play any match in the last month of the game trace. We count the number of played matches every week for each

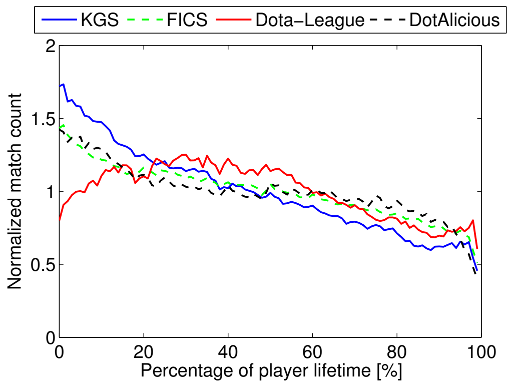


Figure 2.11: Normalized match count over player lifetime.

departure player. After that, we map the weekly match count into the percentage of player lifetime. Next, we calculate the average match count of all players at the same percentage of their lifetime. Finally, as shown in Figure 2.11, we plot the *normalized match count* as the ratio between the match count at one percentage and the average match count of the whole player lifetime. We can find that departure players play mostly at the beginning, and then, they play less matches gradually towards the ends, regardless of the type of games and platforms.

Similarly, Feng et al. [33] shows the EVE Online players spend less time on the game before quitting, with shorter session time and longer intersession time. Tarng et al. [131] try to predict ShenZhou Online gamer's departure on their average daily playtime and playing density.

Player Lifetime and Match Count vs. Number of Friends

We have discussed the influence of friendship on the performance of players in matches (Section 2.4.2). We now study how the friendship affects player lifetime in DotA-League.

Figure 2.12 illustrates that players with more friends generally stick to the game longer (left vertical axis) and play more matches (right vertical axis). The friendship does have a strong correlation with player lifetime ($R^2 = 0.8412$, $P < 0.01$). Thus, it would be a good idea for the game operators to maintain players by reminding players to make more friends and by providing convenient services to support social interaction. Unlike Facebook, where users have on average of about 130 friends, most players here have at most 60 friends.

Player Lifetime vs. Play Strategy

Predicting player lifetime is an important task for a game company, because if a company can predict how long the player's game lifetime will be, it can both leverage some methods

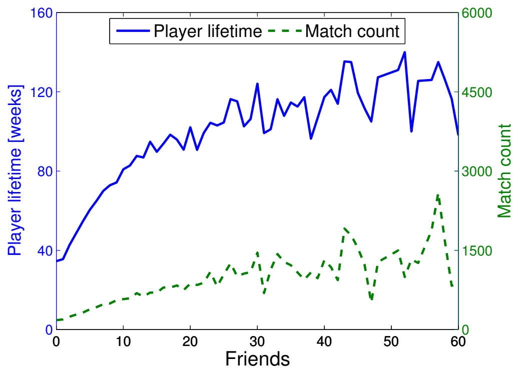
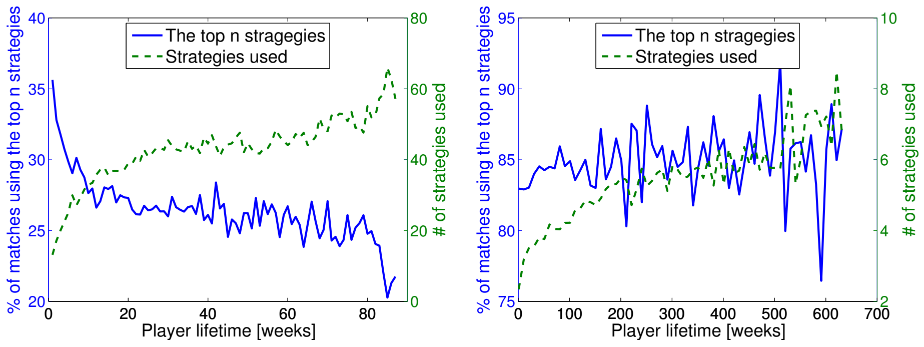


Figure 2.12: Number of friends with player lifetime and match count.



(a) DotAlicious (b) FICS

Figure 2.13: The evolution of play strategy with player lifetime.

to prolong player lifetime and better market in-game or side products. Although lifetime prediction has been researched for the past 5 years, only a small fraction of these studies have taken into account the players' in-game behavior.

We study the number of strategies players use in DotAlicious and FICS. In DotAlicious, the in-game characters of players are called "heroes". Different heroes represent different types of strategies. In FICS, for simplicity, the first move of a player in a match represents a strategy. There are over 100 available strategies in our DotAlicious dataset and 20 available strategies in FICS.

Figure 2.13 shows the average number of strategies used by players with different lifetimes. The horizontal axis shows the lifetime of players, and the percentage of matches using the top n strategies--by match count--is depicted against the left vertical axis, while the right vertical axis shows the number of strategies used. The value of n is 3 and 1 in DotAlicious and FICS, respectively. In general, there is a positive correlation between player lifetime and the number of used strategies ($R^2 = 0.8789$, $P < 0.01$): the longer the player lifetime is, the more strategies he will have used. The top n strategies account

Table 2.9: Graph metrics of the largest connected component.

Trace	**# Comps**	**# Nodes**	**# Edges**	**d** ($\times 10^{-4}$)	$\mathbf{\bar{D}}$
KGS	6,099	819,249	17,884,783	0.53	44
BBO	11	206,333	13,654,906	6.41	132
FICS	2,574	356,244	50,460,185	7.95	283
XFire	198	7,733,276	29,257,283	0.01	8
Dota-League	1	61,171	50,870,316	272	1663
DotAlicious	1	64,083	20,006,143	97.4	624
Dota Garena	1,544	291,706	2,767,594	0.65	19

# Comps is the number of connected components in the graph.
# Nodes, **# Edges**, **d**, and $\mathbf{\bar{D}}$ are the number of nodes, the number of edges, the link density, and the average degree of the largest connected component, respectively.

for a large majority of matches, which indicates that players are conservative. According to Figure 2.13, if the amount of available strategies is larger, players may spend more time in exploring all the strategies. As for the game operators, they may need to provide (better) awards to encourage players to try new strategies.

2.4.4 Analysis of Gaming Graphs

The relationships between entities in game traces can form gaming graphs. The structure and evolution of gaming graphs may become essential for game operation. The formatted Relationship Graph Dataset in the GTA makes it easier to investigate the graph characteristics of various games. In this subsection, we analyze the gaming graph for seven traces, except WoWAH and RS, which do not have the graph information. We form the graphs as follow: for all traces, each node represents a player. Then, for XFire each edge expresses a player-to-player friendship; for all the other traces, each edge represents a played game match.

Table 2.9 shows, for each trace, typical graph metrics of the largest connected component of the gaming graph. A connected component is a sub-graph in which all pairs of nodes are reachable. The largest connected component contains almost all the nodes of the whole graph in each game trace, which means that nearly every player can reach any other player in the trace. From the link density, all these largest connected components are *not* dense. For XFire, the average friend count per user ($\mathbf{\bar{D}}$ in Table 2.9) is much lower than for Facebook: 8 vs. 130. For the other traces, the relationship counts range from 19 to 1663; we leave for future work a study of the strength and meaning of such game relationships.

2.5 Related Work

In the game community, many researchers have published their work by analyzing their own game traces. However, the type and number of game traces used in the previous work are few, and comprehensive comparative experiments are seldom. Most of those research are based on individual game trace [129], [26], [16], [77]. In [13], the authors study the characteristics of online games by four game traces in three different game genres. However, the traces used in this work only cover a small part of the online games and the analysis of game traces from same game genre is not sufficient.

A number of archives have been build in the other computer science areas. The Internet Traffic Archive (ITA) [20] archives internet network traffic traces for the research of network characteristics. The Parallel Workloads Archive (PWA) [132] collects workloads from parallel environments. The Grid Workloads Archive (GWA) [67] is a grid trace repository and a community center for the grid area. For the peer-to-peer community and wireless network community, the Peer-to-Peer Trace Archive (P2PTA) [149] and CRAWDAD [148] are designed, respectively. These archives are limited to their specific areas, they can not cover the game traces.

Social network and large-scale graph analysis are popular research topic in recent years. Large-scale graphs also exist in the gaming area, forming by the game entities and their relationships. We already store more complex information, such as more types of nodes and relationships, in our gaming graphs. Many formats have been designed to enable the exchange of social networks and graphs, but none of them is able to cover the gaming graphs. The SNAP [121] project is a high efficient library to analyze large scale networks and graph, however, the format in this project can not be extended to express multi relationships. The DyNetML [134] is an XML-based format which is extensible to express abundant social network data. The disadvantage of this format is that plenty of markups and delimiter tags are needed to store each data item, which consumes more storage space and require more memory in data processing. This disadvantage will become truly visible when this format is used to store large-scale gaming graphs.

2.6 Summary

We have designed the Game Trace Archive to address the lack of game traces for the game community. We have proposed a unified Game Trace Format that underlies a generic game and OMGN data repository. Our format includes the Relationship Graph Dataset, the Node Dataset, and the Other Game-related Dataset. We have also provided a toolbox and mechanisms to facilitate trace sharing and community building.

The currently archived traces have shown that the GTA can already store different types of game traces, including board game, card game, RTS, MMORPG, and OMGN.

Our example trace analysis has revealed that the GTA can support comparative analysis of gaming traces on the workloads of online games, winning of match-based games, player behavior and evolution, and gaming graph.

Chapter 3

Evaluating the Performance of CPU-Based Graph Processing Systems

In this chapter, we envision seven challenges and propose an empirical method for evaluating graph-processing systems. We define a comprehensive process, and a selection of representative metrics, datasets, and algorithmic classes. We implement a benchmarking suite of five classes of algorithms and seven diverse graphs. Our suite reports on basic (user-lever) performance, resource utilization, scalability, and various overhead. We use our benchmarking suite to analyze and compare six systems. We gain valuable insights for each system and present the first comprehensive comparison of graph-processing systems.

3.1 Overview

Large-scale graphs are increasingly used in a variety of business applications, such as social applications, online gaming, business intelligence and logistics, and bioinformatics [55, 84]. By analyzing the graph structure and characteristics, analysts are able to predict the behavior of the customer, and tune and develop new applications and services. However, the diversity of the available graphs, of the processing algorithms, and of the graph-processing platforms currently available to analysts makes the selection of a platform an important challenge. Although performance studies of individual platforms exist [25, 92], they have been so far restricted in scope and size. In this chapter, we propose a step forward: a comprehensive experimental method for analyzing and comparing graph-processing platforms. We further implement this method as a benchmarking suite and we apply it on six popular platforms. Our initial target is to provide benchmarking functionality for Small and Medium Enterprises (SMEs), who have access to clusters of a few tens of machines.

For both system developers and graph analysts (system users), a thorough understanding of the performance of these *platforms* (which we define as the combined hardware, software, and programming system that is being used to complete a graph processing task), under different input graphs and for different algorithms, is important—it enables informed choices, system and application tuning, and best-practices sharing. However, the execution time, the resource consumption, and other performance and non-functional characteristics of graph-processing systems depend to a large extent on the dataset, the algorithm, and the graph-processing platform. Thus, gaining a thorough understanding of graph-processing performance is impeded by three dimensions of diversity.

Dataset diversity: We are witnessing a significant increase in the availability and collectability of datasets represented as graphs, from road to social networks, and from bioinformatics material to citation databases.

Algorithm diversity: A large number of graph algorithms have been implemented to mine graphs for calculating basic graph metrics [89], for partitioning graphs [11, 24], for traversing graphs [96, 122, 144], for detecting communities [85, 107], for searching for important vertices [12, 106], for sampling graphs [22, 78], for predicting graph evolution [84], etc.

Platform diversity: Many types of platforms are being used for different communities of developers and analysts. Addressing a variety of functional and non-functional requirements, a large number of processing platforms are becoming available. Neo4j [101], HyperGraphDB [66], and GraphChi [80] are examples of efficient single-node platforms with limited scalability. To scale-up, distributed systems with more computing and memory resources are used to process large-scale graphs, but they can even be less efficient than single-node platforms. For example, generic data processing systems such as Hadoop [141], YARN [147], Dryad [68], Stratosphere [138], and HaLoop [7] can scale out on multiple nodes, but may exhibit low performance due to distribution and overheads. Graph-specific platforms such as Pregel [92], Giraph [43], PEGASUS [71], GraphLab [87], and Trinity [116] are designed to provide feasible alternatives, but are not yet thoroughly evaluated for non-trivial algorithms and large datasets.

New performance evaluation and benchmarking suites are needed to respond to the three sources of diversity, that is, to provide comparative information about different performance metrics of different platforms, through the use of empirical methods and processes. However, the state-of-the-art in comparative graph-processing platform evaluation relies nowadays on Graph500 [48], the de-facto standard for comparing the performance of the hardware infrastructure related to graph processing. By choosing BFS as the single representative application and a single class of synthetic datasets, Graph500 has triggered a race in which winners use heavily optimized, low-level, hardware-specific code [130], which is rarely found or reproduced by common graph processing deployments and thus rarely reaches SMEs. Moreover, even the few existing platform-centric comparative studies are usually performed to prove the superiority of a given system over its direct competitors, so they only address a limited set of metrics and do not provide sufficient detail regarding the causes that lead to performance gaps.

Addressing the lack of a comprehensive evaluation method and set of results for graph processing platforms, this chapter addresses a key research question: *How well do graph processing platforms perform?* To answer this question, we propose an empirical performance-evaluation method for (large-scale) graph-processing platforms. Our method relies on defining a comprehensive evaluation process, and on selecting representative datasets, algorithms, and metrics for evaluating important aspects of graph-processing platforms—execution time, resource utilization, vertical and horizontal scalability, and overhead. Using this method, we create the equivalent of a benchmarking suite by selecting and implementing five algorithms and seven datasets from different application domains.

We implement this benchmarking suite on six popular platforms currently used for graph processing—Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j—and conduct a comprehensive performance study. This demonstrates that our benchmarking suite can be applied for many existing platforms, and also provides a first and detailed performance comparison of the six selected platforms. Our approach exceeds previous performance evaluation and benchmarking studies in both breadth and depth: we implement and measure multiple algorithms, use different types of datasets, and provide a detailed analysis of the results. Scale-wise, our study aligns well with SMEs cluster sizes and matches state-of-the-art studies (Section 3.7). Our work also aligns with the goals and ongoing activity of the SPEC Research Group and its Cloud Working Group[1], of which the author is a member.

Our main contributions are:

1. We propose a method for the comprehensive evaluation of graph processing platforms (Section 3.3), which defines an evaluation process, and addresses multiple performance aspects such as raw performance, scalability, and resource utilization. When selecting the algorithms and datasets, the proposed method is equivalent to a benchmarking suite for graph-processing platforms; we select in this chapter five algorithms and seven datasets. We discuss the limitations of the coverage and representativeness of our comprehensive evaluation (Section 3.6).

2. We demonstrate how our benchmarking suite can be implemented for six different graph processing platforms (Section 3.4).

[1]http://research.spec.org/working-groups/rg-cloud-working-group.html

3. We provide a first performance comparison of six graph-processing platforms, quantifying their strong points and identifying their limitations (Sections 3.5).

3.2 Our Vision for Benchmarking

Benchmarking is a traditional approach to evaluate the performance of systems, with many well-known challenges: simplicity, cost- and time-effectiveness, verifiability, etc. However, benchmarking systems under different application can lead to specific challenges. In this section, we discuss seven challenges in benchmarking graph-processing platforms. Although there are more challenges to resolve, we argue that these would lead to a good benchmarking process, similar to what has been achieved by the TPC and SPEC communities for benchmarking databases, CPU power and energy, etc.

3.2.1 Methodological Challenges

Challenge 1. Evaluation process: Traditionally, it is a challenge to define an evaluation process that would define an equivalent benchmarking process for each platform (for example, not controlling the amount of tuning can lead to a war-of-wizards). For graph-processing platforms, the evaluation process needs to fairly define at least the data format, realistic processing workflows, and the multi-tenancy rules—although these concepts have been considered in the past, they need revisiting for graph processing. Although the mathematical notion of a graph allows for only a few varieties, in graph-processing applications we have seen various data structures, input formats, and number of dimensions for the dataset. Similarly to the idea that a single query may expand in several data operations, in graph-processing it is likely that processing workflows comprised of several atomic operations (single algorithms) is representative of the typical analysis task; the evaluation process should also include such workflows. Because graph-processing platforms are typically serving multiple users, much like modern databases and distributed batch-processing systems, the evaluation process should also consider how the workloads of multiple system tenants influence each other.

Challenge 2. Selection and design of performance metrics: To serve more users, one important issue for benchmarking graph-processing platforms is to provide performance metrics for a variety of platform characteristics. Typical performance metrics such as execution time, resource utilization, scalability, system overhead, power consumption, cost, etc., may be included. To compare platforms on top of various types and amounts of hardware resources (e.g. number of cores or size of memory), new normalized metrics may need to be defined and adopted. For example, Graph500 introduces the graph-specific metric *traversed edges per second* (TEPS). We argue that there is much room for metric definition. As another example, as the field spans database, parallel, and distributed systems, normalized metrics for weak and strong scaling of possibly heterogeneous platforms need to be devised. Moreover, there is a need to adapt traditional metrics to an elastic infrastructure, for example because the local infrastructure may be complemented with nodes leased temporarily from Infrastructure-as-a-Service clouds [42].

Challenge 3. Dataset selection: Selecting a representative dataset is a traditional problem in benchmarking, which requires revisiting for each new domain. As we present in section 3.1, graphs may differ significantly in size, structure, directivity, connectivity, etc. The main goal of the dataset selection is to choose relevant graphs with different characteristics; to make the benchmark time- and cost-effective, they should also be *few*, easy to generate at different scales (see Challenge 5), and stored in a similar format (see Challenge 1). Additionally, this challenge also requires that the selected graphs should be able to stress bottlenecks of graph-processing platforms.

Challenge 4. Algorithm coverage: Similar to Challenge 3, we find challenging the selection of a representative, reduced set of graph-processing algorithms, which may stress diverse components of the graph-processing platform. To reduce the number of algorithms, it should be possible to divide them into classes, based on their functionality and to select representative algorithms from each class. This solution also has some limitations: how to define the classes? how to select a representative algorithm from a class? how to allow future algorithms to participate in the benchmark? etc.

3.2.2 Practical Challenges

Challenge 5. Scalability of evaluation and selection processes: It is challenging to allow the users of a benchmark—developers and integrators of platforms, graph analysts, etc.—to cope with the scale of either the evaluation or the selection processes.

For the evaluation process, we aim at benchmarking platforms deployed on both large-scale infrastructure (e.g., wide-area multicluster systems, large data centers, supercomputers, etc.) and small-business infrastructure (e.g., clusters of only a few nodes, a single albeit powerful machine). Thus, the benchmarking suite, and in particular the input datasets, should match various operational scales—for datasets, from megabytes to petabytes. Currently, few real-world graphs of petabyte scale are available for bechmarking activities, and even datasets of hundreds of gigabytes are rare. Graph generators could produce graphs for testing of the required scale, for example the Kronecker generator used in Graph500, but pose important parallel/distributed computing challenges and may not have the characteristics of real-world graphs.

For the selection process, a community-oriented benchmarking process should also be able to match the possibly hundreds of metrics with the interests of the users. For graph analysts, a specific application may need to be matched against an entire database of benchmarking results, and a few most-promising systems should be selected. For system integrators, it would be helpful to identify which algorithms and graphs can stress the system for each selected metric. We believe that designing a community database of open results would be beneficial in addressing this challenge, but its design should be able to accommodate a wide variety of settings and thus remains an open challenge.

Challenge 6. Portability: As we discussed in the methodological challenges, the benchmarking suite includes a number of performance metrics, algorithms, and graphs. When benchmarking a platform, the graph-processing algorithms need to be re-implemented based on the platform's programming language and model and, possibly, also based on infrastructure characteristics. Re-implementing algorithms correctly and re-configuring reasonably of a platform need a solid experience of programming and a detailed understanding of the platform. The challenge is, thus, to design a benchmarking suite that balances the portability requirements with all the desired features.

Challenge 7. Result reporting: Another non-trivial practical aspect is to report benchmarking results, which should be done according to a precisely defined format. Comprehensive and standardized reports traditionally facilitate the understanding and the comparison of the performance of platforms. When users consider several performance metrics when comparing graph-processing platforms, a mechanism to combine the results from different performance metrics and report a single result may not be straightforward—in our experience [51], none of the distributed graph-processing platforms can deliver the best performance across all datasets and algorithms, even for the same metric. Other communities have faced this challenge in the past and were able to solve it. For example, SPEC benchmark results can include a full disclosure of the system configuration parameters; SPEC users can report both baseline (not tuned) and peak (tuned) performance results of systems. However, it took years of development and effort to achieve this by SPEC benchmarks.

3.3 Benchmarking Graph Processing Systems

To address the challenges introduced in Section 3.2, in this section we present an empirical method for evaluating the performance of graph-processing platforms. Our method includes four stages: identifying the performance aspects and metrics of interest; defining and selecting representative datasets and algorithms; implementing, configuring, and executing the tests; and analyzing the results.

3.3.1 Performance Aspects, Metrics, Process

To be able to reason about performance behavior, we first need to identify the performance requirements of graph-processing platforms, the system parameters to be monitored, the metrics that can be used to characterize platform performance, and an overall process how performance is evaluated.

In this study, we focus on four performance aspects:

1. *Raw processing power*: the ability of a platform to (quickly) process large-scale graphs. Ideally, platforms should combine deep analysis and short job runtimes. For SMEs, the latter could mean several minutes.

2. *Resource utilization*: the ability of a platform to efficiently utilize the resources it has. SMEs, who cannot afford inefficient use of their scarce resources, want platforms to waste as little compute and memory resources as possible.

3. *Scalability*: the ability of a platform to maintain its performance behavior when resources are added to its infrastructure. In our method, we test the strong scalability of platforms in both horizontal scale (by adding computing nodes distributedly) and vertical scale (by adding computing cores per node, thus ignoring network effects). Ideally, we want platforms to be able to automatically improve their performance linearly with the amount of added resources, but in practice this gain (or loss) depends both on the number and type of these resources, and on the algorithm and dataset.

4. *Overhead*: the part of wall-clock time the platform does not spend on true data processing. The overhead includes reading and partitioning the data, setting up the processing nodes, and eventually cleaning up after the results have been obtained. Ideally, the overhead should be constant and small relative to the overall processing time, but in practice the overhead may be related to algorithms and datasets.

Although we already use a more comprehensive set of workloads and metrics than the state-of-the-art (Section 3.7), there are still numerous limitations to our method, which we discuss in Section 3.6.

The performance aspects can be observed by monitoring traditional system parameters (e.g., the important moments in the lifetime of each processing job, the CPU and network load, the OS memory consumption, and the disk I/O) and be quantified through various performance metrics. We summarize in Table 4.1 the performance metrics used in this chapter; our technical report [51] defines them more thoroughly.

3.3.2 Selection of Graphs and Algorithms

This section presents a selection of graphs and algorithms, which is akin to identifying some of the main functional requirements of graph-processing systems. We further discuss the representativeness of our selection in Section 3.6.

Table 3.1: Summary of performance metrics.

Metric	**How measured?**	**Derived**	**Relevant aspect (use)**
job execution time (T)	Time the full execution	-	Raw processing power (Figure 3.1, 3.3, 3.4)
Edges per second (EPS)	-	$\#E/T$	Raw processing power (Figure 3.2)
Vertices per second (VPS)	-	$\#V/T$	Raw processing power (Figure 3.2)
CPU, memory, network	Monitoring sampled each second	-	Resource utilization (Figure 3.5, 3.6, 3.7)
Horizontal scalability	T of different cluster size (N)	-	Scalability (Figure 3.8)
Vertical scalability	T of different cores per node	-	Scalability (Figure 3.9)
Normalized edges per second (NEPS)	-	$\#E/T/N$	Scalability (Figure 3.8, 3.9)
Computation time (T_c)	Time actual for calculating	-	Raw processing power (Figure 3.10)
Overhead time (T_o)	-	$T - T_c$	Overhead (Figure 3.10)

$\#V$ and $\#E$ are the number of vertices and the number of edges of graphs, respectively.

Graph Selection

The main goal of the graph selection step is to select graphs with different characteristics but with comparable representation. We use the classic graph formalism [143]: a graph is a collection of vertices V (also called nodes) and edges E (also called arcs or links) which connect the vertices. A single edge is described by the two vertices it connects: $e = (u, v)$. A graph is represented by $G = (V, E)$. We consider both directed and undirected graphs. We do not use other graph models (e.g., hypergraphs).

Regarding the graph characteristics, we select graphs with a variety of values for the number of nodes and edges, and with different structures (see Table 4.2). We store the graphs in plain text with a processing-friendly format but without indexes. In our format, vertices have integers as identifiers. Each vertex is stored in an individual line, which for undirected graphs, includes the identifier of the vertex and a comma-separated list of neighbors; for directed graphs, each vertex line includes the vertex identifier and two comma-separated lists of neighbors, corresponding to the incoming and to the outgoing edges. Thus, we do not consider other data models proposed for exchanging and using graphs such as complex plain-text representations, universal data formats (e.g. XML), relational databases, relationship formalisms (e.g., RDF), etc.

Table 3.2: Summary of datasets.

	Graphs	#V	#E	d	$\bar{D}$	Directivity
G1	Amazon	262,111	1,234,877	1.8	5	directed
G2	WikiTalk	2,388,953	5,018,445	0.1	2	directed
G3	KGS	293,290	16,558,839	38.5	113	undirected
G4	Citation	3,764,117	16,511,742	0.1	4	directed
G5	DotaLeague	61,171	50,870,316	2,719.0	1,663	undirected
G6	Synth	2,394,536	64,152,015	2.2	54	undirected
G7	Friendster	65,608,366	1,806,067,135	0.1	55	undirected

d is the link density of the graphs ($\times 10^{-5}$). $\bar{\mathbf{D}}$ is the average vertex degree of undirected graphs and the average vertex in-degree (or average vertex out-degree) of directed graphs.

Table 3.3: Summary of algorithms.

	Algorithm	Main features	Use
A1	STATS	single step, low processing	decision-making
A2	BFS	iterative, low processing	building block
A3	CONN	iterative, medium processing	building block
A4	CD	iterative, medium or high processing	social network
A5	EVO	iterative (multi-level), high processing	prediction

Why these datasets? We select seven graphs which could match, in scale and diversity, the datasets used by SMEs. Table 4.2 shows the characteristics of the selected graph datasets. The graphs have diverse sources (e-business, social network, online gaming, citation links, and synthetic graph), and a wide range of different sizes and graph metrics (e.g., high vs. low degree, 1,663 vs. 2, respectively, directed and undirected graphs, etc.). The largest dataset (Friendster) in this chapter is larger but of the same order of magnitude in size as the median per-job dataset sizes observed in the workloads of Microsoft, Yahoo, and Facebook [108]. The synthetic graph ("Synth" in Table 4.2) is produced by the generator described in Graph500 [48]. The other graphs have been extracted from real-world use, and have been shared through the Stanford Network Analysis Project (SNAP) [121]) and the Game Trace Archive (GTA) [54].

Algorithm Selection

Why these algorithms? We have conducted a comprehensive survey of graph-processing articles published in 10 representative conferences, in recent years; in total, over 100 articles (see technical report [51]). We found that a large variety of graph processing algorithms exist in practice and are likely used by SMEs. The algorithms can be categorized into several groups by functionality, consumption of resources, etc. We focus on algorithm functionality and select one exemplar of each of the following five algorithmic classes, which are common in our survey: general statistics, graph traversal (used in

Graph500), connected components, community detection, and graph evolution. We describe in the following the five selected algorithms and summarize their characteristics in Table 4.3.

The General statistics (STATS) algorithm computes the number of vertices and edges, and the average of the local clustering coefficient of all vertices. The results obtained with STATS can provide the graph analyst with an overview of the structure of the graph.

Breadth-first search (BFS) is a widely used algorithm in graph processing, which is often a building block for more complex algorithms, such as item search, distance calculation, diameter calculation, shortest path, longest path, etc. BFS allows us to understand how the tested platforms cope with lightweight iterative jobs.

Connected component (CONN) is an algorithm for extracting groups of vertices that can reach each other via graph edges. This algorithm produces a large amount of output, as in many graphs the largest connected component includes a majority of the vertices.

Community detection (CD) is important for social network applications, as users of these networks tends to form communities, that is, groups whose constituent nodes form more relationships within the group than with nodes outside the group. Communities are also important in the gaming industry, as the market has an increasingly larger share of social games or of games for which the social component is important.

Graph evolution (EVO): an accurate EVO algorithm not only can predict how a graph structure will evolve over time, but can also help to prepare for these changes (for example data size increase). Thus, graph evolution is an important topic in the field of large-scale graph processing.

STATS and BFS are textbook algorithms. For CONN, CD and EVO, there are a number of variations. Considering the reported performance and accuracy of these algorithms, we select a cloud-based connected component algorithm created by Wu and Du [144], the real-time community detection algorithm proposed by Leung et al. [85], and the the Forest Fire Model for graph evolution designed by Leskovec et al. [84].

3.4 Experimental Setup

The method introduced in Section 3.3 defines a benchmarking skeleton. In this section we create a full benchmarking suite (bar the issues explained in Section 3.6) by implementing the graph-processing algorithms of a selected set of test platforms, and by configuring and tuning these platforms.

3.4.1 Platform Selection

We use a simple taxonomy of platforms for graph processing. By their use of computing machines, we identify two main classes of platforms: non-distributed platforms and dis-

Table 3.4: Selected platforms.

Platform	Version	Type	Release date
Hadoop	hadoop-0.20.203.0	Generic, Distributed	2011-05
YARN	hadoop-2.0.3-alpha	Generic, Distributed	2013-02
Stratosphere	Stratosphere-0.2	Generic, Distributed	2012-08
Giraph	Giraph 0.2 (revision 1336743)	Graph, Distributed	2012-05
GraphLab	GraphLab version 2.1.4434	Graph, Distributed	2012-10
Neo4j	Neo4j version 1.5	Graph, Non-distributed	2011-10

tributed platforms; distributed platforms use multiple computers when processing graphs. Orthogonally to the issue of distributed machine use, we divide platforms into graph-specific platforms and generic platforms; graph specific platforms are designed and tuned only for processing graph data. Importantly, we omit in our taxonomy parallel platforms; for the scale in our real-world experiments, we see the performance of distributed systems as being a conservative estimate of what a similarly sized but parallel system can achieve.

Why these platforms? We select for this study a graph-specific non-distributed platform, and both graph-specific and generic distributed platforms. Because of what we see that relatively little interest in the community, we do not select for this study any generic and non-distributed platform. Table 4.4 summarizes our selected platforms: Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. These six selected platforms are popular and may be used for graph processing. We introduce each platform in the following, in turn.

Hadoop [141] is an open-source, generic platform for big data analytics. It is based on the MapReduce programming model. Hadoop has been widely used in many areas and applications, such as log analysis, search engine optimization, user interests prediction, advertisement, etc. Hadoop is becoming the de-facto platform for batch data processing. Hadoop's programming model may have low performance and high resource consumption for iterative graph algorithms, as a consequence of the structure of its map-reduce cycle. For example, for iterative graph traversal algorithms Hadoop would often need to store and load the entire graph structure during each iteration, to transfer data between the map and reduce processes through the disk-intensive HDFS, and to run an convergence-checking iteration as an additional job. However, comprehensive results regarding graph-processing using Hadoop have not yet been reported.

YARN [147] is the next generation of Hadoop. YARN can seamlessly support old MapReduce jobs, but was designed to facilitate multiple programming models, rather than just MapReduce. A major contribution of YARN is to separate functionally resource management and job management; the latter is done in YARN by a per-application manager. For example, the original Apache Hadoop MapReduce framework has been modified to run MapReduce jobs as an YARN application manager. YARN is still under development. We select YARN because our hypothesis is that even for the same programming model

(YARN and Hadoop), the architecture of the execution engine matters.

Stratosphere [138] is an open-source platform for large-scale data processing. Stratosphere consists of two key components: Nephele and PACT. Nephele is the scalable parallel engine for the execution of data flows. In Nephele, jobs are represented as directed acyclic graphs (DAG), a job model similar for example to that of the generic distributed platform Dryad [68]. For each edge (from task to task) of the DAG, Nephele offers three different types of channels for transporting data, through the network, in-memory, and through files. PACT is a data-intensive programming model that extends the MapReduce model with three more second-order functions (Match, Cross, and CogGroup, in addition to Map and Reduce). PACT supports several user code annotations, which can inform the PACT compiler of the expected behavior of the second-order functions. By analyzing this information, the PACT compiler can produce execution plans that avoid high cost operations such as data shipping and sorting, and data spilling to the disk. Compiled PACT programs are converted into Nephele DAGs and executed by the Nephele data flow engine. HDFS is used for Stratosphere as the storage engine.

Giraph [43] is an open-source, graph-specific distributed platform. Giraph uses the Pregel programming model, which is a vertex-centric programming abstraction that adapts the Bulk Synchronous Parallel (BSP) model. An BSP computation proceeds in a series of global supersteps. Within each superstep, active vertices execute the same user-defined computation, and create and deliver inter-vertex messages. Barriers ensure synchronization between vertex computation: for the current superstep, all vertices complete their computation and all messages are sent before the next superstep can start. Giraph utilizes the design of Hadoop, from which it leverages only the Map phase. For fault-tolerance, Giraph uses periodic checkpoints; to coordinate superstep execution, it uses ZooKeeper. Giraph is executed in-memory, which can speed-up job execution, but, for large amounts of messages or big datasets, can also lead to crashes due to lack of memory.

GraphLab [87] is an open-source, graph-specific distributed computation platform implemented in C++. Besides graph processing, it also supports various machine learning algorithms. GraphLab stores the entire graph and all program state in memory. To further improve performance, GraphLab implements several mechanisms such as: supporting asynchronous graph computation to alleviate the waiting time for barrier synchronization, using prioritized scheduling for quick convergence of iterative algorithms, and efficient graph data structures and data placement. To match the execution mode of the other platforms, we run all our GraphLab experiments in a synchronized mode.

Neo4j [101] is one of the popular open-source graph databases. Neo4j stores data in graphs rather than in tables. Every stored graph in Neo4j consists of relationships and vertices annotated with properties. Neo4j can execute graph-processing algorithms efficiently on just a single machine, because of its optimization techniques that favor response time. Neo4j uses a two-level, main-memory caching mechanism to improve its performance. The file buffer caches the storage file data in the same format as it is stored on the durable storage media. The object buffer caches vertices and relationships (and their properties) in a format that is optimized for high traversal speeds and transactional writes.

3.4.2 Platform and Experimental Configuration

Platform tuning: The performance of these systems depends on tuning. Several of the platforms tested in this chapter have tens to hundreds of configuration parameters, whose actual value can potentially change the performance of the platform. We use common best-practices for tuning each of the platforms, as explained in our technical report [51].

Hardware: We deploy the distributed platforms on DAS4 [21], which is to provide a common computational infrastructure for researchers within Advanced School for Computing and Imaging in the Netherlands. Each machine we used in the experiments from DAS4 consists of a Intel Xeon E5620 2.4 GHz CPU (dual quad-core, 12 MB cache) and a total memory of 24 GB. All the machines are connected by 1 Gbit/s Ethernet network. NFS is used as the file system in DAS4. The operation system installed on each machine is CentOS release 6.3 with the kernel version 2.6.32. We use a single machine with one single enterprise SATA disk (SATA 3 Gbit/s, 7200 rpm, 32 MB cache) for the Neo4j experiments.

Platform configuration, number of nodes: We deploy the distributed platforms on 20 up to 50 computing machines of DAS4. We set the Neo4j on a single DAS4 machine with regular configuration. For all the experiments of Hadoop, YARN, Stratosphere, and GraphLab, besides the computing machines, we allocate an additional node to take charge of all master services. For Giraph, we use one more node for running ZooKeeper.

Parameters of Algorithms: We try to configure each algorithm with default parameter values. STATS and CONN do not need any parameters. For BFS, we randomly pick a vertex to be the source for each graph. We use only out-edges to propagate for directed graph, thus the directed graphs are not entirely traversed. We set the parameters of algorithms identically on all platforms.

Further experiment configuration: Unless otherwise stated, we repeat each experiment 10 times, and we report the average results from these runs. (An example where 10 repetitions would take too long is presented in Section 3.5.4).

3.5 Experimental Results

In this section we present a selection of the experimental results. We evaluate the six graph processing platforms selected in Section 3.4, using the process and metrics, and the datasets and algorithms introduced in Section 3.3. Compared with the previous work (Section 3.7), our experiments show more comprehensive and quantitative results in diverse metrics.

The experiments we have performed are:

- Basic performance (Section 3.5.1): we have measured the job execution time on a fixed infrastructure. Based on these execution times, we further report throughput numbers, using the edges per second (EPS) and vertices per second (VPS) metrics.
- Resource utilization: we have investigated the CPU utilization, memory usage, and network traffic.
- Scalability (Section 3.5.3): we have measured the horizontal and vertical scalability of the platforms; we report the execution time and the normalized edges per second (NEPS) for interesting datasets.
- Overhead (Section 3.5.4): we have analyzed the execution time in detail, and report important findings related to the platform overhead.

3.5.1 Basic Performance: Job Execution Time

The fixed infrastructure we use for our basic performance measurements is a cluster of 20 homogeneous computing nodes provisioned from DAS4. With the configuration in [51], each node is restricted at using a single core for computing. We configure the cluster as follows. For the experiments on Hadoop and YARN, we run 20 map tasks and 20 reduce tasks on the 20 computing nodes. Due to the settings used for Hadoop [51], the map phase will be completed in one wave; all the reduce tasks can also be finished in one wave, without any overlap with the map phase [42]. In Giraph, Stratosphere, and GraphLab, we set the parallelization degree to 20 tasks, also equal to the total number of computing nodes.

With these settings, we run the complete set of experiments (6 platforms, 5 different applications, and 7 datasets) and measure the execution time for each combination. In the remainder of this section, we present a selection of our results.

Key findings:

- There is no overall winner, but Hadoop is the worst performer in all cases.
- Multi-iteration algorithms suffer for additional performance penalties in Hadoop and YARN.
- EPS and VPS are suitable metrics for comparing the platforms throughput.
- The performance of all the platforms is stable, with the largest variance around 10%.
- Several of the platforms are unable to process all datasets for all algorithms, and crash.

Table 3.5: Statistics of BFS.

	G1	G2	G3	G4	G5	G6	G7
Coverage [%]	99.9	98.5	100	0.1	100	100	100
Iterations	68	8	9	11	6	8	23

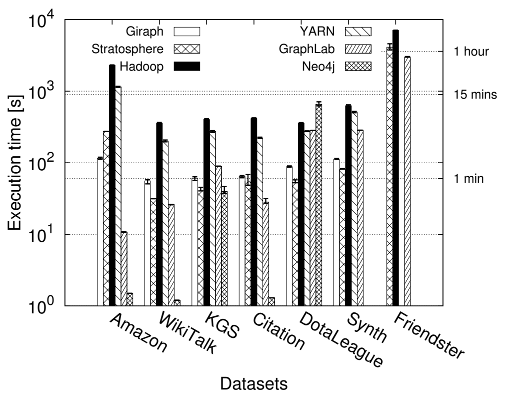


Figure 3.1: The execution time of algorithm BFS of all datasets of all platforms.

Results for One Selected Algorithm

We present here the results obtained for one selected algorithm, BFS (see Section 3.3.2).

Because the starting node for the BFS traversal will impact performance by limiting the coverage and number of iterations of the algorithm, we summarize in Table 3.5 the vertex coverage and iteration count observed for the BFS experiments presented in this section. Overall, BFS covers over 98% of the vertices, with the exception of the Citation (G4) dataset. The iteration count depends on the structure of each graph and varies between 6 and 68; we expect higher values to impact negatively the performance of Hadoop.

We depict the performance of the BFS graph traversal in Figure 3.1 and discuss in the following the main findings. Similarly to most figures in this section, Figure 3.1 has a logarithmic vertical scale.

Hadoop always performs worse than the other platforms, mainly because Hadoop has a significant I/O between two continuous iterations (see Section 3.4). In these experiments, Hadoop does not use spills, so it has no significant I/O within the iteration. As expected, the I/O overhead of Hadoop is worse when the number of BFS iterations increases. For example, although Amazon is the smallest graph in our study, it has the largest iteration count, which leads to a very long execution time. YARN performs only slightly better than Hadoop—it has not been altered to support iterative applications. Although Stratosphere is also a generic data-processing platform, it performs much better than Hadoop and YARN (up to an order of magnitude lower execution time). We attribute this to Stratosphere's ability to optimize the execution plan based on code annotations regarding data sizes and flows, and to the much more efficient use of the network channel.

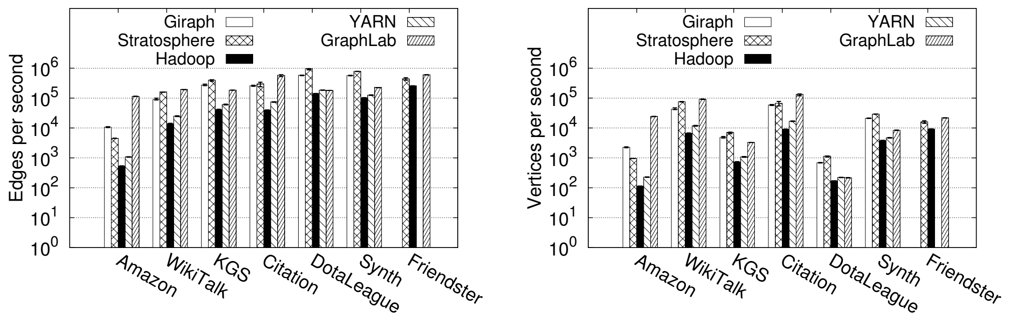


Figure 3.2: The EPS and VPS of executing BFS.

In contrast to the generic platforms, for Giraph and GraphLab the input graphs are read only once, and then stored and processed in-memory. Both Giraph and GraphLab realize a dynamic computation mechanism, by which only selected vertices will be processed in each iteration. This mechanism reduces the actual computing time for BFS, in comparison with the other platforms (more details are discussed in Section 3.5.4). In addition, GraphLab also addresses the problem of smart dataset partitioning, by limiting the cut-edges between machines when splitting the graph. These systemic improvements make the performance of both Giraph and GraphLab less affected by large BFS iteration counts than the performance of other distributed platforms.

Because of the two-level main-memory cache of Neo4j, we differentiate two types of executions: cold-cache (first execution) and hot-cache (follow-up executions). Figure 3.1 depicts the average results obtained for hot-cache executions. The two-level cache allows Neo4j to achieve excellent hot-cache execution times, especially when the graph data accessed by the algorithms fits in the cache. However, the cold-cache execution can be very long: for example, the ratios between the cold-cache and hot-cache BFS executions for Citation and DotaLeague are 45 and 5, respectively. Even for cold-cache execution, Neo4j reads from the database only the graph data needed by the algorithm. This “lazy read” mechanism minimizes the I/O overhead and accelerates traversal on the graphs where the BFS coverage of the graph is limited, e.g., for Citation. However, limited by the resources of a single machine, the performance of Neo4j becomes significantly worse when the graph exceeds the memory capacity. For example, the hot-cache value of Synth is about 17 hours, exceeding the scale of Figure 3.1.

We now report on the *achieved throughput for the BFS algorithm*, in both EPS and VPS, for all platforms and datasets (Figure 3.2). We note that throughput is a metric that takes into account the dataset structure and provides an indication of the platforms performance per data item—be it an edge or a vertex. For example, KGS and Citation, which have similar numbers of edges, file sizes, and BFS iteration counts, achieve similar EPS values on most platforms. The exception is GraphLab, in which the EPS of Citation is about two times larger than that of KGS. This is due to the restriction of GraphLab to

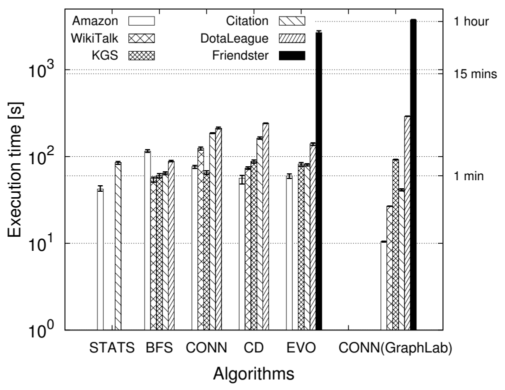


Figure 3.3: The execution time of all algorithms for all datasets running on Giraph, and for CONN running on GraphLab (right-most bars).

process only directed graphs, which has required the conversion of the undirected KGS to a directed version. This operation lead to a doubling in the number of edges, resulting in a proportional increase of the execution time.

Results for Two Selected Platforms

We focus in this section on the graph-specific platforms (Giraph and GraphLab) and discuss their performance for all the algorithms and datasets, as depicted in Figure 3.3.

As Giraph is an in-memory-only platform, its performance is not affected by the large penalties of I/O operations. Figure 3.3 shows that the execution time for most of the experiments is below 100 seconds. However, when the amount of messages between computing nodes becomes extremely large (tens of gigabytes), Giraph crashes. For example, Giraph crashes for the STATS algorithm running on the WikiTalk dataset; for Friendster, the largest of our datasets, Giraph completes only the EVO algorithm, for which our graph evolution algorithm generates relatively few messages. From the selected results, GraphLab performs better than Giraph for the CONN algorithm for most graphs. Moreover, GraphLab is able to process even the largest graph in this study.

Results for Two Selected Datasets

Finally, to understand the impact of algorithm complexity on each platform, we focus now on two interesting datasets—DotaLeague and Citation. We depict their performance, for all algorithms running on all platforms, in Figure 3.4.

Because Friendster is too large for some platforms, we present here the results for graphs that all platforms can process: the second-largest real graph, DotaLeague, and the small Citation graph. Even for the second-largest graph, Giraph, Hadoop and YARN crashed when running STATS; we also had to terminate Stratosphere after running STATS for nearly 4 hours without success; similarly, STATS and CD run for more than 20 hours

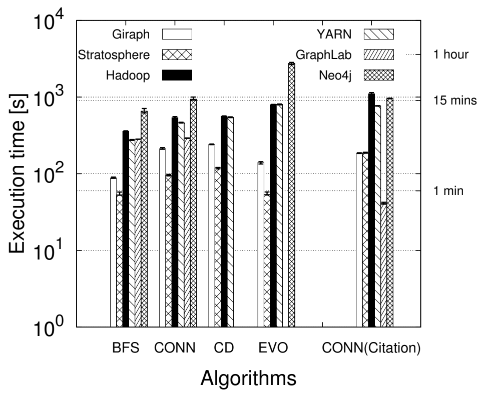


Figure 3.4: The execution time for all platforms, running all algorithms for the DotaLeague dataset, and CONN for the Citation dataset (right-most bars).

in Neo4j and are not shown in Figure 3.4. For the other algorithms, BFS, CONN, CD, and EVO, the number of iterations is between 4 and 6. From Figure 3.4, the execution time of BFS is lower than the execution time of CONN and CD, on all platforms. In each iteration of CONN and CD, many more vertices will be active, in comparison to BFS. Furthermore, in CONN, the number of active vertices stays relatively constant in each iteration, while CD is more compute-intensive and variable. For EVO, Stratosphere takes advantage of its programming model, as it can represent one EVO iteration by a single map-reduce-reduce procedure; in contrast, Hadoop and YARN need to run two MapReduce jobs per iteration and thus their execution time increases.

Citation is much smaller and sparser than DotaLeague. The CONN of Citation takes 20 iterations. The execution time of CONN of Citation on Hadoop, YARN, and Stratosphere increases compared with 6-iteration CONN of DotaLeague. As we explained for the analysis of BFS (Section 3.5.1), more iterations result in higher I/O and other overheads.

3.5.2 Evaluation of Resource Utilization

To understand the resource utilization of each platforms, we investigate in this section the CPU load, memory, and network usage of both the master node and the computing nodes.

For each platform, we execute BFS on DotaLeague. The configuration is consistent to Section 3.5.1. We monitor the platforms by using the Ganglia Monitoring System [40] with a sampling interval of 1 second. The monitoring results includes the usage of local system such as operating system.

To make the resource utilization results comparable, We normalize the execution time of different platforms and, for each platform, of different experiment runs (we use 10 repetitions of each experiment). For each experiment run, we linearly interpolate the real monitoring samples to obtain 100 normalized usage points for each resource. We depict

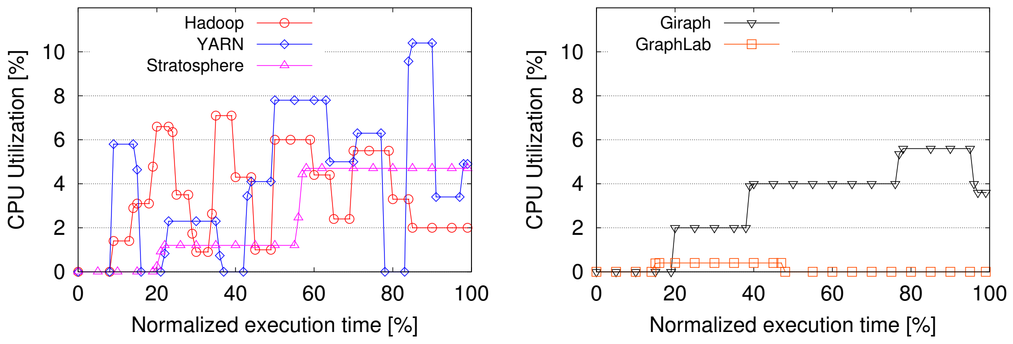


Figure 3.5: The CPU utilization of a computing node.

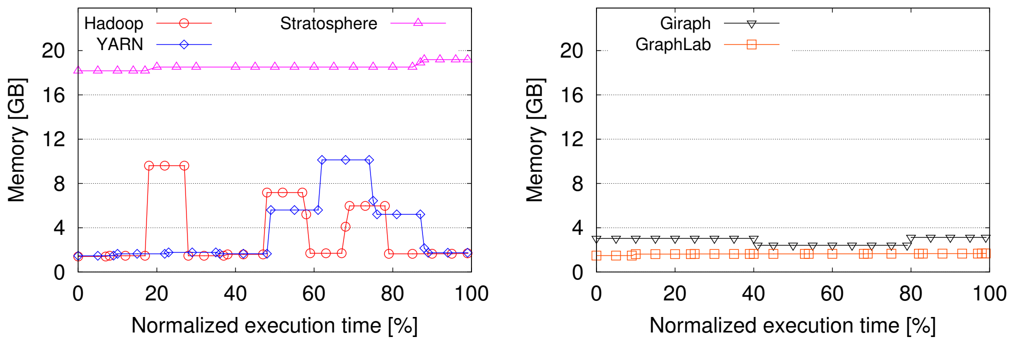


Figure 3.6: The memory usage of a computing node.

in each figure corresponding to the resource utilization of a computing node the results obtained in practice for a real computing node, such that the depiction is the closest to the average resource utilization observed in practice.

Key findings:

- Few resources are needed for the master node of all platforms.
- The resource utilization of the computing nodes varies widely across different platforms.

Possibly because there is only one job submitted to all platforms, the master does not heavily use resources. The CPU utilization and the network traffic have low usage for job management and platform operation (heartbeats, etc.). For all platforms, the CPU utilization is below 0.5% and the network traffic is less than 400 Kbit/s (the only exception is Stratosphere, which sometimes can reach up to 1 Mbit/s). The monitored memory usage of all platforms is around 8 GB, including the memory consumption of operating systems and services.

For computing nodes, Figures 3.5, 3.6, and 3.7 depict the CPU utilization, the memory usage, and the network traffic of all platforms, respectively. The resource utilization of computing nodes in YARN and Hadoop exhibit obvious volatility, due to the BFS job consisting of 6 independent iterations. However, the curves do not actually exhibit 6

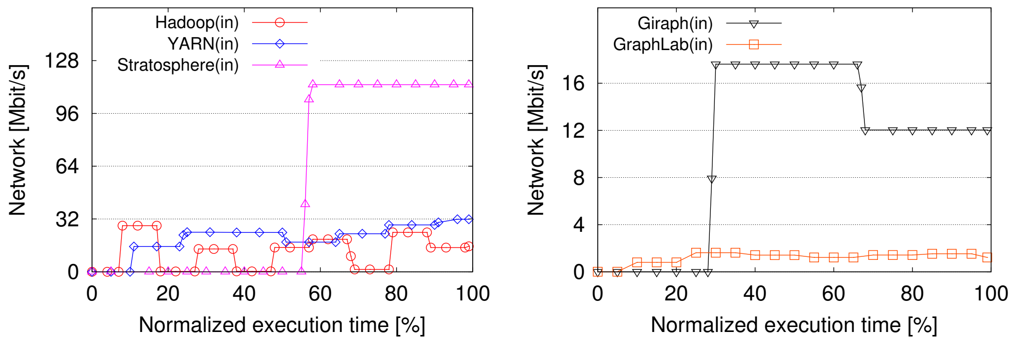


Figure 3.7: The network traffic of a computing node. Note that the scales of vertical axes are different.

usage spikes—the computing node with the resource utilization closest to the average is not used intensively in each of the 6 iterations. The memory usage of Stratosphere keeps around 20 GB, as configured [51]. This is because Stratosphere compute nodes allocate the memory assigned by the configuration immediately after startup. This design may decrease the possibility of resource sharing between Stratosphere and other applications. Moreover, by using the network channel for transporting data, Stratosphere exhibits the heaviest network throughput. Compared to the generic platforms, the resource usage of Giraph and GraphLab are much smaller. As we discussed in Section 3.5.1, the reason is the graph-friendly programming model of Giraph and GraphLab: these platforms only process activated vertices in each iteration, which reduces the resource requirement.

3.5.3 Evaluation of Scalability

In this section, we evaluate the horizontal and vertical scalability of the distributed platforms. Besides the job execution time, we also report the NEPS for comparing the performance per computing unit.

To allow a comparison with the previous experiments, we use BFS results. To test scalability, we use the two largest real graphs in our study, Friendster and DotaLeague (the results of DotaLeague are in our technical report [51]). For testing horizontal scalability, we increase the number of machines from 20 to 50 by a step of 5, and keep using a single computing core per machine. For testing vertical scalability, we keep the cluster size at 20 computing machines and increase the number of computing cores per machines from 1 to 7. We step up the number of map (reduce) tasks and parallelization degree equally to the available computing cores.

Key findings:

- Some platforms can scale up reasonably with cluster size (horizontally) or number of cores (vertically).

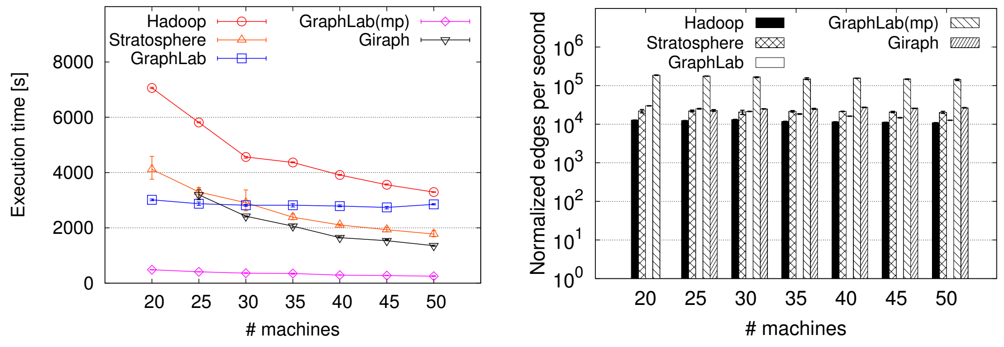


Figure 3.8: The horizontal scalability (left) and NEPS (right) of processing G7.

- Increasing the number of computing cores may lead to worse performance, especially for small graphs.
- The normalized performance per computing unit mostly decreases with the increase of cluster size and with the number of computing cores per node.

Horizontal Scalability

Figure 3.8 shows the horizontal scalability of BFS for Friendster (G7). Most of the platforms presents significant horizontal scalability, except for GraphLab, which exhibits little scalability. The horizontal scalability of GraphLab is constrained by the graph loading phase using one single file. We thus explore tuning GraphLab: for GraphLab(mp) we split the input file into *m*ultiple separate *p*ieces, as many as the MPI processes. GraphLab(mp) has much lower execution time than GraphLab. Moreover, GraphLab(mp) is scalable, as its execution time decrease from about 480 seconds to 250 seconds when resources are added.

We further investigate the performance per computing unit (computing node) to check if they also be improved. We calculate the EPS from the execution time and normalize it by the number of computing nodes to get the NEPS. The maximum value of NEPS can be reached at different sizes of the cluster, for different platforms. For example, the NEPS of Hadoop and Giraph peaks at 30 and 40 computing nodes of Friendster, respectively. However, the general trend of NEPS is to decrease with the increase of cluster size. We have obtained similar results for the vertex-centric equivalent of NEPS, NVPS.

Vertical Scalability

Figure 3.9 shows the vertical scalability of running BFS for Friendster (G7). There is no result of Giraph and YARN of Friendster, because both YARN and Giraph crashed on 20 computing machines. For Friendster, both Hadoop and Startosphere can benefit from using more computing cores. However, after 3 cores, the improvement become negligible.

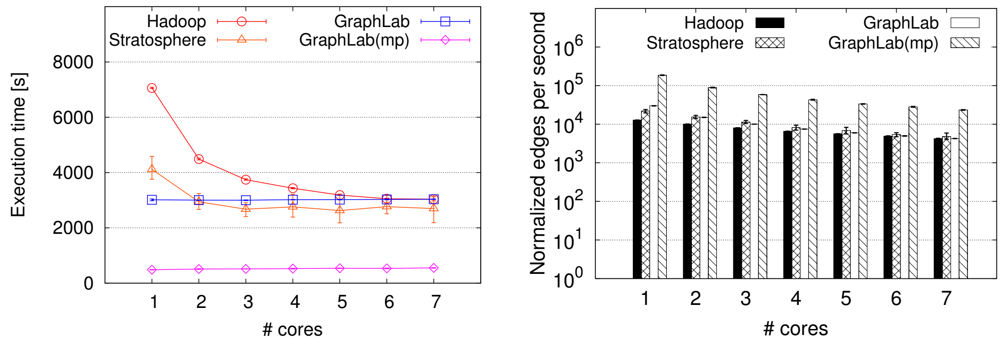


Figure 3.9: The vertical scalability (left) and NEPS (right) of processing G7.

By using more cores, graphs can be processed with higher parallelism, but may also incur latency, for example, due to concurrent accesses to the disk. For GraphLab(mp), for which we split the Friendster file into more pieces with the increase of the number MPI processes, the job execution time does not decrease correspondingly. The reason is that each MPI instance (or machine) has a just single loader for input files, thus in one machine, the MPI processes cannot load graph pieces in parallel.

We check the performance per computing unit (computing core) by NEPS in vertical scalability. We can find similar results to that of horizontal scalability, all NEPS drops for all platforms. The competition between computing cores makes the reduce of execution time not significant enough to improve the average performance of computing cores.

3.5.4 Evaluation of Overhead

In this section, we evaluate two elements of overhead: data ingestion time and execution time overhead.

Key findings:

- The data ingestion time of Neo4j matches closely the characteristics of the graph. Overall, data ingestion takes much longer for Neo4j than for HDFS.
- The data ingestion time of HDFS increases nearly linearly with the graph size.
- The percentage of overhead time in execution time is diverse across the platforms, algorithms, and graphs.

For Neo4j, data ingestion process converts input graphs to the format used by the Neo4j graph database. In contrast, the distributed platforms evaluated in this chapter use HDFS, which means for them data ingestion consists of data transfers from the local file system to HDFS. GraphLab even does not need data ingestion if using the local file system (i.e., NFS). Only for Neo4j, because data ingestion takes long (up to days), we only evaluate the data ingestion for Neo4j through one experiment repetition.

Table 3.6: Data ingestion time.

	G1	G2	G3	G4	G5	G6	G7
HDFS [s]	1.2	1.8	3.0	3.9	7.0	10.9	312.0
Neo4j [**h**]	2.0	17.2	2.6	28.8	3.7	24.7	N/A

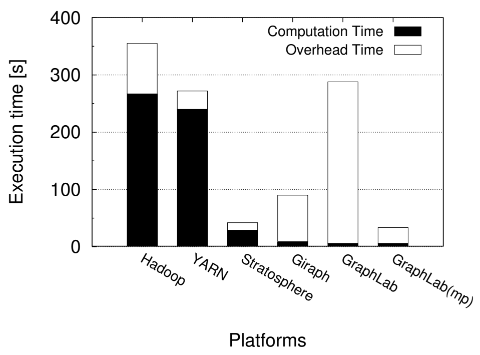


Figure 3.10: The execution time breakdown of platforms for BFS.

Table 3.6 summarizes the data ingestion results. The data ingestion time of Neo4j is up to several orders of magnitude longer than that of HDFS. In our experimental environment, which uses enterprise-grade magnetic disks, the data ingestion time of HDFS increases by about 1 second for every 100 MB of graph data. In contrast, the data ingestion time of Neo4j depends on the structure and scale of graphs, so it changes irregularly across the datasets in this study. Dominguez-Sal et al. [25] report similar results about data ingestion time in their survey of graph database performance.

We define computation time as the time used for making progress with the graph algorithms. The overhead time is the remainder from subtracting the computation time from the job execution time. Thus, the overheads include the time for read and write, and for communication. The fraction of time spent with overheads varies across the platforms (Figure 3.10). Although BFS is not a compute-intensive algorithm, Hadoop and Stratosphere need to traverse all vertices, which increases their computation time. In GraphLab, most of the time is spent on loading the graph into memory and on finalizing the results. The percentage of overhead time on each platform is closely related to the complexity of the algorithm and the characteristics of graph. For example, we also found that for Citation, the percentage of overhead time is 98% and 70% for BFS and CONN, respectively (see technical report [51]).

3.6 Discussion

The method proposed in Section 3.3 raises several methodological and practical issues that prevent it from being a benchmark. We argue that our method can result in meaningful, comprehensive performance evaluation of graph-processing platforms, but the path towards an industry-accepted benchmark still raises sufficient challenges. Outside the scope of this work, we continue to pursue resolving these issues via the SPEC Cloud Working Group.

Methodologically, our method has limitations in its process, workload design, and metrics design. Specifically, our *method* does not offer a detailed, infrastructure- and platform-independent process; for example, it does not limit meaningfully the amount of tuning done to a system prior to benchmarking and it does not precisely specify the acceptable components of a platform (should a cloud-based platform include the Internet linking its users to the data center?). The *workload design*, although it covers varied datasets and algorithms, does not feature an industry-accepted process of selection for them, and does not select datasets and algorithms that can stress a specific bottleneck in the system under test. Proving *algorithmic coverage* is currently not feasible, due to field fragmentation and lack of public workload traces; solving this ”chicken-and-egg” problem cannot be properly addressed without collecting workload traces for several years and from several major operators.

Metrics-wise, our method does not provide only *a single result*—which helps with the analysis of the causes of performance gaps between platforms—; does not provide metrics for a variety of interesting platform characteristics (e.g., power consumption, cost, efficiency, and elasticity); and could do more in terms of *normalized metrics* (i.e., by normalizing by various types of resources provided by the system, such as number of cores or size of memory). Using other metrics is outside the scope of our work, as different communities are interested in different operational aspects (SPEC identifies tens of relevant metrics).

From a practical perspective, our method has limitations in portability, time, and cost. The *portability* is limited by the need to re-implement algorithms for each platform and to re-configure platforms for each experiment. The *time* spent in implementing our method is analyzed in our technical report [51]. The *cost* of performing a benchmark, in particular in *tuning*, is a non-trivial issue, for which few benchmarks provide a solution. Another non-trivial practical aspect is reporting (an outcome of the analysis stage), which our method does not precisely specify. In contrast, SPEC benchmark users can report results for baseline (not tuned) and peak (tuned) systems, and SPEC results include a full disclosure of the parameters used in configuring the systems; however, SPEC benchmarks are sophisticated products and the result of years of development.

Experiments with larger environments and datasets, and with new algorithms and metrics, can add to the bulk of the results presented in this work. However, the need for such experiments is not supported by existing published evidence: as we show in Section 3.7, our work already extends and complements previous work. Requiring experiments in larger clusters, while relevant for companies such as Facebook (thousands of nodes), does not match the needs of SMEs that want efficient graph-processing, and the information

Table 3.7: Overview of related work. Legend: V–vertices, E–edges, C–computers.

Platforms	**Algorithms**	**Dataset type**	**Largest dataset**	**System**
Neo4j, MySQL [136]	1 other	synthetic	100 KV	1 C
Neo4j, etc. [25]	3 others	synthetic	1 MV	1 C
Pregel [92]	1 other	synthetic	50 BV	300 C
GPS, Giraph [112]	CONN, 3 others	real	39 MV, 1.5 BE	60 C
Trinity, etc. [116]	BFS, 2 others	synthetic	1 BV	16 C
PEGASUS [71]	CONN,2 others	synthetic, real	282 MV	90 C
CGMgraph [14]	CONN, 4 others	synthetic	10 MV	30 C
Hadoop, GraphLab, etc. [28]	1 other	real	3 MV, 117 ME	32 C
PBGL, CGMgraph [49]	CONN, 3 others	synthetic	70 MV, 1 BE	128 C
HaLoop, Hadoop [7]	2 others	synthetic, real	1.4 BV, 1.6 BE	90 C
Hadoop, PEGASUS [70]	1 other	synthetic, real	1 BV, 20 BE	32 C
Our work	**5 classes**	**synthetic, real**	**66 MV, 1.8 BE**	**50 C**

we have about state-of-the-art (Section 3.7).

These limitations also affect other benchmarks [35] and performance evaluation studies included in the related work of this study. As we indicated in the introduction of our article, we point out that the de-facto standard in benchmarking graph-processing platforms is Graph500 (one algorithm and one graph type); in contrast, our work provides a significant improvement in both algorithms (processing patterns and scope) and datasets (two public graph archives, several application domains, and various graph structures).

3.7 Related Work

Many previous studies focus on performance evaluation of graph-processing, for different platforms. Table 3.7 summarizes these studies and compares them with our work. Overall, for the studies in our survey, most of the datasets included in previous evaluation are synthetic graphs. Although some of the synthetic graphs are extremely large, they may not have the characteristics of real graphs. Our evaluation selects 6 *real* graphs and 1 synthetic graph with various characteristics. *Relative to our study, fewer classes of algorithms are used to compare the performance of platforms.* From our observation, a very limited number of metrics have been reported, with many of the previous studies focusing only on the job execution time. *Our work evaluates performance much more in-depth, by considering more types of metrics.* Finally, previous research compares few platforms; in contrast, *we investigate 6 popular platforms with different architectures. Our environment is of similar scale with state-of-the-art studies in distributed systems.*

3.8 Summary

A quickly increasing number of platforms can process large-scale graphs, and have thus become potentially interesting for a variety of users and application domains. We focus in this work on SMEs, which are businesses with little resources to spare in their graph-processing clusters. To compare in-depth the performance of graph-processing platforms, and thus facilitate platform selection and tuning for SMEs, we have proposed in this work an empirical method and applied it to obtain a comprehensive performance study of six platforms.

Our method defines an empirical performance evaluation process and selects metrics, datasets, and algorithms; thus, it acts as a benchmarking suite despite not covering all the methodological and practical aspects of a true benchmark. Our method focuses on four performance aspects: raw performance, resource utilization, scalability, and overhead. We use both performance and throughput metrics, and we also use normalized metrics to characterize scalability. We implement a benchmarking suite that uses five representative graph algorithms—general statistics, breadth-first search, connected component, community detection, and graph evolution—, and seven graphs that represent graph structures for multiple application domains, with sizes up to 1.8 billion edges and tens of GB of stored data.

Using our benchmarking suite, we conduct a first detailed, comprehensive, real-world performance evaluation of six popular platforms for graph-processing, namely, Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. Our results show quantitatively and comparatively the highlights and weaknesses of the tested platforms. The main lessons are listed at the start of each experiment.

Chapter 4

Evaluating the Performance of GPU-Enabled Graph Processing Systems

In this chapter, we adapt and extend our empirical method in Chapter 3, by identifying new performance aspects and metrics, and by selecting and including new datasets and algorithms. By selecting nine diverse graphs and three typical graph-processing algorithms, we conduct a comparative performance study of three GPU-enabled systems, Medusa, Totem, and MapGraph. We present the first comprehensive evaluation of GPU-enabled systems with results giving insight into raw processing power, performance breakdown into core components, scalability, and the impact on performance of system-specific optimization techniques and of the GPU generation. We present and discuss many findings that would benefit users and developers interested in GPU acceleration for graph processing.

4.1 Overview

Many graph-processing algorithms have been designed to analyze graphs in industry and academic applications, for example, item and friend recommendation in social networks [84], cheater detection in online games [54], and subnetwork identification in bioinformatics [64]. To address various graphs and applications, many graph-processing systems have been developed on top of diverse computation and storage infrastructure. Among them, GPU-enabled systems promise to significantly accelerate graph-processing [59]. Understanding their performance, for example to select, tune, and extend these systems, is very challenging. Previous studies [52, 58] have investigated the performance of popular CPU-based distributed systems, such as Giraph [43], GraphLab [87], and Hadoop [141]. However, few of them include GPU-enabled systems. To address this problem, in this chapter we conduct the first comprehensive assessment of GPU-enabled graph-processing systems (including GPU-only and hybrid CPU and GPU systems).

We have identified three dimensions of diversity that complicate the performance evaluation of graph-processing systems in our previous work [52]. *Dataset diversity* originates from the variety of application areas for graph processing, from genomics to social networks, from citation databases to online games, all of which may create unique graph structures and characteristics. *Algorithm diversity* is the result of the different insights and knowledge that users and analysts want to gain from their graphs—a large number of algorithms have been developed for calculating basic graph metrics, for searching for important vertices, for detecting communities, etc. *System diversity* derives from the uncoordinated effort of different groups of developers who try to solve specific graph-processing problems, while tuning for their existing hardware infrastructure. Many graph-processing systems have appeared in recent years, from single-node systems such as GraphChi [80] and Totem [41] to distributed systems such as Giraph [43]; from generic systems such as Hadoop [141] to graph specific systems such as GraphX [46]; from CPU-based systems such as GraphLab [87] to *single-node* GPU-enabled systems such as Medusa [152] and MapGraph [39]. (To date, no mature, distributed graph-processing system using GPUs is publicly available.)

Understanding the performance of single-node GPU-enabled systems is important for two main reasons. Firstly, we argue that many datasets already fit to be processed in-memory on such systems. This corresponds, for example, to the datasets in use at many Small and Medium Enterprise (SMEs), and thus may affect up to 60% of the entire industry revenue [29]. Secondly, single-node systems are representative for, performance-wise, and the basic building block of future GPU-clusters for graph processing.

Understanding the performance of graph-processing systems is difficult. There is no analytical approach to understand their performance comprehensively. Thus, experimental performance evaluation studies [52, 58] have been recently proposed. However, they do not cover GPU-enabled systems. Moreover, many new challenges, such as different formats of in-memory graph representations, many optional optimization techniques provided by GPU-enabled systems, and different types of equipped GPUs, make it challenging to thoroughly understand the performance of GPU-enabled systems.

Our vision [57] is a four-stage empirical method for the performance evaluation of any graph-processing system. In this chapter, we extend our previous method [52] for evaluating graph-processing systems to include GPU-enabled systems. We define several new performance metrics to comprehensively evaluate the interesting performance aspects of GPU-enabled graph-processing systems—raw processing power, performance breakdown, scalability, the impact on performance of system-specific optimization techniques and of the GPU generation. We then conduct experiments, by implementing 3 typical graph algorithms and selecting 9 datasets with different structures, on 3 GPU-enabled systems—Medusa, Totem, and MapGraph. Our main contributions are:

1. We propose a method for the performance evaluation of GPU-enabled graph-processing systems (Section 4.2). This method extends significantly our previous work for evaluating the performance of graph-processing systems, by defining new performance aspects and metrics, and by selecting new datasets and algorithms.

2. We demonstrate how our method can be used for evaluating and comparing GPU-enabled systems in practice. We setup comprehensive experiments (Section 4.3), which we then conduct for three GPU-enabled graph-processing systems (Section 4.4). Last, we also identify, for these systems, various highlights and limitations (Section 4.5).

4.2 Extended Method for GPU-Enabled Systems

Our previous method for the performance evaluation of graph-processing systems [52] consists of four main stages: identifying interesting and important performance aspects and metrics; defining and selecting workloads with representativeness and coverage; implementing, configuring, and executing the experiments; and analyzing and presenting results in standard format. To address the challenges of the performance evaluation of GPU-enabled systems, in this section we adapt and extend our previous method, by identifying new performance aspects and metrics, and by selecting and including new datasets and algorithms.

4.2.1 Performance Aspects, Metrics, Process

To understand the performance of GPU-enabled systems, we identify five important performance aspects: three of them are adapted from our previous work, and two are newly designed relative to our previous work. For each aspect, we use at least one performance metric to quantify and characterize system performance. We further adapt for GPU-enabled systems the process for measuring and calculating the metrics.

We consider five performance aspects in this chapter:

1. *Raw processing power* (adapted from previous work): reflects the user perception of how quickly graphs are processed. We report in this chapter the run time of the algorithm for GPU-enabled systems.

2. *Performance breakdown* (adapted from previous work): the algorithm run time is not sufficient to understand all the details of system performance. Breaking down

the total execution time into separate processing stages (system initialization time, algorithm run time, and data transfer time) enables the in-depth comparison of systems and, possibly, the identification of bottlenecks.

3. *Scalability* (adapted from previous work): the ability of a system to maintain its performance behavior when resources are added to its infrastructure. In our method, we test the vertical scalability (by adding GPUs) of systems in both strong and weak scaling. The observed changes in performance depend on both the number of added GPUs, and the algorithm and dataset. Indirectly, scalability allows us to reason about how well do graph-processing systems utilize accelerators.

4. *The impact on performance of system-specific optimization techniques* (newly designed): systems that can use different types of computing resources, for example both CPUs and GPUs, allow programmers to provide different implementations of the same algorithm, optimized for the different hardware. We study the impact of such optimizations (e.g., load balancing, graph representation), to understand their impact on system performance.

5. *The impact on performance of the GPU generation* (newly designed): several GPU generations are currently present in computing infrastructures, from mobile devices to servers and clusters. Understanding the correlation between their characteristics (compute capability, the number of cores, and memory capacity) and the system performance could guide users towards optimal (cost, performance) choices for their applications.

We summarize in Table 4.1 the performance metrics used to quantify the five performance aspects. For each of the metrics, we define how it can be obtained: by direct measurement or by calculation using measured parameters and dataset (i.e., graphs) properties. We define the total execution time (T_E) as the time from submission until completion. For each submission, we do not write output data to disk, but transfer the output data from GPUs to host memory. Algorithm run time (T_A) is the time used for actually executing the graph algorithms. Total execution time can be divided into times for different processing stages of the whole execution, including graph and configuration initialization time (T_I), algorithm run time (T_A), data transfer time from device to host (T_{D2H}), and overhead time (T_O) which includes the overhead in the initialization stage and the clear up stage. T_I can be further split into (1) graph initialization time (T_{I-G}, which includes reading and building graph in the host memory and transferring the graph data from host to device (T_{H2D})) , and (2) algorithm configuration time (T_{I-C}, which includes setting up algorithm-related configuration parameters and initial values in GPUs). We formulate the relationship of the total execution time and its breakdown as follows:

$T_E = T_{I-G} + T_{I-C} + T_A + T_{D2H} + T_O$

Table 4.1: Summary of performance metrics used in this study.

Metric	**How measured?**	**Derived**	**Relevant aspect (use)**
Total execution time (T_E)	Total time of the full execution	-	Raw processing power (Table 4.6)
Algorithm run time (T_A)	Time of the algorithm running	-	Raw processing power (Figure 4.1, 4.2, 4.3)
Time breakdown	Time of the detailed execution	-	Performance breakdown (Table 4.6)
Strong scaling	T_A of multiple GPUs (N) for the same graph	-	Scalability (Figure 4.4)
Weak scaling	T_A of multiple GPUs (N) for different graphs	-	Scalability (Figure 4.6)
Normalized edges per second (NEPS)	-	$\#E/T_A/N$	Scalability (Figure 4.5)
Speedup	-	T_A/T'_A	Optimization, GPUs (Figure 4.7, 4.8, 4.9)

$\#E$ is the number of all edges of the executed algorithm. T'_A is the algorithm run time of a different setup.

We define Edges Per Second (EPS) as the ratio between the number of all edges of the executed algorithm and the algorithm run time. EPS is a straightforward extension of the TEPS metric used by Graph500 [48]. To investigate the performance per computing unit, we further define the performance metric Normalized Edges Per Second (NEPS) as the ratio between EPS and the total number of computing units (GPUs in this chapter). For the same algorithm running on the same dataset, but with different setups (optimization techniques and GPU generations), we define the speedup as the ratio between the algorithm run time of baseline (see Section 4.4.4 and 4.4.5 for our baseline settings) and that of a different setup.

4.2.2 Selection of Graphs and Algorithms

In this section, we discuss our selection of graphs and algorithms, which we used to evaluate the GPU-enabled graph-processing systems.

Graph Selection

We select nine different graphs and summarize their information in Table 4.2. We select both directed (Amazon, WikiTalk, and Citation) and undirected graphs (KGS, DotaLeague, Scale-22 to Scale-25). To comply with the requirement of many GPU-enabled systems, we need to store input undirected graphs in a directed manner. Thus, for each undirected edge, we create two directed edges as an equivalent. The selected graphs match well to the datasets used by SMEs in terms of scale and diversity. The graphs are

Table 4.2: Summary of datasets used in this study.

Graphs	**V**	**E**	**d**	**$\bar{\text{D}}$**	**Max D**
Amazon (D)	262,111	1,234,877	1.8	5	5
WikiTalk (D)	2,388,953	5,018,445	0.1	2	100,022
Citation (D)	3,764,117	16,511,742	0.1	4	770
KGS (U)	293,290	22,390,820	26.0	76	18,969
DotaLeague (U)	61,171	101,740,632	2,719.0	1,663	17,004
Scale-22 (U)	2,394,536	128,304,030	2.2	54	163,499
Scale-23 (U)	4,611,439	258,672,163	1.2	56	257,910
Scale-24 (U)	8,870,942	520,760,132	0.7	59	406,417
Scale-25 (U)	17,062,472	1,047,207,019	0.4	61	639,144

V and **E** are the vertex count and edge count of the graphs. **d** is the link density ($\times 10^{-5}$). **$\bar{\text{D}}$** is the average vertex out-degree. **Max D** is the largest out-degree. (D) and (U) stands for the original directivity of the graph. For each original undirected graph, we transfer it to directed graph (see Section 4.2.2).

from diverse sources (e-business, social networks, synthetic graphs), and different characteristics (e.g., high vs. low average degree, directed and undirected graphs). The Scale-22 to Scale-25 are undirected graphs created by the Kronecker generator introduced in Graph500 [48], with the scale from 22 to 25 and edge factor of 16. The other graphs have been collected from real-world applications, and have been shared through the Stanford Network Analysis Project (SNAP) [121]) and the Game Trace Archive (GTA) [54].

Algorithm Selection

Considering the simplicity of the programming model of several GPU-enabled systems, we avoid algorithms using complex messages and mutating graph structures. Based on our comprehensive survey of graph-processing algorithms and applications [51], we select BFS, PageRank, and WCC as representative for three popular algorithms classes: graph traversal (used in Graph500), general statistics, and connected components, respectively. We summarize the characteristics of these algorithms in Table 4.3.

Breadth First Search (BFS) is a widely used algorithm in graph traversal. BFS can be used as a building block for more complex algorithms, such as all-pairs shortest path and item search. BFS is a textbook algorithm. The PageRank algorithm (PageRank) is originally designed to rank websites in search engines. It can also be used to compute the importance of vertices in a graph. Several versions of PageRank have been proposed. In this chapter, we use the version described in Medusa [152]. Weakly Connected Component (WCC) is an algorithm for extracting groups of vertices connected via graph edges. For directed graphs, we say a group of vertices is weakly connected if any vertex in this group can be linked by an edge (no matter the direction) to another vertex in this group. We select in this chapter an implementation of WCC created by Wu and Du [144].

Table 4.3: Summary of algorithms used in this study.

Algorithm	Main features	Use
BFS	iterative, low processing	building block
PageRank	iterative, medium processing	decision-making
WCC	iterative, medium processing	building block

Table 4.4: Summary of systems used in this study.

System	Version	Type	Release date
Medusa	Medusa-0.2	Multiple GPUs	2013-02
Totem	Trunk version	Hybrid, multiple GPUs	2014-08
MapGraph	MapGraph 0.3.2	Single GPU	2014-04

4.3 Experimental Setup

In this section, we make a selection of GPU-enabled graph-processing systems, discuss the implementation of the graph-processing algorithms on the selected systems, and set the configuration of the parameters for running the algorithms.

4.3.1 System Selection

Compared with the number of CPU-based graph-processing systems, there are fewer single-node GPU-enabled systems, and no distributed GPU-enabled graph-processing systems available for the public. Thus, in this chapter, we select three of the most mature single-node GPU-enabled graph-processing systems: Medusa, Totem, and MapGraph. Table 4.4 summarizes our selected systems. We introduce each system in the following.

Medusa [152] is a graph-processing framework designed to help programmers use the GPU computing power with writing only sequential code. To achieve this goal, Medusa provides a set of user-defined APIs to hide the GPU programming details. Medusa can support multiple GPUs. Medusa extends the Bulk Synchronous Parallel (BSP) model by applying a "Edge-Vertex-Message" (EMV) model for each superstep. The EMV model breaks down the vertex-centric workload into separate chunks; the key concepts related to a chunk are vertices, edges, and messages. Compared with a vertex-centric programming model, the fine-grained EMV model can achieve better workload balance of threads [152]. Medusa supports four different formats to store graphs in-memory: the vertex-oriented format Compressed Sparse Rows (CSR, or AA used in [152]), the edge-oriented format (ESL), the hybrid format (HY) of CSR and ESL, and the column-major adjacency array (MEG, or CAA used in [152]). HY and MEG are designed to reduce the uncoalesced memory access on GPUs. A graph-aware message buffer scheme is designed by Medusa to achieve better performance of processing messages between vertices. To maintain this

Table 4.5: Experimental setup for each experiment in Section 4.4.

Section	Systems	Algorithms	Datasets	Metrics	GPU (Machine Type)	Graph Formats
4.4.1	All	All	6	Algorithm run time	GTX 480 (Type 1)	CSR
4.4.2	All	All	2	Total execution time and its breakdown	GTX 480 (Type 1)	CSR
4.4.3	Medusa, Totem	PageRank	4	Strong and weak scaling, NEPS	GTX 580 (Type 2)	CSR
4.4.4	All	PageRank	6	Speedup	GTX 480 (Type 1)	CSR, HY, MEG
4.4.5	All	All	6	Speedup	GTX 480, GTX 580, K20m (Type 1, 3, 4)	CSR

buffer, a message index needs to be stored for each edge. For multiple GPUs, Medusa provides a multi-hop replication scheme and overlapping of computation and communication to alleviate the pressure from data transfers between partitions.

Totem [41] is a graph-processing system that can leverage both the CPU and the GPU (hybrid) as computing units by assigning graph partitions to them. Totem can also support multiple GPUs (with or without the CPU). Totem uses a vertex-centric programming abstraction under the BSP model. Each superstep of the BSP model includes three phases: a computation phase in which each computing unit executes the algorithm kernel on its assigned partitions, a communication phase in which each computing unit exchanges messages, and a synchronization phase to deliver all messages. Totem strictly uses CSR to represent graphs in-memory. To alleviate the cost of communication between partitions, Totem uses user-provided aggregation to reduce the amount of messages and maintains two sets of buffers on each computing unit for overlapping communication and computation. Totem implements other optimizations to improve the performance, for example, partitioning graphs by the vertex degrees and placing higher-degree vertices on CPU.

MapGraph [39] is an open-source project to support high-performance graph processing. The latest version (see Table 4.4) of MapGraph can only support a single GPU. MapGraph uses a modified Gather-Apply-Scatter model [87] to present each superstep of graph-processing algorithms. In the Gather phase, vertices collect updated information from in-edges and/or out-edges. During the Apply phase, every active vertex in the current superstep updates its value. In the Scatter phase, vertices send out messages to their neighbors. For each superstep, MapGraph maintains an array called frontier which consists of active vertices, to reduce the computation. The frontier for the next superstep is created in the Scatter phase of the current superstep. MapGraph uses CSR and the Compressed Sparse Column format (CSC [93], which is a reverse topology index of CSR) to store graphs. MapGraph adapts two strategies, dynamic scheduling and two-phase decomposition, to balance the workload of different threads.

Notation: From hereon, we use `M` for Medusa, `T-H` for Totem in hybrid mode using both the CPU and the GPU, `T-G` for Totem using the GPU(s) as the only computing resource, and `MG` for MapGraph.

4.3.2 System and Experiment Configuration

Hardware: We perform our experiments on DAS4 [21], which is a cluster with many different types of machines to cater different computation requirements of researchers in the Netherlands. We use four types of machines from DAS4 to conduct our experiments. Type 1 includes an Nvidia GeForce GTX 480 GPU (1.5 GB onboard memory) and an Intel Xeon E5620 2.4 GHz CPU. Type 2 is equipped with 8 Nvidia GeForce GTX 580 GPU (3 GB onboard memory per GPU) and dual Intel Xeon X5650 2.66 GHz CPUs. Type 3 consists of an Nvidia GeForce GTX 580 GPU (3 GB onboard memory) and an Intel Xeon E5620 2.4 GHz CPU. Type 4 has an Nvidia Tesla K20m GPU (5 GB onboard memory) and an Intel Sandy Bridge E5-2620 2.0 GHz CPU. The machines are used for different experiments as shown in Table 4.5.

Algorithms: Some of the algorithms belong to libraries distributed with the systems, but the programming details may be different. When this is the case, we select a unique implementation, as described in Section 4.2.2. For each algorithm, we set the parameters identically on all systems. For BFS, we use the same source vertex for each graph on all systems. For PageRank, we consider maximum iteration as the only termination condition and set maximum iteration to 10 times. WCC does not need any specific parameter configuration.

System tuning: Several configuration parameters could be tuned in each system. The tuning of parameters can change the performance of these systems. We explore the influence of several common techniques for system tuning in Section4.4.4. For the other experiments, we use the default settings of each system. For example, for the hybrid mode of Totem, we place on the CPU higher-degree vertices, that is, the vertices whose total degree is about one third of the number of edges of the whole graph.

Dataset considerations: Because the WCC algorithm considers that two vertices are connected when there is an edge between them, when running the WCC algorithm for *directed* graphs (Amazon, WikiTalk, and Citation), we create a reverse edge for each pair of vertices which are originally connected by a single directed edge. The new datasets are *AmazonWCC*, *WikiTalkWCC*, and *CitationWCC*, with the number of edges 1,799,584, 9,313,364, and 33,023,481, respectively.

Further configuration and settings: For all systems, the GPU compiler is Nvidia CUDA 5.5. We use CUDPP 2.1 [19] and Intel TBB 4.1 [65] as third-party libraries for Medusa and Totem, respectively. We repeat each experiment 10 times, and we report the arithmetic mean. We only show error bars in our scalability test, because in all the other experiments our results from 10 runs are very stable, with the largest variance under 5%.

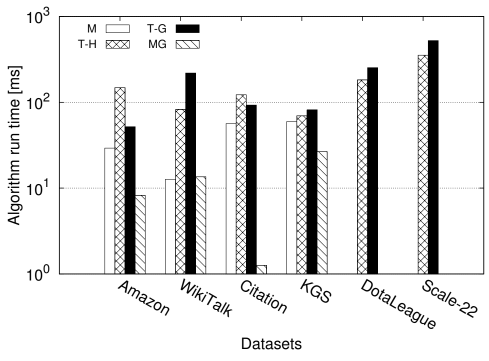


Figure 4.1: The algorithm run time for BFS of 6 datasets on all systems. (Missing bars are explained in text.)

4.4 Experimental Results

In this section we present our experimental results. Table 4.5 summarizes our experimental setups. The experiments we have performed are:

- Raw processing power (Section 4.4.1): we have measured the algorithm run time and we report it for all combination of algorithms, datasets, and systems.
- Performance breakdown (Section 4.4.2): we have analyzed the total execution time in detail. We show the breakdown of the total execution time as introduced in Section 4.2.
- Scalability (Section 4.4.3): we have measured the vertical scalability of Totem and Medusa in both strong scaling and weak scaling.
- The impact on performance of system-specific optimization techniques (Section 4.4.4): we have applied system-specific optimization techniques and we report the impact they have on the performance of systems.
- The impact on performance of the GPU generation (Section 4.4.5): we have investigated the behavior of all three systems on three different generations of GPUs and we report the performance changes we have observed.

4.4.1 Raw Processing Power: Algorithm Run Time

In this section, we reported a full set of experiments (all algorithms, all systems, and 6 datasets) and analyze the algorithm run time.

Key findings:

- Totem is the only system that can process all 6 datasets for all algorithms. Medusa and MapGraph crash for some of the setups.
- There is no overall best performer, but in most cases Totem performs the worst.

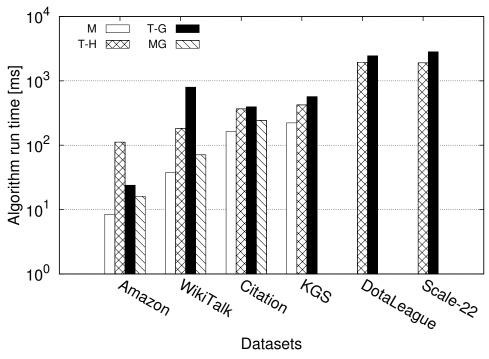


Figure 4.2: The algorithm run time for PageRank of 6 datasets on all systems. (Missing bars are explained in text.)

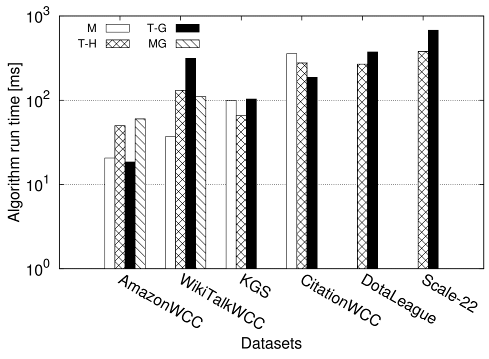


Figure 4.3: The algorithm run time for WCC of 6 datasets on all systems.(Missing bars are explained in text.)

- The optimization techniques used by the graph-processing systems lead to inconsistent performance benefits across different algorithms.
- Relative to the performance we have observed on CPU-based systems [52], the results of the GPU-enabled systems studied in this chapter are significantly faster.

We report results for the NVIDIA GTX 480 GPU, which fits all real-world datasets and the Scale-22 synthetic one (experiments on larger datasets are discussed in Section 4.4.3). We show the algorithm run time, for each combination of setup parameters, in Figures 4.1, 4.2, and 4.3. For the horizontal axis of each figure, we order the datasets by their number of edges, from left to right.

We depict the performance of BFS in Figure 4.1. We see that only Totem can handle all 6 datasets. Medusa and MapGraph crash attempting to construct DotaLeague and Scale-22 in-memory and report an “out of memory” error. Although in these experiments, all systems use the CSR format to store graphs in-memory, the implementation details are different. Totem strictly represents the graph in the CSR format by using two arrays V and E: array V contains the start indices that can be used to fetch the neighbor lists, which are stored in array E. Medusa uses a structure of arrays, which includes extra data such as the number of edges for each vertex and the message index for each edge. In MapGraph, a graph is represented in both the CSR format and the CSC format in-memory. The usage of CSC doubles the memory consumption.

In Figure 4.1, the two modes of Totem have longer run times than either Medusa or MapGraph. An important reason for this is the different parallelism granularities of the kernel. In Totem, the number of threads is the same as the number of vertices and each thread processes one vertex [59]. This mapping can result in workload imbalance because every thread's workload is skewed by the degree of its assigned vertex. The performance of Totem worsens when the vertex degrees are highly skewed, as seen for example, for running T-G on the WikiTalk dataset (the degrees of most vertices in WikiTalk are smaller than 10K, but there is one vertex with more than 100K neighbors). This imbalance can be alleviated by assigning these higher-degree vertices to the CPU, as shown by the result of T-H on Wikitalk. In Medusa, the whole workload is divided into three phases which target individual vertices, edges, and messages. Medusa uses a fixed amount of blocks and threads per block to process all vertices and all edges. The workloads of threads are well balanced by assigning vertices and edges to threads in turn. MapGraph adapts complex dynamic scheduling and two-phase decomposition to balance the workload of threads.

For the BFS algorithm (Figure 4.1), MapGraph performs best for all the graphs it can handle. We attribute this advantage to the design of a *frontier* which maintains active vertices for each superstep. For the BFS algorithm, the active vertices in each superstep can be significantly less than the full set of vertices. Thus, for each iteration of BFS on MapGraph, only a part of vertices are accessed and computed. Although Medusa and Totem do not compute non-active vertices, both systems have to access all vertices. The impact of the frontier is significant when the set of active vertices is small. For example, the algorithm run time of Citation (BFS traverse coverage is 0.1%) is much shorter than other datasets (BFS traverse coverages are greater than 98.5%). However, the implementation of a frontier may have negligible impact for other algorithms, and may even cause crashes of the system due to lack of memory.

For PageRank (see Figure 4.2), MapGraph cannot outperform Medusa with any dataset because all vertices are active in each superstep (according to our implementation of this algorithm, see Section 4.2.2). This is in contrast to our findings for BFS (Figure 4.1) regarding the impact of the frontier. Furthermore, MapGraph cannot process KGS because it doest not have enough memory for maintaining such a larger frontier.

The comparison of the algorithm run time of WCC on three systems is shown in Figure 4.3. Unlike the results of BFS and PageRank, the performance of Totem is not always worse than Medusa and MapGraph; Totem exhibits better performance in both hybrid mode and GPU-only mode on AmazonWCC, KGS, and CitationWCC. There are three main reasons that lead to this result. Firstly, a large amount of updated information needs to be send between supersteps of WCC. Totem aggregates the updated information sent to the same vertex that can reduce the inter-superstep communication time. Secondly,

Table 4.6: The breakdown of running the BFS algorithm on the Amazon dataset and the WCC algorithm on the AmazonWCC dataset. (All time values in milliseconds.)

	BFS on Amazon				**WCC on AmazonWCC**			
	M	MG	T-H	T-G	M	MG	T-H	T-G
T_{I-G}	1278.1	1064.9	339.1	316.6	1787.1	1519.0	477.8	452.4
T_{I-C}		0.7	0.5	0.3		0.2	1099.9	1062.0
T_{H2D}	8.1	3.3	1.1	1.8	10.3	4.3	1.5	2.3
T_A	29.3	8.2	148.0	51.9	20.6	59.2	49.9	18.5
T_{D2H}	1.5	0.6	0.6	0.4	1.5	0.4	0.7	0.8
T_O	723.7	18.2	3.7	46.5	739.5	18.0	4.1	46.4
T_E	2032.5	1092.7	491.9	415.7	2548.8	1596.9	1632.5	1580.2

the distribution of vertex degrees of AmazonWCC, KGS, and CitationWCC is not highly skewed. Thirdly, the computation of the algorithm is not intensive. The second and third reasons result in a relatively balanced workload for each thread.

From Figures 4.1, 4.2, and 4.3, we also find that the structure of graphs have consistent impact on the algorithm run time of all three algorithms in two modes of Totem. For instance, the algorithm run time of T-H is always more than that of T-G on Amazon, while the algorithm run time of T-H is always less on WikiTalk, no matter which algorithm runs.

4.4.2 Performance Breakdown

In this section, we report, for each algorithm, the total execution time and its breakdown.

Key findings:

- The time for reading the graph and for constructing the graph in-memory dominates the total execution time.
- The initialization time of the systems should be reduced.

Because the performance breakdown of PageRank is very similar to that of BFS, we show the breakdown of the total execution time on Amazon for just BFS and WCC in Table 4.6. Note that the input file of WCC is AmazonWCC, making the graph initialization time T_{I-G} for all systems longer than that of BFS on Amazon.

Overall, we notice that for all algorithms, the initialization time (including T_{I-G} and T_{I-C}) is the major part of the total execution. Thus, the performance of initialization is essential for improving the overall performance of such systems. Compared with Totem and MapGraph, Medusa needs more time for initialization because it reads the input file by words, not by lines, and even if a graph is not partitioned, the data structures for building partitions are created and calculated. Medusa also shows longer overhead, mainly caused by the initialization of system, such as configuring the L1 cache and shared memory of the GPU. The graph initialization and algorithm configuration in Medusa are aggregated.

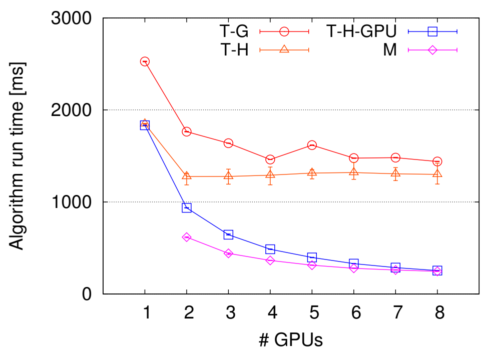


Figure 4.4: The strong scaling of Totem and Medusa on Scale-22. (The missing point for Medusa, # GPUs =1, is explained in Section 4.4.1.)

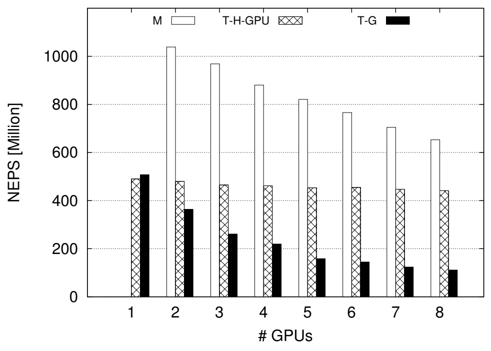


Figure 4.5: The normalized edges per second of processing Scale-22. (The missing bar for Medusa, # GPUs =1, is explained in Section 4.4.1.)

MapGraph uses a two-step procedure to build CSR and CSC formats in-memory. A graph input file is read into the Coordinate (COO) format [93], the tuples in COO are then sorted and the CSR and CSC formats are constructed from COO. However, when algorithms do not need the CSC format for execution, the time for building the CSC format is wasted.

For the WCC algorithm, each vertex is assigned an initial value using its vertex ID. However, in the hybrid mode of Totem, input graphs are partitioned and all vertices are re-assigned to new IDs in each partitions. Each partition keeps a map for mapping new IDs in this partition to original IDs in the input graph. Thus, for the configuration of WCC, Totem needs to access the map once for initializing the value of each vertex. Totem does not support special mechanism for initializing one partition graph, thus the configuration time of the GPU-only mode is as high as that of the hybrid mode. Medusa faces the same long configuration problem when it uses multiple GPUs.

4.4.3 Evaluation of Scalability

In this section, we evaluate the scalability of Totem and Medusa with using multiple GPUs.

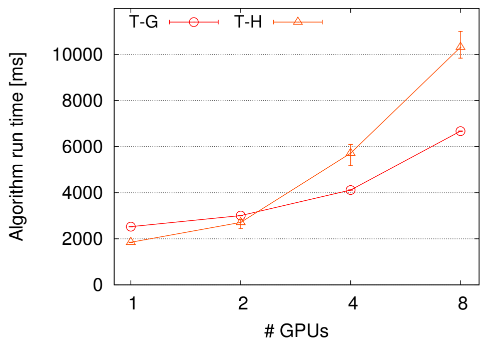


Figure 4.6: The weak scaling of Totem.

Key findings:

- Totem and Medusa show reasonable strong and weak scaling with the increase of the number of GPUs.
- Increasing the number of GPUs may not always lead to performance improvement.

All experiments in this section are executed on a machine with 8 Nvidia GTX 580 3GB GPUs. We assign vertices randomly to GPUs. Our scalability tests are using PageRank because it is the most compute-intensive algorithm in this chapter as shown in Section 4.4.1. We test both strong scaling and weak scaling. For strong scaling, we use the Scale-22 graph which is the largest graph that can be handled by Totem with only one GPU. We increase the number of GPUs from 1 to 8. The strong scaling results for T-H include both the CPU workload (constant) and the GPUs workload (scaled). For weak scaling, we use four generated graphs from Scale-22 to Scale-25, and test them on 1, 2, 4, and 8 GPUs.

For strong scaling, we observe that Medusa exhibits better scalability than Totem (Figure 4.4). The algorithm run time of Medusa keeps decreasing by adding more GPUs. For Totem, in both T-H and T-G, the algorithm run time does not change too much after a certain number of GPUs. However, due to the random placement of vertices, the workload per GPU may not be balanced. Combined with the increasing time for communication, the decreasing trend of the algorithm run time is not obvious with adding GPUs. As we discussed in Section 4.4.1, the workload of each thread on a GPU is not well balanced for Totem, which may cause the bad scaling after using 4 GPUs in T-G. As for T-H, because the algorithm run time is dominated by the CPU computation, when using 2 GPUs or more, the performance remains almost constant. This can be proved by the longest algorithm run time of all GPUs (represent as T-H-GPU). We also notice that T-H shows more unstable behavior (note the error bars), which indicates that the algorithm execution on the GPUs is more stable than on the CPU.

We show in Figure 4.5 the normalized edges per second (NEPS) for Medusa and Totem. To compute NEPS, we first calculate EPS by dividing the algorithm run time by the number of edges accessed by PageRank, then we normalize EPS by the number of GPUs. For the hybrid mode of Totem, we only consider the scaled workload on GPUs, see T-H-GPU in Figure 4.5. We compute the EPS of T-H-GPU using the longest algorithm run time of all GPUs and the number of edges placed on GPUs (according to the configuration in Section 4.3, about two thirds of edges are processed on GPUs). The NEPS of T-H-GPU remains relatively constant at around 450 million. Because the higher-degree vertices are mainly assigned to the CPU, the degrees of the vertices placed on the GPUs are not vary, leading to a relatively balanced workload for each GPU and for each thread on a same GPU.

The weak scaling of T-G is presented in Figure 4.6. We do not have results for Medusa because the experiments using Scale-22 to Scale-25 crashed due to the high memory consumption. T-H is only shown as a reference to T-G: due to the use of both a single CPU (workload is not scaled) and multiple GPUs (workload is scaled). The efficiency of weak scaling of T-G is 84%, 61%, and 38%, for using 2, 4, and 8 GPUs, respectively. The efficiency decreases because the total workload does not grow linearly with the increase of the graph scale, and this may also because the vertex degrees of larger graphs are more skewed (Table 4.2).

4.4.4 Evaluation of System-Specific Optimization Techniques

Many optimization techniques can be used to change the performance of the graph-processing systems studied in this chapter. In this section, we evaluate the performance of Medusa when storing the graph in-memory using different representations, the performance of Totem when using the different configurations for the virtual warp-centric technique [62], and the performance of MapGraph when using different thresholds for choosing its scheduling strategies.

Key findings:

- Performance improvements depend significantly on system-specific optimization techniques.
- Advanced techniques may not always have a positive impact of the performance on systems.

We choose PageRank because it is the most compute-intensive algorithm in our study. To focus on the performance of the GPU, we do not report result from T-H.

Medusa can support several graph representations, CSR, ESL, hybrid (HY) format of CSR and ESL, and MEG. HY requires a threshold value to calculate the proportion of CSR and ESL. Figure 4.7 shows the speedup of building graphs in the MEG format (M-MEG) and the HY format (with the threshold of 4 and of 16, represented as M-HY-4 and M-HY-16, respectively). The threshold values for M-HY are the default value (16) and

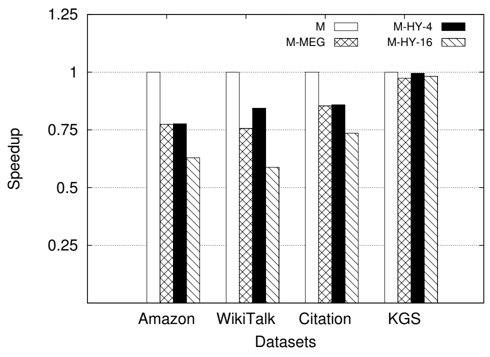


Figure 4.7: The speedup of different graph representations on Medusa.

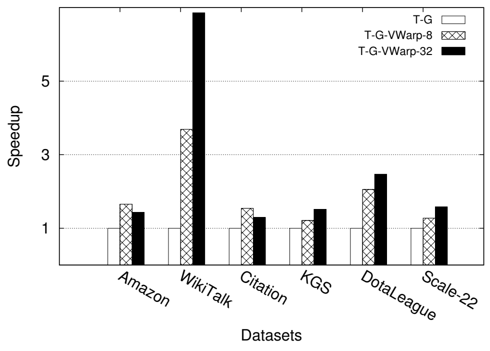


Figure 4.8: The speedup of using different virtual warp-centric setups on Totem.

representative for the average vertex degree (4). We set the performance of Medusa using the CSR format as the baseline. It is very surprising that all MEG and HY experiments have worse performance than the CSR format. This result is different from the result of [152] with running PageRank on a RMAT dataset on Medusa, in which MEG and HY are better. We cannot directly compare our results because as we shown in our previous work [52], results can be very sensitive to datasets. M-HY-4 significantly outperforms M-HY-16 on Amazon, WikiTalk, and Citation because the threshold is closer to their average vertex degrees, which is similar to the result of [152].

To improve the performance of Totem, we use the virtual warp-centric technique to balance the workload of threads in the algorithm kernel. Figure 4.8 illustrates the speedup of the algorithm run time by using virtual warp (with virtual warp size of 8 and of 32, represented as T-G-VWarp-8 and T-G-VWarp-32, respectively). The warp size values are the default value (32) and representative for the graph properties (8). We set the performance of T-G as the baseline. From Figure 4.8, we notice that both T-G-VWarp-8 and T-G-VWarp-32 can obtain significant improvement, with the highest speedup of around 7. T-G-VWarp-8 has better performance than T-G-VWarp-32 on Amazon and Citation, whose average degrees are closer to 8. This finding matches to the result of the work of [62] with running BFS on several datasets.

Table 4.7: Summary of the GPU generations used in this study. SP/DP denote single/double precision operations.

	GTX 480	**GTX 580**	**K20m**
Frequency (GHz)	1.40	1.57	0.71
# Cores	480	512	2496
Peak GFLOPS (SP/DP)	1344.0/672.0	1603.6/801.8	3519.3/1173.1
Memory capacity (GB)	1.5	3	5
Peak Memory Bandwidth (GB/s)	177.4	193.0	208.0

We have also investigated the performance of tuning MapGraph. In the scatter phase, dynamic scheduling and two-phase decomposition can be used to create the frontier of the next superstep. MapGraph uses a threshold on the frontier size of the current superstep to determine which strategy would be executed. We tune the threshold for running PageRank on all datasets with various values (from 1 to 20000, default value 1000). The algorithm run time is not sensitive to the threshold, with a variance within 2% for different thresholds on the same dataset and algorithm. We further check the threshold influence on BFS and WCC, and we get the same result as PageRank.

4.4.5 Evaluation of the Impact of the GPU Generation

In this section, we run all experiments of Section 4.4.1 on two other GPUs: GeForce GTX 580 and Tesla K20m.

Key findings:

- Memory consumption is a key issue of Medusa and MapGraph, when processing the largest graphs in our study.
- Using a GPU with improved processing capability can help, but not always.

The details of GPUs are introduced in Table 4.7. We compare the performance of systems deployed on different generations of GPUs. We present a representative selection of results from the whole set of experiments. To focus on the performance impact of the GPU generation, we do not report results from the hybrid mode of Totem in this section.

Table 4.8 shows, for WCC, the change of processable datasets of Medusa and MapGraph on GTX 480, GTX 580, and K20m. We choose the WCC algorithm because on GTX 480, it has the most number of datasets that cannot be processed on Medusa and MapGraph. We use "Y" to depict a dataset that can be processed and "N" for crashes. For each crash, we present the reason why it happens. "F" represents a crash that occurs when the graph cannot **F**it into the GPU memory at the initialization time. "R" represents a crash that happens during the **R**un time of WCC. For each R, we further detail if it is caused by out of **M**emory or CUDA **E**rror. From Table 4.8, we find that more datasets can

Table 4.8: The success of running the WCC algorithm on Medusa (M) and MapGraph (MG). ("Y" denotes successful processing. All other values denote crashes, see text for details.)

		GTX 480	GTX 580	K20m
KGS	M	Y	Y	Y
	MG	N (R: M)	Y	Y
CitationWCC	M	Y	Y	Y
	MG	N (R: M)	N (R: M)	Y
DotaLeague	M	N (F)	N (R: E)	N (R: E)
	MG	N (F)	N (R: M)	N (R: M)
Scale-22	M	N (F)	N (F)	N(R: E)
	MG	N (F)	N (F)	N (R: M)

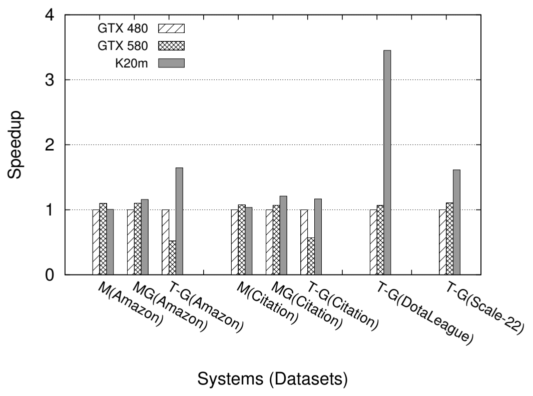


Figure 4.9: The speedup of running PageRank using different GPU generations.

be processed by using GPUs with larger memory. For example, MapGraph can handle KGS and CitationWCC on K20m. We also observe that DotaLeague and Scale-22 still cannot be processed by Medusa and MapGraph. For DotaLeague and Scale-22 on MapGraph, all crashes are caused by out of memory, either in initializing the graph or running the algorithm with large frontier. For Medusa, although memory is not the bottleneck on K20m, CUDA errors are reported by its library CUDPP 2.1 [19].

In Figure 4.9, we report the obtained speedup for every system when using different GPU generations. We set the performance on GTX 480 as a baseline and we pick the PageRank algorithm because it has the longest algorithm run time in our study. For the datasets, we choose Amazon (smallest dataset), Citation (the largest dataset that can be processed by all systems on GTX 480), DotaLeague (the largest real-world dataset), and Scale-22 (synthetic graph). In most cases, the performance we observed increases when using more powerful GPUs. For Medusa and MapGraph, the performance improvement is not significant. We even find performance degradation: for instance, T-G's performance on GTX 580 is worse than that on GTX 480. Overall, we note that simply migrating systems to more powerful GPUs may not be sufficient to obtain much higher performance, as we would expect. To get better performance, additional parameter tuning may be performed.

4.5 Qualitative Analysis of User Experience

Performance is a key issue for selecting systems. From the perspective of an end user, the usability of the systems is also important. In this section, we discuss our user-experience with each of the selected system. We also compare the experience of using GPU-enabled systems with our prior experience as users of CPU-based systems.

For Medusa, as the designers of Medusa claim, we could use the fewest lines of code, relative to the other systems considered in this study, to implement the core part of the algorithms. However, it takes more lines of code and more effort to design the data structure for each of vertices, edges, and messages. The graph representation and the design of the message buffer are not memory-efficient, which limit the scale of the graphs that can be processed by Medusa. The errors reported by the third-party messaging library (see Section 4.4.5) reveal that Medusa needs further validation with this library. Learning how to use Medusa is not easy, because the documentation is scarce.

For Totem, we needed a majority of the lines of code for the core part of algorithms, because no high-level API is provided. We had to implement two different versions of every algorithm, for the CPU and for the GPU. The performance of Totem relies on the implementation of algorithms, as users have to address the details for coding the kernel for the GPU. Totem can process the largest datasets among the three systems, because it has efficient memory usage, and because it can use the storage resources of the host. The documentation about Totem is also scarce, but the clear structure of the project and a large number of algorithm examples can help users get familiar with the system better than Medusa.

MapGraph provides a set of APIs for users. Similarly to Medusa, users do not need to touch kernel-programming on the GPU. MapGraph lacks the ability to handle large datasets, as it is the most memory-consuming system in our study. MapGraph has better documentation than Medusa and Totem, but a comprehensive user tutorial is still needed. For the future, MapGraph promises to evolve towards a distributed system that can use GPU-clusters.

Compared with CPU-based systems [51], there are still many aspects to be improved in GPU-enabled graph-processing systems. We point out three main issues: the scale of graphs could be processed is rather small, primarily due to lack of memory; it is difficult to implement graph algorithms with complex message delivery and with graph structure mutation, because the GPU programming models may not be suitable for these aspects; the developer and user community is smaller and less active than for CPU-based systems.

4.6 Related Work

Motivated by the increasing practical need for graph-processing systems, many studies have focused on the performance evaluation of graph-processing systems in the past two years. Combined, this body of work compares the performance of many graph-processing systems, using many performance aspects and metrics, tens of diverse datasets, and various graph-processing algorithms and applications. However, individual studies rarely combine these desirable features of a performance evaluation study. This work complements all these previous studies with a process that focuses on a new class of systems (GPU-enabled instead of CPU-based) and reports on a more diverse set of metrics.

Elser and Montresor [28] evaluate the performance of 5 distributed systems for graph processing but use only 1 graph algorithms and only 1 performance metrics. Our previous work [52] proposes an empirical method for benchmarking CPU-based graph-processing systems and reports the performance of the systems using more metrics and broader workload concerns. Han et al. [58] use similar performance metrics to [52] and focus on a family of Pregel-like systems. Lu et al. [88] include the influence of algorithmic optimizations and the characteristics of input graphs.

Most of the previous evaluation studies were proposed on CPU-based systems. Relatively few performance evaluation studies focus on GPU-enabled systems. Most of these studies were proposed by system designers to exhibit their system, and may lack the method or bias necessary for this kind of studies. Specifically, the previous studies on the performance of GPU-enabled systems lack representative workloads [38], performance metrics [39, 152], and comparative systems [41, 76]. Our study is the first in-depth performance evaluation study of GPU-enabled graph-processing systems.

4.7 Summary

Using the capability of GPUs to process graph applications is a new promising branch of graph-processing research. A number of GPU-enabled graph-processing systems, with different programming models and various optimization strategies, have been developed, which raises the challenging question of understanding their performance. We conduct in this chapter the first comprehensive performance evaluation of GPU-enabled graph-processing systems.

This method significantly extends our previous work on the topic, by adapting performance aspects and introducing performance metrics that focus on GPU-enabled systems, by adding more datasets, and by focusing on important graph-processing algorithms. We focus on the following performance aspects: raw processing power, performance breakdown, scalability, the impact on performance of system-specific optimization techniques and of the GPU generation. We use at least one performance metric to quantify each performance aspect, such as total execution time and its breakdown to measure detailed performance, normalized metric NEPS to characterize scalability, etc. We select 9 datasets with diverse characteristics from both real-world domains and popular synthetic graph generators, up to scales of more than 1 billion directed edges. We also select 3 graph algorithms that are commonly used by the GPU graph-processing community.

We use the proposed method and report the first comprehensive performance evaluation and comparison of 3 GPU-enabled graph-processing systems, Medusa, Totem, and MapGraph. We show the strengths and weaknesses of each system and list key findings for each of our experiments. Overall, we conclude that understanding the performance of these systems can be very useful, but requires an in-depth experimental study.

Chapter 5

Designing Streaming Graph Partitioning Policies

In this chapter, we model the execution time of distributed graph-processing systems. By analyzing this model under the load of realistic graph-data characteristics, we propose a method to identify important performance issues and then design new streaming graph-partitioning policies to address them. By using three typical large-scale graphs and three popular graph-processing algorithms, we conduct comprehensive experiments to study the performance of our and of many alternative streaming policies on a real distributed graph-processing system. We also explore the impact on performance of using different real-world networks and of other real-world technical details. We further discuss how to use our results and the coverage of our model and method.

5.1 Overview

The scale of graphs is increasing rapidly in recent years, and has already exceeded the processing capabilities of single machines. Distributed graph-processing systems such as Pregel [92], GraphLab [87], and GraphX [46], have been designed and developed to process large-scale graphs by using the computation and memory capabilities of clusters. For such systems, graph partitioning is essential in achieving good performance, because it determines the computation workload of each working machine and the communication between them. Many streaming graph partitioning policies [126,133,146] have been proposed to efficiently partition graphs into balanced pieces for distributed graph-processing systems. *Streaming* graph partitioning treats graph data as an online stream, by reading the data serially and then determining the target partition of a vertex when it is accessed. However, the impact on the overall system performance of these partitioning policies has not been thoroughly evaluated on real graph-processing systems, and the understanding of the performance issues raised by such policies when used in real-world graph-processing systems is currently relatively limited. Gaining such knowledge can lead to the design of new policies, to new methods for tuning existing policies, and in general to better system design for distributed graph processing. Addressing this lack of understanding is the goal of our present work, in which we model, analyze and design new policies, and experimentally compare streaming graph-processing policies in real-world environments.

In this chapter we address the following five important challenges in partitioning large-scale graphs. The first challenge is partitioning graphs into splits with balanced numbers of vertices while minimizing edge-cuts, which is an NP-complete problem [1]. For graphs with billions of edges [15], the partitioning time can become too long, even when using partitioning heuristics. Second, many graphs of interest are not static but dynamic, with vertices and edges being added all the time. As a consequence, graph partitioning is then an online streaming process rather than an offline process. Third, the performance of partitioning depends on the graph-processing application. Fourth, because they are designed to address the needs of specific communities, each with their own applications and domains of expertise, graph-processing systems are designed around different programming models and generally take different evolutionary paths. The core programming model, which specifies how the system performs computation on vertices and how the distributed components of the system communicate, can affect the performance impact of partitioning. Fifth, the structure and capacity of the cluster used may impact the performance effect of a partitioning policy on the run time of graph-processing systems. For instance, switching the network from relatively low-speed Ethernet to high-speed InfiniBand, or the level of heterogeneity of a cluster [146] may change the relative merits of partitioning policies.

Many graph-partitioning approaches have been proposed to address these challenges, from offline partitioning heuristics to online, streaming, graph-partitioning policies. These partitioning-centric studies focus on the design of reasonable partitioning policies that are based on heuristics and rely on a limited set of theoretical metrics, such as the edge cut ratio [126], the number of vertices per partition [75, 112], etc. The partitions are created online by real-world graph-processing systems, which indicate that empirical metrics, such as partitioning time and algorithm run time are important for system developers and users. However, few partitioning policies have been proposed from the perspective of real systems. In contrast to such policies, the policies designed from a more theoretical perspective lack of simplicity and of considering the relationship between the computation and the communication, because they use relatively complicated heuristics and focus on minimizing the communication. And also, few experiments have been conducted on real graph-processing systems to evaluate the performance of existing partitioning policies. As our own and related studies [52, 56, 88] of entire graph-processing systems have shown, the results reported from narrow experiments can misreport performance by orders of magnitude, especially when the input workloads and the algorithms change from the conditions tested in the limited studies.

In this chapter, we address the challenges of streaming graph partitioning and the problem of relative lack of understanding about streaming graph-partitioning policies. Our main contributions are:

1. We model the run time of distributed graph-processing systems. We set the objective function of partitioning to minimizing the run time (Section 5.2). Our model extends related work [146] by including different programming models and implementation of graph-processing systems.

2. We conduct an experimental analysis of the performance implications of partitioning policies, using our run time model and conducting real-world measurements on a real-world graph-processing system—PGX.D [61]. We find out what graph characteristics are closely related to the run time. We further propose streaming graph policies based on the run-time-influencing graph characteristics (Section 5.3).

3. We evaluate and compare the performance of our policies, other streaming alternative, and also the start-of-the-art offline partitioner—METIS [72] on PGX.D, by using 3 large-scale graphs and 3 popular graph-processing algorithms. We use a set of metrics to present the partitioning performance, such as run time, partitioning time, edge cut ratio, scalability, etc. We also consider the impact of different real-world networks (Ethernet and InfiniBand) and the impact of a common technique (selective ghost node) used by graph-processing systems (Section 5.4).

4. We discuss how to use our results and the coverage of our method for different types of real-world graph-processing systems (Section 5.5).

5.2 A Model of Graph Processing Systems

In this section, we model the run time of different types of graph-processing systems and we discuss the objective function of graph partitioning of real graph-processing systems. We focus on graph-processing systems that follow the BSP programming model, that is, for which the graph-processing algorithm is executed in super-steps or iterations. Our model focuses on two-phase systems (described later in this section), but it can also represent single-phase systems such as the Pregel-based Apache Giraph. We consider in our model machine-level and thread-level programming abstractions, and blocking and parallel I/O. Conceptually, our model derives non-trivially from prior work; in contrast to the prior model of Xu et al. [146], which is the closest related work to our present study, our model considers a much larger variety of systems and has a higher granularity of processing units.

Similarly to the model of Xu et al. [146], suppose we have M working machines running N iterations of the same process. If T_i^k is the run time on machine i of the k-th iteration of some application, and if T^k denotes the (total) run time of the k-th iteration across all machines, then we have:

$$T^k = \max_i \{T_i^k\}, k = 1, 2, \ldots, N. \tag{5.1}$$

The total run time T_r of the application running on multiple machines can now be presented as:

$$T_r = \Sigma T^k, k = 1, 2, \ldots, N. \tag{5.2}$$

We assume conservatively that in each iteration all vertices are active (that is, considered for processing) and that messages are sent to all their neighbors, for three reasons. First, many popular algorithms match well this assumption, such as community detection [105] and PageRank [104]. Second, previous policies, and in particular the commonly used family of policies based on METIS, partition the whole graph with all its vertices and edges, so they implicitly follow this assumption. Last, predicting, for different algorithms, which of the vertices and edges become active during an arbitrary iteration is an open and challenging problem, but not a part of real-world graph-processing systems. Currently, no real graph-processing system is able to make prediction-based workload balancing in each iteration. Under this conservative assumption, the run time of every iteration on each machine can be considered to be equal, say to value $\overline{T}_i$, and so we can simplify Eq. (2) to:

$$T_r = N \times \max_i \{\overline{T}_i\}. \tag{5.3}$$

From the survey [95], there are three vertex-centric programming abstractions of graph-processing systems: one-phase abstraction, two-phase abstraction, and three-phase abstraction. For each iteration, the one-phase programming abstraction runs a single computation function, which consists of three computation tasks: processing incoming messages, applying vertex values, and preparing outgoing messages. The communication starts after the completion of the single computation function. The one-phase abstraction is often used in practice, for example in Pregel-like systems [43, 92]. The two-phase abstraction usually refers to two computation phases: the scatter phase (for preparing outgoing messages) and the gather phase (for processing incoming messages and applying vertex values). The communication happens between the scatter phase and the gather phase. The two-phase abstraction has been implemented in systems such as PGX.D [61]. Importantly, most one-phase systems can be converted to two-phase systems [95], but the reverse may not be true. We summarize in Table 2 the notation we propose for the time of

Table 5.1: Notations for the time of computation and communication of machine i.

Symbol	**Meaning**
$\overline{T}_i^g$	time spent processing incoming messages and applying vertex values across all threads.
$\overline{T}_{i,l}^g$	time spent processing incoming messages and applying vertex values in the l-th thread, $l = 1, 2, ..., L$.
$\overline{T}_i^s$	time spent preparing outgoing messages across all threads.
$\overline{T}_{i,q}^s$	time spent preparing outgoing messages in the q-th thread, $q = 1, 2, ..., Q$.
$\overline{T}_i^x$	time spent in communication, data transfers.
L	number of threads involved in processing incoming messages and applying vertex values
Q	number of threads involved in preparing outgoing messages

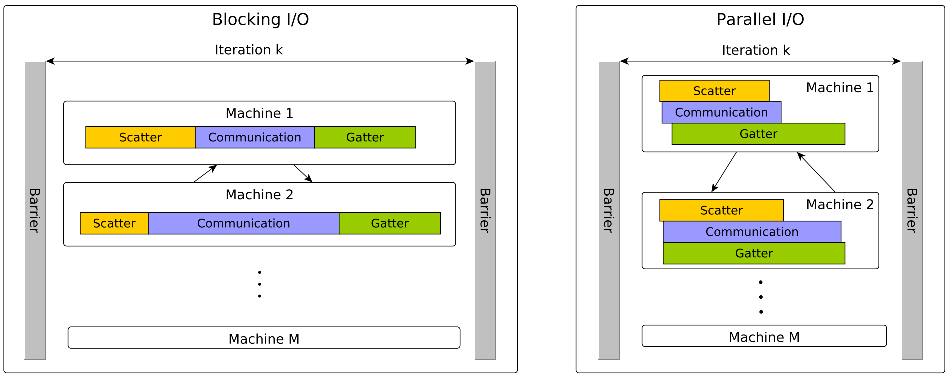


Figure 5.1: The computation phases and communication in one iteration of the Scatter-Gather abstraction.

the computation tasks and for the communication. The three-phase systems usually use the vertex-cut partitioning, which is out of the scope for this work. We further discuss three-phase systems, as a future extension of our modeling work, in Section 5.5.2.

Graph-processing systems can use one of the following two I/O modes, between computation and communication: *blocking I/O* and *parallel I/O*. With blocking I/O, computation and communication are executed serially. With parallel I/O, computation and communication can execute in parallel, with at least parts of the execution overlapped. For blocking I/O, $\overline{T}_i$ is the sum of the time spent on all computation phases and on communication. For parallel I/O, $\overline{T}_i$ is determined by the longest among the two computation phases and communication. We show in Figure 5.1 two computation phases and communication in one iteration of the Scatter-Gather abstraction.

Another important aspect of graph processing that we consider in our model is the granularity of the programming abstraction. In real graph-processing systems, where *multi-threading* has been used to accelerate computation, the run time of a computation phase is determined by the thread with the longest run time.

Table 5.2 summarizes the run time of a single iteration executed on machine i for different programming abstractions and I/O modes, in coarse-grained *machine-level* and fine-grained *thread-level*. Because the one-phase abstraction uses a single computation

Table 5.2: The time for one iteration ($\overline{T}_i$) for different programming abstractions and I/O modes

System	I/O block, machine-level	I/O block, thread-level	I/O parallel, machine-level	I/O parallel, thread-level
One-phase	$\overline{T}_i^g + \overline{T}_i^s + \overline{T}_i^x$	$\max(\overline{T}_{i,l}^g + \overline{T}_{i,l}^s) + \overline{T}_i^x$	$\max(\overline{T}_i^g + \overline{T}_i^s, \overline{T}_i^x)$	$\max(\max(\overline{T}_{i,l}^g + \overline{T}_{i,l}^s), \overline{T}_i^x)$
Two-phase	$\overline{T}_i^g + \overline{T}_i^x + \overline{T}_i^s$	$\max(\overline{T}_{i,l}^g) + \overline{T}_i^x + \max(\overline{T}_{i,q}^s)$	$\max(\overline{T}_i^g, \overline{T}_i^s, \overline{T}_i^x)$	$\max(\max(\overline{T}_{i,l}^g), \max(\overline{T}_{i,q}^s), \overline{T}_i^x)$

function, all computations for a vertex are always executed by the same thread, which means processing incoming messages and applying vertex updates cannot be parallelized with preparing outgoing messages. For the parallel I/O mode of the two-phase abstraction, the threads of a working machine need to be assigned to different computation phases to gain all the possible performance through parallelism. Thus, the assignment of the threads is an important factor for the run time of working machines. Moreover, for threads in the same phase, being able to balance their workload is crucial for achieving high performance.

The models we summarized in Table 5.2 are used to determine the graph characteristics that may have an impact on the run time of graph-processing systems (see Sections 5.3.1 and 5.3.2). However, the models cannot be used to (precisely) predict the run time of graph-processing systems, because the relationships between every time component (such as $\overline{T}_i^g$) of the models and the graph characteristics are not explored. It is non-trivial to formulate uniform relationships for various graphs, datasets, and systems.

The main target of partitioning graphs for real graph-processing systems is to achieve the shortest run time. Similarly to Xu et al. [146], we set the objective function for finding a graph partitioning that minimizes the total run time T_r:

$$\min\{T_r\} = N \times \min\{\max_i\{\overline{T}_i\}\} \tag{5.4}$$

In the following section, we investigate what are the interesting graph characteristics that affect the run time of the computation phases and communication, and we use this information to design new partitioning policies.

5.3 Design of Graph Partitioning Policies

The aim of this section is to design good graph-partitioning policies. In order to do so, we want to identify the graph characteristics that have significant impact on the run time of graph-processing systems. In Section 5.3.1, we propose a method for identifying such graph characteristics, and in Section 5.3.2 we empirically validate this method in the PGX.D graph-processing system. Then in Section 5.3.3 we design new streaming graph-partitioning policies according to the graph characteristics we identified.

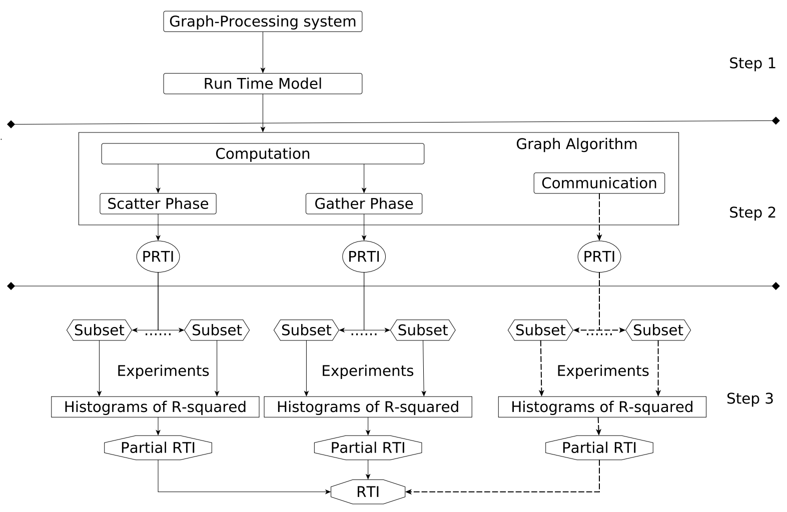


Figure 5.2: Our 3-step method for identifying the run-time-influencing graph characteristics of a two-phase system. (The selection from the communication component may not be performed if the communication is overlapped by the computation, depicted as dashed lines and further discussed in Section 5.3.2.)

Table 5.3: The characteristics of a partition of a graph

Characteristic	**Symbol**	**Definition**
Number of vertices	$\#V$	vertex count
Remote in-degree	D_{ri}	the number of in-edges from other partitions
Remote out-degree	D_{ro}	the number of out-edges to other partitions
Local in-degree	D_{li}	the number of in-edges in the partition
Local out-degree	D_{lo}	the number of out-edges in the partition (equal to local in-degree)
Total in-degree	D_{ti}	the sum of remote in-degree and local in-degree
Total out-degree	D_{to}	the sum of remote out-degree and local out-degree
Remote degree	D_r	the sum of remote in-degree and remote out-degree
Local degree	D_l	the sum of local in-degree and local out-degree
Total degree	D_t	the sum of remote degree and local degree

5.3.1 Identifying the Run-Time-Influencing Characteristics

As many popular graph-processing systems [43, 92] can only process directed graphs, we consider without loss of generality graph-processing systems that use a directed graph representation. In Table 5.3 we distinguish a number of characteristics of a partition of a directed graph that may have an impact on the run times of graph processing algorithms. Our target is to identify the graph characteristics that actually have the strongest such impact. We propose the following three-step method to achieve this, which is illustrated for a two-phase system in Figure 5.2.

Step 1: Determine the run time model of the graph-processing system from the possibilities listed in Table 5.2.

Step 2: Determine the Potential Run-Time-Influencing (PRTI) graph characteristics that may have an impact on the run time given the model determined in Step 1. These characteristics represent the candidate set for Step 3 of our method. The PRTI graph characteristics may vary for different graph-processing algorithms and perhaps even for the different components (e.g., computation and communication) of the same graph-processing algorithm, and for the model of the graph-processing system. For each component (even each phase if the model includes multiple phases) of the algorithm, we select a set of PRTI graph characteristics according to the graph entities operated by the graph algorithm. For example, the number of vertices ($\#V$) is always selected for the scatter phase (one phase of the computation component) because vertices are processed during the computation, and the remote out-degree (D_{ro}) is selected for the communication component if the algorithm sends messages by remote out-going edges.

Step 3: Identify from the PRTI graph characteristics the actual Run-Time-Influencing (RTI) graph characteristics that are strongly related to the run time of (a phase or component of) an algorithm. In order to do so, we first create different candidate subsets from each set of PRTI graph characteristics for the partial set of RTI graph characteristics. We will show how to create these subsets in Section 5.3.2. We take an experimental approach to pick the appropriate subset. For each experiment, we measure the run time of each working machine and we calculate the values of the graph characteristics of the partition stored on it shown in Table 5.3. For each candidate subset, we conduct a linear regression [98] between the run times of the working machines and the values of the graph characteristics in that subset of the partitions assigned to them. In this way, we obtain a value of the R-squared (R^2) coefficient from every experiment.

We perform multiple experiments using different setups (in terms of system configurations, datasets, and graph-partitioning policies) and we build a histogram with the numbers of occurrences of the R^2 value in given ranges (for an example, see Table 5.6). We select as the partial set of RTI graph characteristics the subset of PRTI with the most occurrences in the highest range of R^2 values. After having obtained the partial sets of RTI characteristics of multiple phases/components of an algorithm, they can be combined to form the set of RTI characteristics of the whole algorithm. The RTI graph characteristics are strongly determined by the behavior of graph algorithms and the model of graph-processing systems. Using different datasets may affect the coefficients of the linear regression between the run times and the values of the subset of PRTI graph characteristics, but not affect the distribution of the R^2 values. So, the obtained RTI graph characteristics are also applicable for other graphs that are not used in the experiments.

5.3.2 Empirical Results Validating the Method

We will now empirically validate the method from the previous section for the PGX.D graph-processing system.

Step 1: We use Table 5.2 to identify the run time model corresponding to PGX.D. As PGX.D is a multi-threaded graph-processing system with two-phase abstraction and parallel I/O, its run time model is:

$$\overline{T}_i = \max(\max(\overline{T}^g_{i,l}), \max(\overline{T}^s_{i,q}), \overline{T}^x_i). \tag{5.5}$$

Step 2: We seek to understand the operation of PGX.D in order to select the PRTI characteristics. In PGX.D, the threads assigned to the scatter phase and the gather phase are called worker threads and copier threads, respectively. PGX.D balances the workload across its worker threads with the edge-chunking technique and across its copier threads with the max-slot first strategy. So, $\max(\overline{T}^g_{i,l})$ and $\max(\overline{T}^s_{i,q})$ are equal to the average run time of worker threads and copier threads, respectively. PGX.D uses the continuation mechanism to buffer and combine messages between working machines to reduce communication. A dedicated poller thread is maintained in each working machine for sending and receiving messages. PGX.D implements a commonly used technique, called Selective Ghost Node (SGN), to further reduce the network traffic. SGN duplicates the high-degree vertices (ghosts) in each partition. A vertex is selected as a ghost if the sum of its in-degree and out-degree is larger than a pre-defined threshold. The use of SGN is optional for users.

We apply Step 2 of our method on different components of the PageRank algorithm. The scatter phase in PageRank reads all vertices and prepares messages to remote neighbors through the out-edges of each vertex. Therefore, the PRTI graph characteristics of the scatter phase are the number of vertices, the remote out-degree, the local out-degree, and the total out-degree. The gather phase in PageRank processes all incoming messages and updates each vertex, and so, the PRTI graph characteristics of the gather phase are the number of vertices and the remote in-degree. The only PRTI graph characteristic of the communication component in the PageRank algorithm is the remote out-degree.

Step 3: In order to identify the RTI graph characteristics for PGX.D, we perform experiments with PageRank (maximum 10 iterations) with PGX.D deployed on a 16-machine cluster in Oracle Labs with properties as in Table 5.4. We explain the experimental setup in terms of the system configurations, the datasets, and the partitioning policies that we employ.

We use four system configurations of worker and copier threads: w24c2, w18c8, w12c14, and w6c20 [61], where the notation w24c2 means that we set 24 worker threads and 2 copier threads in each working machine, etc. We conduct experiments with or without using the SGN technique.

Table 5.4: The environment of our experiments

Category	**Item**	**Detail**
CPU	Type	Intel Xeon E5-2660
	Frequency	2.20 GHz
	Parallelism	2 socket * 8 core * 2 HT
Network	Card	Mellanox Connect-IB
	Switch	Mellanox SX6512
	Raw BW	56 Gbit/s (per port)
Software	OS	Linux 2.6.32 (OEL 6.5)
	Compiler	gcc 4.9.0

We use three large-scale graphs, Twitter, Scale_26, and Datagen_p10m (see Table 6.4). The Twitter dataset is one of the largest publicly available real-world datasets and consists of a graph of its users with the follower relationships between them. Scale_26 is a synthetic graph generated by the Graph500 generator, with a scale factor of 26. Graph500 is the de-facto standard for comparing hardware infrastructures for graph processing systems. Datagen_p10m is created by the Linked Data Benchmark Council (LDBC) generator, which aims to produce graphs with structures and properties similar to those of real-world social networks, such as Facebook. The LDBC generator is used by the Graphalytics project [10], which is an active big data benchmark for graph-processing systems. The two generated graphs contain roughly 1 billion edges, and are comparable in size to the Twitter dataset. We set the threshold for ghost selection of SGN to 50,000 for Twitter and Scale_26, and to 600 for Datagen_p10m.

We use three streaming graph-partitioning policies incorporated in PGX.D, viz. the in-degree balanced policy (I), the out-degree balanced policy (O), and the total degree balanced policy (IO). All policies assign vertices to partitions by balancing the in-degree, the out-degree, or the total degree across partitions. To this end, each policy first determines the average (in-/out-/total) degree per partition, and then assigns vertices sequentially to the partitions, going from one partition to the next when the (in-/out-/total) degree of the former exceeds the corresponding average.

We obtain 72 executions of PageRank by combining all system configurations with or without SGN (4x2), all datasets (3), and all partitioning policies (3). We find that the run time of PageRank is dominated by either the scatter phase or the gather phase. As PGX.D optimizes the network traffic and is deployed on the high-speed InfiniBand network, the communication time in PageRank is overlapped by both the scatter and gather phases. In Figure 5.3, we show the run time of a single iteration, the run time of the longest worker thread, and the run time of the longest copier thread of PageRank for Twitter when using the w12c14 configuration. We notice that for all policies the run time of a single iteration

Table 5.5: Summary of datasets

Dataset	**V**	**E**	**d**	**D̄**	**Q1**	**Q2**	**Q3**	**Max D**	**Type & Source**
Twitter	41,652,230	1,468,365,182	8	35	5	13	34	3,081,112	Real-world, Public [79]
Scale_26	32,804,978	1,073,741,824	10	33	1	4	17	1,710,236	Synthetic, Graph500 [48]
Datagen_p10m	9,749,927	687,174,631	72	70	30	87	204	648	Synthetic, LDBC [81]

V and **E** are the vertex count and edge count of the graphs. **d** is the link density ($\times 10^{-7}$). **D̄** is the average vertex out-degree. **Q1**, **Q2**, and **Q3** are the first quartile, median, and the third quartile of vertex total-degree, respectively. **Max D** is the largest vertex total-degree.

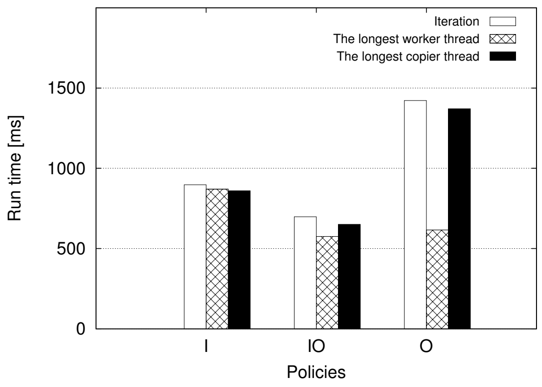


Figure 5.3: The run time of an iteration, of the longest worker thread, and of the longest copier thread of PageRank for Twitter using system configuration w12c14.

is approximately 50 ms higher than the maximum run time of the longest worker and copier threads, which is due to the overhead of the system. We have similar findings for other system configurations. Thus, we only need to consider the scatter and gather phases to select the RTI graph characteristics.

For each execution, we measure the run times of the scatter and gather phases, respectively, of the sixteen working machines and we calculate for each machine the values of the graph characteristics of its graph partition. We consider different candidate subsets of RTI characteristics from the PRTI set, which we evaluate empirically in order to determine the RTI graph characteristics. We could consider all subsets of the PRTI set as candidates, but in practice, we can use previous knowledge to consider fewer subsets. For example, we derive three subsets of characteristics from the PRTI set of the scatter phase: the number of vertices (#V) and the total out-degree (D_{to}), only D_{to}, and the local out-degree (D_{lo}). We consider the subset consisting of only D_{to} because many previous partitioning policies focus on it. The subset of #V and D_{to} is considered because during our experiments we found that some partitions with similar D_{to} but different #V have (significantly) different run times. The subset of D_{lo} is randomly created as a control subset in order to show how weak the relationship between the run time of the scatter phase and a randomly selected graph characteristic can be.

For every candidate subset for each phase we create a histogram of the R^2 values for

Table 5.6: The numbers of experiments with PageRank that have the value of R-squared (R^2) for the scatter phase in the indicated ranges.

Range	$\overline{T}_i^g$ **with** #V **and** D_{to}	$\overline{T}_i^g$ **with** D_{to}	$\overline{T}_i^g$ **with** D_{lo}
[0.9, 1]	39	28	10
[0.8, 0.9)	8	11	7
[0.7, 0.8)	9	9	1
[0.6, 0.7)	5	4	6
[0, 0.6)	11	20	48

all 72 experimental setups. In Table 5.6 we show the histograms for the scatter phase. From this table, we identify as the RTI graph characteristics of the scatter phase the number of vertices (#V) and the total out-degree (D_{to}) because they have the highest number of values of R^2 in the range of [0.9, 1]. Unlike previous policies, which focus on the communication component by minimizing edge-cuts, our results show that the number of vertices is also an important factor. Similarly, for the gather phase, we identify the number of vertices (#V) and the remote in-degree (D_{ri}) as the RTI graph characteristics. Combining the results of both phases we identify as the complete set of RTI graph characteristics for PageRank the number of vertices (#V), the total out-degree (D_{to}), and remote in-degree (D_{ri}).

We have conducted a similar set of experiments for the weakly connected component (WCC) algorithm, which computes the maximal groups of vertices connected by edges. The RTI graph characteristics of WCC are the number of vertices (#V), the total degree (D_t), and the remote degree (D_r).

5.3.3 Four New Graph Partitioning Policies

In this section, we design four new graph-partitioning policies based on the findings from the experiments in Section 5.3.2. The first of these, called the *degree-balanced* (DB) policy, is new, while the other three of these are randomized versions of the I, IO, and O policies from Section 5.3.2.

Our target is to design a good partitioning for graph-processing systems in general, not for a specific algorithm. Combining the RTI graph characteristics identified by running PageRank and WCC, we show that the number of vertices is a common characteristic. For different algorithms, they may propagate messages through in- or out-edges. It is difficult to determine which graph characteristics about degree we should balance. We decide to select total in-degree and total out-degree, because of two main reasons. First, the remote or local degree of a partition can only be calculated after the finish of partitioning. We cannot use them during the execution of partitioning. Second, from the perspective of the system, balancing the total in-degree and total out-degree is a generic way to cover

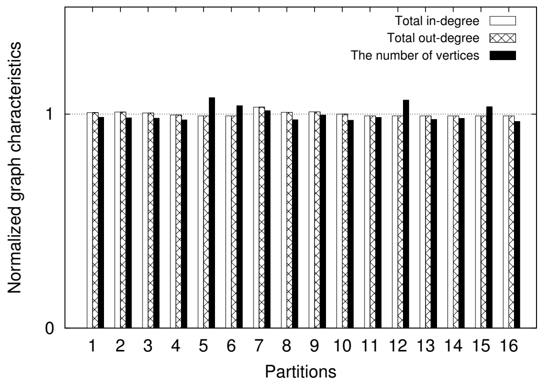


Figure 5.4: The normalized values of the graph characteristics achieved by the DB policy for Twitter.

different algorithms. Thus, the primary purpose of our DB policy is to balance the total in- and out-degree per partition, and its secondary purpose is to balance the sum of the in-degree and out-degrees across the partitions by setting a constraint on the number of vertices of the partitions.

With DB, every next vertex is assigned to the degree-smallest of what we call the opposite partitions. For a vertex with in-degree V_i and out-degree V_o, a partition with total current in-degree D_{ti} and total current out-degree D_{to} is called *opposite* if $V_i > V_o$ and $D_{ti} \leq D_{to}$, or the other way around. The *degree-smallest* partition is the partition with the smallest sum of its current total in-degree and total out-degree. We set a *constraint* on the number of vertices per partition to ensure that they do not become too imbalanced. In the DB policy, this constraint is flexible and can be set by the user. The process of assigning a vertex to a partition by the DB policy is shown in Policy 1.

In order to show the balance of the partitions created by the DB policy, we apply it to the three datasets (Twitter, Scale_26, and Datagen_p10m) to create 16 partitions each. We set the constraint on the size of the partitions to 1.5 times their average size (we assume the size of the graph to be known ahead of time). In order to show the balance, we normalize the number of vertices, the total in-degree and the total out-degree of each partition relative to their average values across all partitions. Figure 5.4 shows that the graph characteristics are very well balanced for the Twitter partitions. For the Scale_26 and Datagen_p10m graphs, we achieve similar results. We have also partitioned the graphs into different numbers of partitions (2, 4, 8, and 32), and also then we achieve balanced partitions. Our results even indicate that we can achieve balanced numbers of vertices without setting a constraint.

From the experimental results in Section 5.3.2, we find that the run time of the machines varies even though they have equal numbers of edges to process in the I, IO, and O policies. The reason is that the numbers of vertices of the partitions, which are run-time-influencing, are not balanced. To address this issue, we change the streaming order of the

Policy 1 The DB policy

Input: V_i, V_o, the constraint on the number of vertices C, a sorted queue of partitions $P[M]$ with ascending $D_{ti} + D_{to}$, the number of partitions M

Output: the index of the assigned partition $Index$, a sorted queue of partitions after the assignment

```
 1: $Flag \leftarrow 0$    ▷ *$Flag$ indicates if there is an opposite partition of the vertex in the queue.*
 2: **if** $V_i > V_o$ **then**
 3:     **for** $j = 1 \rightarrow M$ **do**
 4:         **if** $D^j_{ti} \leq D^j_{to}$ **then**    ▷ *$D^j_{ti}$ and $D^j_{to}$ is the current total in-degree and the current total out-degree of the $j$-th partition $P^j$.*
 5:             Assign the vertex to $P^j$, update $D^j_{ti}$ and $D^j_{to}$.
 6:             $Flag \leftarrow 1$, $Index \leftarrow j$
 7:             **break**
 8:         **end if**
 9:     **end for**
10:     **if** $Flag = 0$ **then**    ▷ *Cannot find an opposite partition for the vertex.*
11:         Assign the vertex to $P^1$, update $D^1_{ti}$ and $D^1_{to}$.    ▷ *Assign the vertex to the smallest/first partition.*
12:         $Index \leftarrow 1$
13:     **end if**
14: **else if** $V_i < V_o$ **then**
15:     **for** $j = 1 \rightarrow M$ **do**
16:         **if** $D^j_{ti} \geq D^j_{to}$ of $P^j$ **then**
17:             Assign the vertex to $P^j$, update $D^j_{ti}$ and $D^j_{to}$.
18:             $Flag \leftarrow 1$, $Index \leftarrow j$
19:             **break**
20:         **end if**
21:     **end for**
22:     **if** $Flag = 0$ **then**
23:         Assign the vertex to $P^1$, update $D^1_{ti}$ and $D^1_{to}$.
24:         $Index \leftarrow 1$
25:     **end if**
26: **else**    ▷ $V_i = V_o$
27:     Assign the vertex to $P^1$, update $D^1_{ti}$ and $D^1_{to}$.    ▷ *Assign the vertex to the smallest/first partition.*
28:     $Index \leftarrow 1$
29: **end if**
30: **if** #$V$ of $P^{Index} \geq C$ **then**
31:     Remove $P^{Index}$ from the queue.
32:     $M \leftarrow M - 1$
33: **end if**
34: Ascending sort the partition queue $P[M]$ by $D_{ti} + D_{to}$ of each partition
```

vertices in these policies, from the sequential ordering to a *random ordering*, which accesses vertices randomly. There are also other stream orderings, such as the *BFS ordering* and the *DFS ordering*. We select the random ordering for three main reasons. First, from the evaluation of Stanton and Kliot [126], the random ordering has comparable performance to the BFS and DFS orderings in many cases. Second, the BFS and DFS orderings need to pre-traverse the graphs, which is time consuming, in particular for large graphs. The traverse time may be even longer than the partitioning time. Third, the BFS and DFS orderings can be more complicated when a graph has multiple connected components. By using the random ordering of each original policy in PGX.D, we create three new policies called RI, RIO, and RO, in which "R" stands for the random ordering. Figure 5.7 shows a comparison of the O and RO policies. The RO policy achieves more balanced numbers of vertices across partitions, while keeping the balance of the total degrees.

Many graphs are not static, but mutate over time. Although we only cover static graphs in our experiments, our partitioning policies can be used to partition mutating graphs online as well, obviating the need to re-partition a graph after it has changed. For example,the DB policy does not need to know meta information of the graph (such as the number of vertices and edges) or the neighborhoods of vertices to assign vertices. When partitioning a mutating graph, it can simply assign new vertices one-by-one based on its rules, and update the meta information of every partition (such as the total in-degree and total out-degree). However, many graph-processing systems cannot support online graph-partitioning policies and process mutating graphs. We are not able to show the ability of partitioning mutating graphs of our policies in our experiments.

5.4 Experimental Results

In this section we conduct comprehensive experiments with different graph partitioning policies, applications, and system configurations. In Section 5.4.1 we present our experimental setup, and at the end of the section we explain the experiments reported in later sections.

5.4.1 Experimental Setup

Experimental environment: We keep using the same cluster as shown in Table 5.4. Besides using InfiniBand, in Section 5.4.5 we also evaluate the performance on 1 Gbit/s Ethernet. We run all experiments on 16 working machines, except for the scalability test in Section 5.4.4, in which we use four different numbers of machines (2, 4, 8, and 32).

Datasets: We will only present the results of executing graph-processing algorithms on large-scale graphs. In fact, we have also run experiments on a smaller graph, Livejournal [121] (with 4,847,571 vertices and 68,993,773 edges). However, the performance

Table 5.7: Experimental setup for each experiment in Section 5.4.

Section	Algorithms	Datasets	Metrics	Threads	Network	SGN technique
5.4.2	PageRank	Twitter	Run time	All	InfiniBand	No
5.4.3	PageRank	Twitter, Scale_26, Datagen_p10m	ECR, SD	w12c14	InfiniBand	No
5.4.4	All	Twitter, Scale_26, Datagen_p10m	Run time, scalability	w12c14	InfiniBand	No
5.4.5	All	Twitter, Scale_26, Datagen_p10m	Performance ratio	w12c14	InfiniBand, Ethernet	Yes
5.4.6	-	Twitter, from Scale_22 to Scale_26	Partitioning time	-	-	-

differences of the graph-partitioning policies are quite small in that case. In Section 5.4.6, we include four more Graph500 graphs than we have used in Section 5.3.2, with the scale factor running from 22 to 25. For these graphs, the numbers of vertices and edges are doubled with every step of the scale factor.

Algorithms: We have conducted a comprehensive survey of graph-processing algorithms [51]. Our survey covers over 100 research articles published in 10 representative conferences (including VLDB, SIGKDD, SIGMOD, etc.) in recent years. Graph algorithms in previous publications can be categorized into different classes by functionality. We find that the top 3 occurred classes of algorithms are graph traversal, general statistics, and connected components. The percentages of the occurrence of these 3 classes of algorithms are 46.3%, 16.1%, and 13.4%, respectively. In total, they have about 70% occurrence among all types of algorithms. We select one exemplar algorithm from each of these 3 classes, Breadth-First Search (BFS) from graph traversal, PageRank from general statistics, and Weakly Connected Components from connected components. PageRank and BFS propagate updates through out-edges. WCC propagates updates through both in- and out-edges, and does not need any parameter. For PageRank, the termination condition is set to maximum 10 iterations. For BFS, we select the same source vertex for each graph for all partitioning policies.

Partitioning policies: In total, we evaluate 12 graph-partitioning policies: 2 streaming policies (R and H) commonly used by graph-processing systems, 2 streaming policies (LDG and CB) from the literature, the 3 original streaming policies (I, IO, and O) used in PGX.D, our 4 new streaming policies (RI, RIO, RO and DB) presented in Section 5.3.3, and the state-of-the-art partitioner (M). Except for RI, RIO, and RO, all policies use the sequential ordering of the graphs. We summarize the partitioning policies in Table 5.8. According to the experimental results of the CB policy [146], we set its degree threshold percentage to 30%.

The experiments we have conducted are as follows:

- In Section 5.4.2, we evaluate the impact of the configurations of worker threads and copier threads.
- In Section 5.4.3, we measure the workload imbalance of partitions by using the edge cut ratio and the standard deviation of normalized run-time-influencing graph characteristics.

Table 5.8: Twelve partitioning policies in our experiments.

Policy	**Streaming**	**Mechanism**
R	Yes	Randomly assign a vertex to a partition.
H [92]	Yes	Hash partitioning.
LDG [126]	Yes	Assign a vertex to the partition, which has most neighbors of the vertex.
CB [146]	Yes	Assign a vertex to a partition with the smallest workload or with the least incremental workload.
I [61]	Yes	Balance the in-degree of partitions, original policy in PGX.D.
IO [61]	Yes	Balance the total-degree of partitions original policy in PGX.D.
O [61]	Yes	Balance the in-degree of partitions, original policy in PGX.D.
RI	Yes	The I policy using random ordering, proposed in this work.
RIO	Yes	The IO policy using random ordering, proposed in this work.
RO	Yes	The O policy using random ordering, proposed in this work.
DB	Yes	The greedy degree-balanced policy, proposed in this work.
M [72]	No	METIS, multi-level graph partitioning

- In Section 5.4.4, we show the run time of graph-processing algorithms with different datasets. We also present the scalability of each partitioning policy.
- In Section 5.4.5 we report the performance of using Ethernet and the impact of using the selective ghost node technique.
- In Section 5.4.6 we investigate the time spent on graph partitioning, considering different numbers of partitions and graph sizes.

A summary of the experiments, and of the remaining sections, is in Table 5.7.

5.4.2 The Impact of Worker and Copier Threads

There are many possible configurations with different numbers of worker threads and copier threads. The configuration of worker threads and copiers threads can significantly influence the performance of PGX.D [61]. In this section, we explore the impact of the thread configuration on 12 partitioning policies.

Key findings:

- The configuration of worker and copier threads has a significant impact on the run time of PGX.D for all partitioning policies.
- In most experimental runs, the thread configuration w12c14 shows the best performance.

We use four configurations, w24c2, w18c8, w12c14, and w6c20, which give a reasonable coverage of the possible configurations. Figure 5.5 shows the run time of PageRank for the Twitter dataset. In general, the best performance is obtained from either w12c14 or w18c8 for different partitioning policies. We also conduct other groups of experiments, with different algorithms, datasets and machines. In most cases, the configuration of w12c14 achieves the best performance, and so we empirically use this as our default thread configuration for the following experiments.

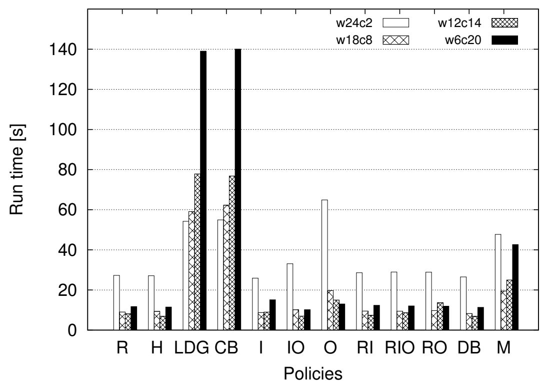


Figure 5.5: The run time of PageRank for Twitter with four thread configurations.

5.4.3 Workload Distribution

In this section we discuss the workload distribution among working machines. The workload includes two parts, the communication workload between working machines and the computation workload on each machine.

Key findings:

- The edge cut ratio is not a good indicator for the quality of partitioning for real graph-processing systems, at least when communication is not the performance bottleneck of the system.
- The standard deviation of the normalized run-time-influencing graph characteristics can be used to measure the imbalance of the computation workload.
- The design of partitioning policies should not only focus on minimizing the communication, but also on balancing the communication between pairs of machines.

The edge cut ratio (ECR) is defined as the ratio of the number of edges that connect vertices that are placed in two partitions over the total number of edges in the graph. ECR is used by many previous studies to measure the total communication workload. We show the ECR of the 12 partitioning policies on Twitter, Scale_26, and Datagen_p10m in Figure 5.6. Because CB, LDG, and M consider the neighborhoods of the vertex to be assigned and of the already assigned vertices in each partition, they are the top 3 policies that achieve the lowest ECR for all three datasets (except that LDG ranks sixth for Datagen_p10m). In contrast, the ECR of other policies is very high, because they assign vertices without considering their neighborhoods.

We use the standard deviation (SD) of the normalized (see Section 5.3.3 for the normalization) run-time-influencing (RTI) graph characteristics (i.e., the number of vertices, total out-degree, and total in-degree) to understand the computation workload across working machines. Figure 5.7 shows the results for Twitter, which is partitioned into 16 splits. As shown in Figure 5.4, the Twitter partitions under the DB policy have balanced RTI graph characteristics, so the SD of all normalized RTI graph characteristics is

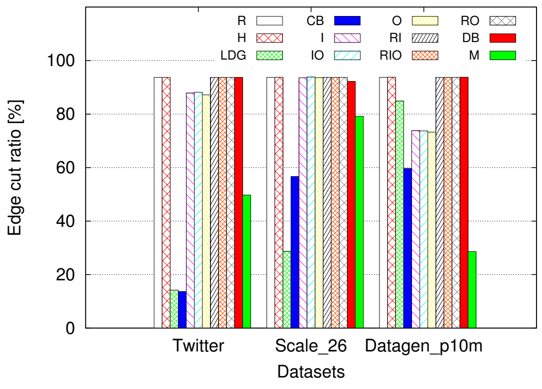


Figure 5.6: The Edge Cut Ratio of all partitioning policies for 3 datasets.

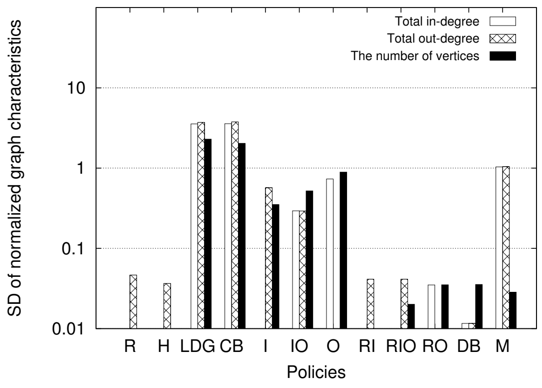


Figure 5.7: The standard deviation of the normalized RTI graph characteristics for Twitter for all partitioning policies (the values of missing bars are too small to display).

small. We also find that the SDs for the CB and LDG policies are significantly higher than for the other policies. The reason is that vertices are accumulated to very large partitions to reduce edge cuts in CB and LDG. For the M policy, although the SD of the normalized number of vertices is small, the SDs of the normalized total in-degree and out-degree are relatively large, which indicates that communication is not balanced between pairs of working machines. Surprisingly, the random-based policies (R and H) also obtain small SD (we have repeated the R partitioning 5 times with different random seeds and obtained consistent results).

In Figure 5.8 we show the run time of PageRank on all 3 datasets for all graph partitioning policies. The LDG, CB, and M policies result in the longest run times, even though they achieve a low ECR. The reason is that communication is not the dominant workload in PGX.D when using the high-speed InfiniBand, as we have discussed in Section 5.3.2. This means that ECR is not a good metric when the communication is not the dominant part of the workload. We find that in general, the partitioning policy with smaller SD of the RTI graph characteristics leads to shorter run time, and so SD can be used as a metric to evaluate the quality of partitioning for computation-dominated processing. Except for

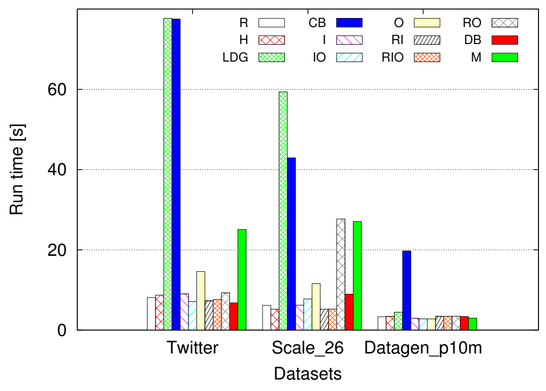


Figure 5.8: The run time of PageRank for 3 datasets with all partitioning policies.

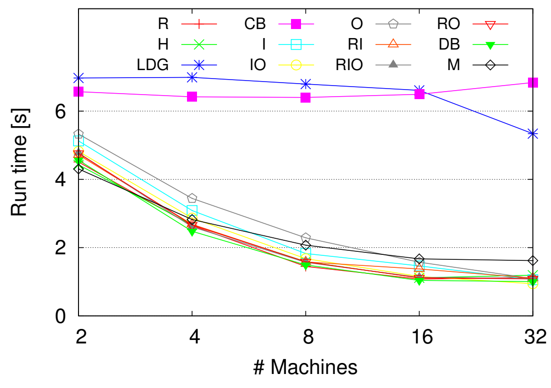


Figure 5.9: The scalability of the BFS algorithm for Twitter.

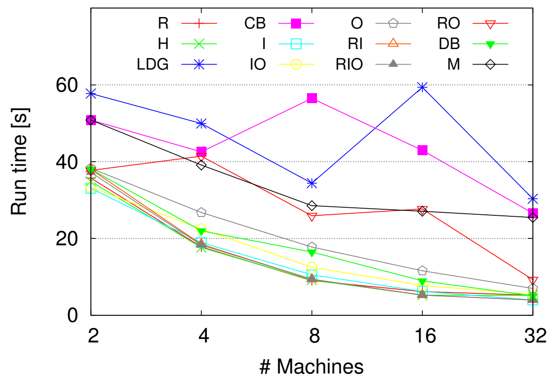


Figure 5.10: The scalability of the PageRank algorithm for Scale_26.

the CB, LDG, M, and O policies, the SD of the other policies is less than 0.5 and their run times are very close to each other. In practice, it is useful to find a threshold for SD beyond which the run time of graph processing may significantly increase. This threshold may be determined by analyzing the statistics obtained from many more experiments with various algorithms and datasets.

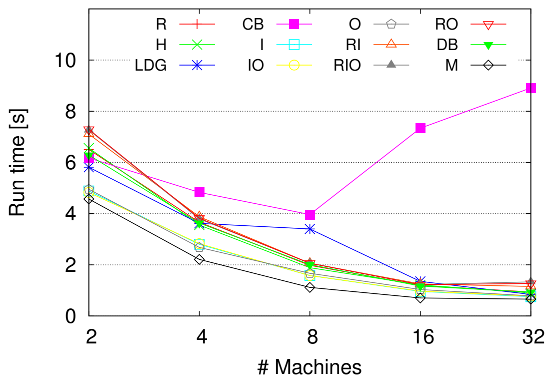


Figure 5.11: The scalability of the WCC algorithm for Datagen_p10m.

5.4.4 The Impact of the Partitioning Policies on Performance

In this section we present the performance impact of the partitioning policies on the performance of graph algorithms for different algorithms, datasets, and number of working machines.

Key findings:

- The Degree-Balanced policy achieves good performance, while previous streaming policies from the literature (LDG and CB) perform the worst.
- The graph structure has an impact on the performance of graph partitioning.
- Most partitioning policies show reasonable scalability with the increase of the number of working machines (partitions).

The run time of PageRank for 3 datasets with all partitioning policies is depicted in Figure 5.8. There is no overall winner among the partitioning policies, but LDG and CB have the worst performance as the computation workload of for these policies is highly skewed between working machines (see Figure 5.7). DB achieves good performance for all graphs. For the Twitter graph, the run time of PageRank is the shortest. Random ordering cannot always help to achieve good performances evidenced by the O and RO policies for partitioning Scale_26. The impact of graph partitioning is more significant in highly skewed graphs, such as Twitter and Scale_26. For Datagen_p10m, we see that only CB has obvious performance impact. Both LDG and M yield results comparative to those other partitioning policies. Simple partitioning policies, such as the commonly used H policy, perform well for most algorithms and graphs. The reason is that computation is the dominant workload in our experiments and the H policy balances normalized RTI graph characteristics as shown in Figure 5.7.

In Figures 5.9, 5.10, and 5.11 we show that most partitioning policies exhibit good scalability when increasing the number of worker machines up to 16—the benefit of increasing the number of machines from 16 to 32 is not significant. An important reason is that the workload is not heavy enough when processing the graphs with more than 16 machines (i.e., the hardware resource is redundant). For LDG and CB, the scalability is not obvious. To reduce edge-cuts, no matter how many number of partitions, LDG and CB may place vertices to a small subset of partitions, which dominates the run time of the algorithms. We also find that the random ordering results in poor scalability, such as the RO policy shown in Figure 5.10.

5.4.5 The Impact of Network and the Selective Ghost Node

In this section, we compare the performance impact of using 56 Gbit/s InfiniBand versus 1 Gbit/s Ethernet, and of using selective ghost node (SGN), which is a commonly used technique in graph-processing systems for reducing network traffic.

Key findings:

- The run time of graph-processing algorithms on high-speed InfiniBand is orders of magnitude smaller than on low-speed Ethernet.
- Using the selective ghost node technique may not always have a positive impact on the performance.

We report the performance of InfiniBand relative to Network when running 3 algorithms with Twitter in Figure 5.12. In all experiments, using InfiniBand leads to much better performance, from 10 times to nearly 900 times faster than the Ethernet. It is very interesting that the performance ratio can be as much as hundreds times, while the bandwidth of the InfiniBand is only about 50 times larger than that of the Ethernet. It may because that the communication is not balanced between pairs of machines. For example, one machine may have heavy communication with multiple other machines. Other machines may have to wait that machine to finish their communication, which makes the data transfer and message processing extremely slow.

We show the performance improvement for PageRank of 3 datasets by using SGN on InfiniBand and on Ethernet in Figures 5.13 and 5.14, respectively. Not all values are positive, indicating that using SGN cannot always help to achieve good performance, because the time synchronizing ghost nodes can be longer than the run time reduced by using SGN. Overall, the performance change on Ethernet is larger than that on InfiniBand, because Ethernet is more sensitive to the change of network traffic.

5.4.6 The Time Spent on Partitioning Graphs

The complexity of the partitioning policies and the time spent on partitioning graphs are also important for us to determine the choice of policies. Because the M policy is implemented in an offline single-machine partitioner, and the LDG and CB policies need

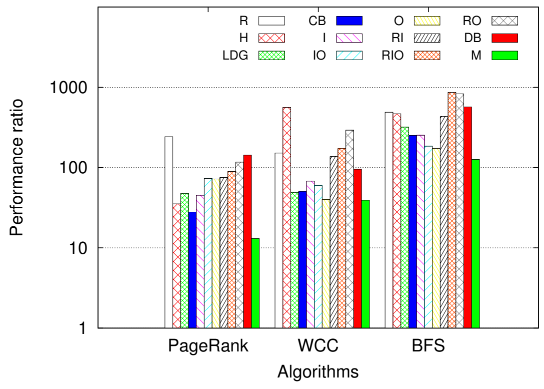


Figure 5.12: The performance ratio of 3 algorithms for Twitter on InfiniBand relative to Ethernet (vertical axis has logarithmic scale).

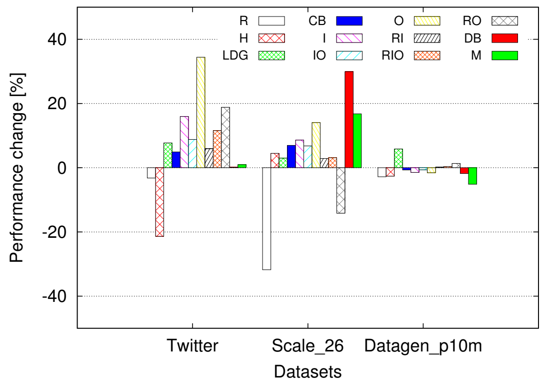


Figure 5.13: The performance change of PageRank for 3 datasets when using SGN on InfiniBand.

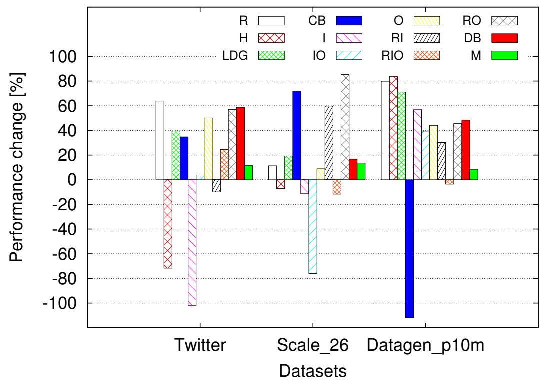


Figure 5.14: The performance change of PageRank for 3 datasets when using SGN on Ethernet.

to acquire the global information to assign vertices, it is non-trivial to implement these policies in a distributed manner. In this section, we compare the time spent on partitioning graphs on a single machine.

Key findings:

- The LDG, CB, and M policies need much more time for partitioning graphs than the other streaming policies.
- The number of partitions has a significant impact on the partitioning time of LDG and CB.
- The partitioning time of all policies increases linearly with the size of the graph.

We first explore the time spent on partitioning the same graph into different numbers of partitions. In Figure 5.15, we show the time of each policy for partitioning Twitter into 2, 4, 8, 16, and 32 partitions, respectively. For the M policy, we use another machine (equipped with two Intel Xeon CPU E5-2699 2.30 GHz processors and 384 GB memory), because the M policy runs out of memory when using the working machine in Table 5.4. LDG, CB, and M are the policies with the longest partitioning time. The M policy applies a multi-level scheme, in which the coarsening phase is complex and time consuming. This long partitioning time of M matches a previous experiment [133], where more than 8.5 hours is needed to partition the Twitter graph using a less powerful machine. For the assignment of a vertex, the LDG and CB policies need to traverse all partitions to calculate the number of its neighbors in each partition. To assign some low-degree vertices in CB, counting the edges between each pair of partitions is also required. The traversal of partitions is very expensive. With the increase of the number of partitions, the LDG and CB policies need to spend significantly more time on partitioning, because of the complexity of the traversal process. Except for LDG and CB, we observe time increase of DB, which is incurred by sorting the partition queue, the size of which is equal to the number of working machines. In practice, the size of clusters is limited, many of which have less than thousands of machines. Thus, the impact of increasing the number of partitions is limited for the DB policy.

We also investigate the partitioning time on different sizes of graphs. Figure 5.16 shows the time spent on partitioning Graph500 graphs with 5 different scales (from Scale_22 to Scale_26). We partition each graph into 16 splits. Similarly to Twitter, we use the same machine with 384 GB memory only for executing the M policy with Scale_26, because out of memory. LDG, CB, and M are the slowest policies. All partitioning policies exhibit good scalability with increasing the size of graphs.

5.5 Discussion

In this section, we discuss how to use our results and how to extend the use of our model and method to more graph-processing systems.

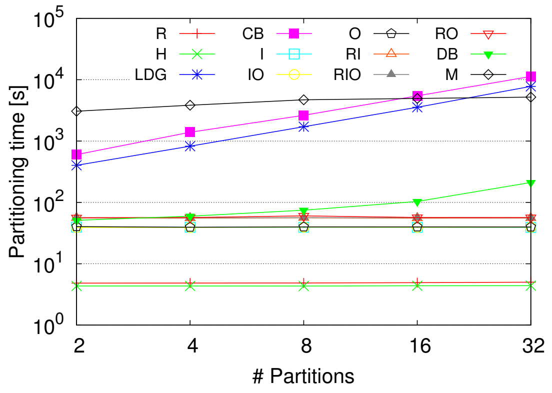


Figure 5.15: The time spent on partitioning the Twitter graph into different numbers of partitions for all policies (vertical axis has logarithmic scale).

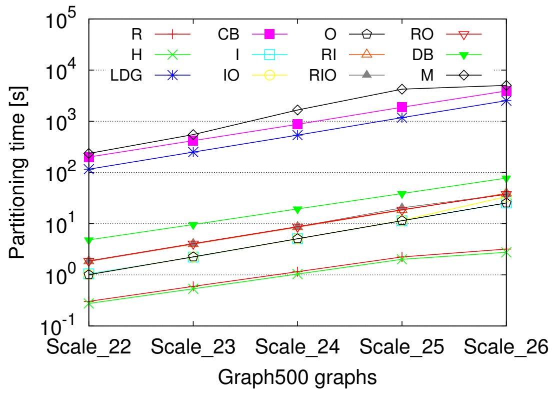


Figure 5.16: The time spent on partitioning Graph500 graphs into 16 partitions for all policies (vertical axis has logarithmic scale).

Table 5.9: Key findings of our experiments.

Section	**Key findings**
5.4.2	The configuration of worker and copier threads has a significant impact on the run time of PGX.D. In most experimental runs, the thread configuration w12c14 shows the best performance.
5.4.3	The ECR is not a good indicator for the quality of partitioning for real graph-processing systems. The SD of the RTI graph characteristics can be used to measure the imbalance of the computation workload. The design of partitioning policies should also focus on balancing the communication between machines.
5.4.4	The DB policy achieves good performance, while LDG and DB perform the poorest. The graph structure has an impact on the performance of graph partitioning. Most partitioning policies show reasonable scalability with the increase of the number of partitions.
5.4.5	The run time of algorithms on InfiniBand is orders of magnitude smaller than on Ethernet. Using the selective ghost node technique may not always have a positive impact on the performance.
5.4.6	The LDG, CB, and M policies need much more time for partitioning graphs than the other streaming policies. The number of partitions has a significant impact on the partitioning time of LDG and CB. The partitioning time of all policies increases linearly with the size of the graph.

5.5.1 How to Use Our Results

We summarize the key findings of our experiments in Table 5.9. Key findings in Section 5.4.4 and Section 5.4.6 are about the performance of partitioning policies. It is difficult to obtain clear rules as to which partitioning policy should be used for which graph-processing system, which algorithm, and which graph. We identify four main reasons for this difficulty. First, graph-processing systems are designed and implemented with specific goals and optimization techniques. It is not easy to quantify the impact of these implementations and techniques on the performance of graph-partitioning policies. Second, graph algorithms have various behaviors. The impact of graph algorithms on the performance of partitioning policies is significant. Third, graphs have diverse structures and characteristics. It is very difficult to identify the typical graph structures and the most important graph characteristics that can represent a given graph [88]. In practice, the identified structures and characteristics should be easily calculated, which is crucial for large-scale graphs. Fourth, heterogeneous hardware infrastructure (different CPU, amount of memory, network connection, etc.) also has significant impact. For the same combination of graph-processing system, algorithm, and graph, if the deployed cluster is changed, the best partitioning policies may also change.

Although it is non-trivial to obtain best practice, we discover and summarize some generic suggestions for designing and using policies. Key findings in Section 5.4.2 and Section 5.4.5 are closely related to the PGX.D system and its hardware infrastructure. They may not be applicable for other systems, but these findings indicate that system configuration and tuning should be carefully conducted (for different partitioning policies). Key findings in Section 5.4.3 are more generic and can be used by other researchers to design and measure the performance of their graph-partitioning policies. Our DB policy cannot always outperform other partitioning policies in all cases, but in general, it achieves good performance (short run time of graph algorithm and fast partitioning process), if any other graph system that falls in the same run time model of PGX.D, we would suggest to use the DB policy.

5.5.2 The Coverage of Our Model and Method

In Section 5.2, we propose a run time model of two-phase graph processing systems, which also encompasses *one-phase* systems. In our experiments, we use PGX.D (a real-world production system based on the two-phase abstraction) as the real graph-processing system. Because we have tested our work on production-quality code, and because of the simplicity of the conversion between the one-phase abstraction and the two-phase abstraction [95], our work also indicates that our method could be applied with trivial adaptations to systems using the one-phase abstraction.

We now discuss the extensions needed to apply our work to systems based on the *three-phase* abstraction. A typical three-phase abstraction is the Gather-Apply-Scatter (GAS) model, which is first implemented in PowerGraph [45]. Vertex-cut partitioning is often implemented in GAS systems: a vertex can have multiple copies, each of which is distributed to a working machine. One copy is selected as the master, and others are mirrors. In GAS, the gather phase collects the local incoming information for vertices, then calculates their partial vertex values. The apply phase collects all partial values and computes final vertex values. Last, the scatter phase distributes the update to corresponding edges. There are two periods of communication in the GAS model, with one period between the gather and apply phases for sending partial vertex values to the master, and another between the apply and scatter phases for distributing final vertex values to all mirrors. We extend our run time model to GAS systems, for example, by observing that the run time becomes the sum of the time spend on each of the three computation phases and the two communication periods in the blocking I/O mode. Next, we can use our method to pick out run-time-influencing graph characteristics for vertex-cut partitioning, and proceed design new policies. (Using these steps, we have already completed a preliminary model for three-phase systems, but we do not report the outcome in this work, as we have not proceeded with the design of new policies and have not conducted meaningful experiments with them.)

5.6 Related Work

5.6.1 Graph Processing Systems

Single machines with limited resources are unable to handle growing modern graphs. *Generic* distributed data-processing systems, such as Hadoop [141], have first been adapted to analyze and process large-scale graphs on clusters. However, because of the limitation of programming models, generic data-processing systems cannot support iterative graph-processing applications very well. It has been reported that the performance of generic data-processing systems, for graph-processing applications, is much worse than *specific* graph-processing systems [43, 87, 92]. This has become a common knowledge in the graph-processing community.

Many graph-processing systems adapt the vertex-centric paradigm, in which graph-processing algorithms are implemented from the perspective of each vertex of graphs. The Bulk Synchronous Parallel (BSP) computing model has been used by many graph-processing systems, such as Pregel [92] and Hama [114], mainly because the BSP model simplifies the design and implementation of iterative graph-processing algorithms. A BSP computation of a graph-processing algorithm consists of a series of global iterations (or supersteps). In each iteration, active vertices execute the same user-defined function, generate messages, and transfer them to neighbors that are not located in the same machine.

Synchronization is needed between two consecutive iterations to ensure that all vertices have been processed and all messages have been delivered. The cost of synchronization in BSP systems may incur performance degradation, especially when the workload between working machines is not balanced. To improve performance, graph-processing systems, such as GraphLab [87] and GraphHP [17], have used asynchronous models to avoid using barriers for synchronization and to reduce the performance degradation caused by imbalanced workload. The use of asynchronous models increases the complexity of graph-processing systems and, in some cases, creates redundant messages [151] when executing graph algorithms.

Graph-processing systems can be categorized into three main *multi-phase* systems, based on their vertex computation abstractions [95]: *one-phase* [43, 92], *two-phase* [61,109,128], and *three-phase* [39,45]. The main computation in graph processing includes processing incoming messages, applying vertex updates, and preparing outgoing messages. In each multi-phase abstraction, the main computation is placed and executed in different computation phases. For example, in *Scatter-Gather*, which is a two-phase abstraction, the scatter phase prepares outgoing messages, and the gather phase collects incoming messages and applies updates to vertex values. We will further analyze and discuss these three abstractions in Section 5.2 and Section 5.5.

5.6.2 Graph Partitioning Policies

The study of partitioning policies for graph-processing is based on two main disciplines, graph partitioning and performance analysis. We survey in this section the related materials published in each of these two disciplines, in turn. Overall, ours is one of the few studies combining theoretical work in graph partitioning with experimental comparison of policies using several algorithms and datasets, which, as we indicate in the introduction, is important for the validity of the results. Our main findings from this survey, regarding graph partitioning, are summarized in Table 5.10.

Graph Partitioning. Graph partitioning has been explored and studied for a long time in many research areas [73, 95], from scientific workflow scheduling [44] to recent work on large-scale graph processing [46]. Balanced graph partitioning, which aims to balance the number of vertices in each partition while minimizing the communication between partitions, is known as the k-way graph partitioning problem and has been proved to be NP-hard [1]. To achieve an approximate solution, many *traditional heuristics* [72, 110] have been proposed. Many of them adapt the *multi-level partitioning* scheme, which typically includes *three main phases* [110], coarsening to reduce the size of the graph, partitioning the reduced graph, and uncoarsening to map back partitions for the original graph. The prominent example of multi-level partitioning, METIS [72] and its family of partitioning policies [32], are used by the community because of their high-quality

Table 5.10: Graph-processing systems using different partitioning approaches.

Partitioning approach	**Example heuristics**	**Example systems using the approach**
Traditional heuristics	METIS [72], ParMETIS [74]	-
Streaming	Hash, LDG [126]	Giraph [43], HeAPS [146]
Vertex-cut	Random, Balanced p-way [45]	PowerGraph [45], GraphX [46]
Dynamic	Exchange [112], Migration [75]	GPS [112], Mizan [75]
Chunking	File size	Hadoop [141], Stratosphere [138]

partitions and relatively fast partitioning speed. However, we identify three main reasons for which these heuristics may be unable to handle the partitioning problem for distributed graph-processing systems. First, most distributed graph processing systems are designed for large-scale graphs, with millions of vertices and billions of edges. For partitioning policies designed explicitly for single-node operation, such as METIS, large-scale graphs and their intermediate partitioning data often do not fit in the main memory of the system, which causes spills to disk and severe performance degradation, and in our experience even system crashes. For multi-node heuristics such as ParMETIS [74], using them in practice may be complex and time consuming, because they need a global view of graphs and slow synchronization for partitioning. Second, these heuristics are designed to operate offline. They need to access the entire graph for every partitioning operation, which makes them relatively inefficient for growing and changing graphs. Third, many of the heuristics are designed for scientific computing workloads. In particular, they have been designed to solve k-way partitioning problem, by recursively executing 2-way partitioning when k is a power of 2. They may not be able to effectively partition real-world graphs representative for other domains, and in particular real-world graphs with arbitrary values of k [2].

To address the problems faced by offline heuristics, online streaming graph partitioning policies have been proposed for distributed graph-processing systems. *Hash partitioning*, a type of streaming graph partitioning, is used in many graph processing systems, such as Pregel-like systems [43, 92], because of its simplicity and short partitioning time. The drawbacks of hash partitioning for real large-scale graphs are obvious. For computation, partitions created by hash partitioning policies from highly-skewed real graphs [45] can have an even number of vertices but will often include partitions where vertices have very diverse in-/out-degrees, case in which graph-processing algorithms such as Breadth-First Search (BFS) traversal will incur high computation imbalance. For communication, hash partitioning does not consider any locality of vertices and edges. There may be an inordinate amount of edge-cuts between partitions, which results in intensive network traffic. To conclude, hash partitioning policies have so far not considered highly-skewed graphs, and result when used on real-world graphs in partitions that lead to imbalanced computation and communication.

Many studies make efforts in two main directions to obtain balanced graph partitions. The first direction is to design more complex steaming graph-partitioning policies. Stanton and Kliot [126] propose more than ten streaming policies. Many factors are selected and used in these policies, such as the relationship between the vertex to be assigned and the current vertices in the partition, buffering for assigning a group of vertices, and streaming orders. From their evaluation, a *linear-weighted deterministic greedy policy* (LDG) performs the best. In LDG, a vertex is assigned to the partition with the most neighbors, while using the remaining capacity of partitions as a penalty. Tsourakakis et al. [133] formulate a partitioning *objective function*, considering the costs of edge cut and the size of partitions. Based on this function, they design a streaming graph partitioning, FENNEL, which is a greedy policy using different heuristics to place vertices. Closest to our work, to address heterogeneity of computing hardware and network, Xu et al. [146] build a model for the heterogeneous environments and discuss a time-minimized objective function from the perspective of graph-processing systems. They propose six streaming graph partitioning policies and evaluate their performance in both homogeneous and heterogeneous environment. From their experimental results, the *combined policy* (CB) achieves the best performance in homogeneous environment and reasonably good performance in different settings of heterogeneous environment. They use the analytical method to estimate the workload of the whole computation. In our model, we further divide the whole computation and use real experiments to find out run-time-influencing graph characteristics. Our method can be more precise. Advanced streaming graph partitioning policies can achieve comparable performance of METIS [126, 146].

The second direction is to partition graphs by *vertex-cut* [45,46]. Vertex-cut partitioning places edges, instead of vertices, to different partitions. According to percolation theory [127], good vertex-cuts can be achieved in power-law graphs. Evenly placing edges can reduce the workload imbalance and the large communication of high-degree vertices, which are represented as multiple replicas and stored in different partitions. Vertex-cut partitioning has its drawbacks. System-wise, the graph-processing system needs to allow a single vertex's computation to span multiple machines, which increases the complexity of the system. Performance-wise, too many pieces of vertex replicas can still generate high communication, primarily to synchronize vertex status. We summarize our survey of this class of graph-partitioning policies in Table 1, in the row "Vertex-cut". Vertex-cut partitioning is used by few graph-processing systems. In our work, we focus on edge-cut partitioning, which is used by more systems.

To avoid the workload imbalance incurred by static streaming partitioning and vertex-cut partitioning and also by the execution of algorithms (for example, active vertices vary in each iteration during the process of the BFS algorithm), *dynamic repartitioning* is moving vertices between working machines during the execution of algorithms. The general process of dynamic repartitioning methods can be abstracted as the following sequence of four steps : discover workload imbalance of computing machines, find the pairs of computing machines for migrating vertices, determine which vertices are required to move, and migrate selected vertices from its source to destination. Mizan [75] selects the execution time of each machine as the metric for workload imbalance and maintains a distributed hash table to record the position of vertices. GPS [112] simply uses the outgoing messages as the workload-imbalance metric. When computing machines are paired, they will exchange vertices rather than migrate vertices from one to another. Both Mizan and GPS take a *delay migration strategy* to alleviate the overhead of migration of vertices and their associated data. Shang et al. [115] focus on how much of the workload should be moved between pairs of working machines and on which vertices should be moved. They also propose several constraints to improve the benefit of migration. We show systems that support dynamic repartitioning in Table 5.10, in the row "Dynamic".

Partitioning performance. Although many graph partitioning methods and policies have been proposed, their performance has not been thoroughly evaluated with various input graphs and algorithms. Theoretical metrics, such as the edge cut ratio and modularity [4, 126, 133] are generally used to measure the quality of partitions. For real graph-processing systems, these metrics do not directly represent the performance of partitioning [146]. In practice, metrics such as the run time of graph-processing algorithms, partitioning time, and the variance of the run time on different machines/threads represent the performance of bottleneck components in real graph-processing systems. Meyerhenke et al. [97] design their graph partitioning heuristic based on label propagation and size constraints for social networks and web graphs. Guerrieri and Montresor [50] discuss the properties of high quality partitions and introduce a distributed edge-partitioning framework. Both studies lack experimental results from executing algorithms on real graph processing systems, to show the performance of their partitioning methods in practice. Stanton and Kliot [126], FENNEL [133], and Xu et al. [146] compare the performance of many streaming partitioning policies on data-processing systems. The systems they run experiments on are not (advanced) graph-processing systems—Spark's generic data processing for Stanton and Kliot, Hadoop for FENNEL, and a prototype of Pregel for Xu et al., contrast starkly with highly optimized production systems such as GraphLab [87] and Giraph [43]. Their evaluations are also limited to the use of a single algorithm, PageRank; our own and related studies [52, 56, 88] have shown that the results obtained from a single algorithm do not characterize well the performance expected from the general field of graph processing. In contrast, in this work, we conduct comprehensive experiments on an advanced distributed graph-processing system—PGX.D, using 3 representative algorithms, 3 large-scale graphs with billions of edges from different domains, different practical configurations and in particular different types of network, and many different performance metrics.

5.7 Summary

Graph partitioning is an important aspect of achieving high performance when designing and using distributed graph-processing systems. Many graph partitioning policies have been proposed so far, aiming to minimize communication, balance the number of vertices on each working machine, and reduce the time spent on partitioning, etc. However, most of the partitioning policies are not designed from the perspective of real-world distributed graph-processing systems. In addition, the performance of existing partitioning policies has not been evaluated in-depth on real systems. In this chapter, we address this situation by proposing models, partitioning policies, and an experimental evaluation of different partitioning policies in graph processing.

We model the run time of different types of graph-processing systems. We set minimizing the run time as the objective function of partitioning policies. The models we proposed cover the one-phase and two-phase systems, using the blocking I/O and parallel I/O modes, in machine-level and thread-level.

We propose a method to identify run-time-influencing graph characteristics by analyzing the run-time model and by understanding the relationship between different graph characteristics and the run time. Based on the run-time-influencing graph characteristics, we design new graph partitioning policies to obtain balanced partitions.

We use many metrics to evaluate the performance of twelve partitioning policies. We select in our experiments three popular graph-processing algorithms and three large-scale graphs from both real world and synthetic graph generators. We also evaluate the impact of real-world networks and a commonly used technique in graph-processing systems. Our results indicate that the newly-designed DB partitioning policy shows good performance, while existing streaming policies, such as LDG and CB, do not perform well.

We also discuss our preliminary work and ideas regarding how to use our results and the coverage of our model and method.

Chapter 6

Designing Distributed Heterogeneous Graph Processing Systems

In this chapter, we design and implement three families of distributed heterogeneous systems that can use both the CPUs and GPUs of multiple machines. We further focus on graph partitioning, for which we compare existing graph-partitioning policies and a new policy specifically targeted at heterogeneity. We implement all our distributed heterogeneous systems based on the programming model of the single-machine TOTEM, to which we add a new communication layer for CPUs and GPUs across multiple machines to support distributed graphs, and a workload partitioning method that uses offline profiling to distribute the work on the CPUs and the GPUs. We conduct a comprehensive real-world performance evaluation for all three families. To ensure representative results, we select three typical algorithms and five datasets with different characteristics. Our results include algorithm run time, performance breakdown, scalability, graph partitioning time, and comparison with other graph-processing systems. They demonstrate the feasibility of distributed heterogeneous graph processing and show evidence of the high performance that can be achieved by combining CPUs and GPUs in a distributed environment.

6.1 Overview

Increasingly large graphs are being generated every day, not only by big companies such as Facebook [30] and LinkedIn [86], but also by Small and Medium Enterprises (SMEs) [120] such as Wikimedia for online encyclopedia [142], Friendster for social networks [36] and XFire for online gaming [145]. To process these graphs, many graph-processing systems, using a variety of hardware platforms (e.g., multiple CPUs, GPUs, or combinations thereof) have been designed and implemented. With CPUs and GPUs becoming increasingly more powerful and affordable, SMEs who could previously invest only in CPU-based commodity clusters can now afford to buy a heterogeneous environment. However, current graph-processing systems cannot operate on both distributed and heterogeneous settings. This raises the important research question of *How to design a distributed and heterogeneous graph processing system?* In this chapter, we explore systematically this question, through the design and experimental evaluation of three families of distributed heterogeneous graph-processing systems.

Typical *distributed CPU-based* graph-processing systems such as Pregel [92], GraphX [46], and PGX.D [61] can handle large graphs by using multiple machines, but choose to ignore the additional computational power of accelerators because of the increased complexity of the programming environment. GPU-enabled systems, on the other hand, can accelerate graph processing considerably [59], but choose to ignore the distributed environment because of the added complexity of (multi-layered) partitioning. For example, Medusa [152] and Gunrock [137] can utilize multiple GPUs on a *single* machine and TOTEM [41] is a *single-machine* heterogeneous graph-processing system that can use one CPU and multiple GPUs. MapGraph [38, 39] can use GPUs from multiple machines.

In this chapter, we combine the scalability of distributed CPU-based graph-processing systems with the computational power and energy efficiency of GPU-enabled graph-processing systems. Specifically, we design and implement distributed heterogeneous graph-processing systems that can use *both* the CPUs and the GPUs of multiple machines. Our study bridges the gap between existing distributed CPU-based systems and GPU-enabled systems for large-scale graph processing.

We explore the design space of these distributed heterogeneous systems with a focus on partitioning. Graph partitioning is mandatory for systems with multiple processing units [53,146]. Well balanced partitions can improve the performance of graph-processing systems, but the way to build and balance them depends heavily on the system characteristics. In the presence of heterogeneity in the platform, balance is difficult to determine and achieve [90, 117].

In this chapter, we propose three different graph-partitioning architectures: Distributed-Parallel (DP), Parallel-Distributed (PD), and Combined (C). For the DP and PD architectures, we select and combine existing graph-partitioning policies that have promising characteristics for the respective phases; for C systems, we design a new policy to construct balanced partitions for multiple CPUs and GPUs.

To understand the performance differences between these systems and policies in the context of real-life graph processing applications, we implement our distributed CPU+GPU systems on top of the popular system TOTEM, from which we adopt the programming model, the data structures, and several optimization techniques. To enable the inter-machine communication necessary for a distributed system, we enhance the communication layer of TOTEM (a single-machine system in which data is transferred only between the CPU and the local GPUs) to use MPI [102] and GPUDirect [47]. We further address other technical issues, such as separately building partitions on working machines and aggregating results. Finally, because TOTEM doesn't provide a method to compute the workload partitioning between the CPU and the GPU, we propose a new method that leads to balanced partitions among processing units. Our method is based on the approach proposed by Shen et al. [117], which uses an offline profiling method to compute the relative capability of the CPU and the GPU. We extend this method to determine a balanced partitioning of the input datasets on the CPU and the GPU.

We comprehensively evaluate the performance of three families of distributed heterogeneous graph-processing systems with different graph-partitioning policies. In our experiments, we show how our systems can process large-scale graphs faster than single-machine GPU-enabled systems and distributed CPU-based systems. Moreover, we show that the systems we design can analyze large-scale graphs that cannot be handled by single-machines systems.

Our contribution is four-fold:

1. We explore the design space of distributed heterogeneous graph-processing systems (Section 6.3). Specifically, we explore three families of such systems using different graph-partitioning architectures.

2. We design and select graph-partitioning policies for the three families of systems (Section 6.3).

3. For each of the three families of systems, we implement the first working system (Section 6.3).

4. We conduct comprehensive experiments to evaluate the performance of our distributed heterogeneous graph-processing systems (Section 6.4).

6.2 Extended BSP-Based Programming Model

Through comprehensive experiments, we have already evaluated the performance of existing GPU-enabled graph-processing systems [56]. Our past results indicate that TOTEM is, among the systems we have tested, the most reliable one. Furthermore, TOTEM has clear and comprehensive data structures for representing graphs and partitions, and several optimization techniques for improving performance. In this section, we introduce the programming model and the most important features of TOTEM (as shown in Figure 6.1).

TOTEM is a vertex-centric system following the Bulk Synchronous Parallel (BSP) programming model [135]. In the BSP model, iterative graph algorithms are executed

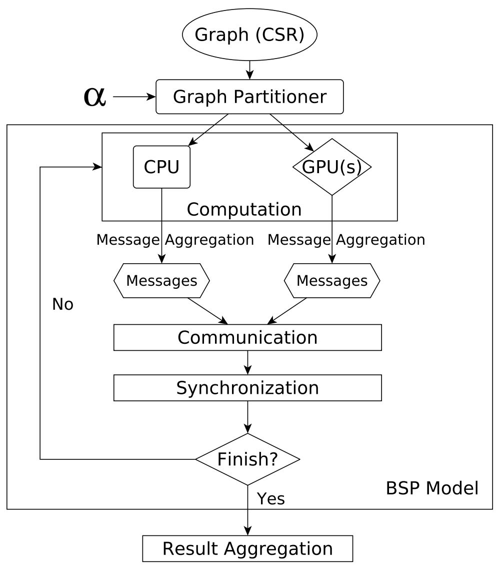


Figure 6.1: The BSP-based programming model and features of a single-machine heterogeneous graph-processing system.

in multiple, consecutive supersteps. Each superstep coordinates processing data in parallel, across physical processing units (P). Graph vertices are partitioned across processing units. To avoid processing all vertices during each superstep, vertices can be activated and only the active vertices are processed. Each superstep includes three phases: a *computation phase* during which all active vertices in each partition execute the same operations of the graph algorithm; a *communication phase* during which vertices send messages via *cut edges*, that is, edges whose vertices belong to partitions managed by different processing units, to other partitions; and a *synchronization phase*, which guarantees that all messages are transferred.

To use memory efficiently, TOTEM uses the Compressed Sparse Rows (CSR) [6] data format to represent graphs and their partitions. The CSR format includes two arrays, the vertex array V and the edge array E. Each element in V stores, for a vertex, the head of its list of neighbors, as an index in E, and E stores for each vertex index a list of its neighbors. Input graphs are split into multiple partitions, which are further assigned to the CPU and the GPU(s).

A message aggregation technique is implemented in TOTEM to reduce the number of messages sent via cut edges between partitions. The messages sent from vertices in a partition to the same remote vertex are temporarily buffered. All messages for the same remote vertex are combined into one message before sending. To allow for this aggregation, each partition maintains two sets of buffers: the outbox buffers and the inbox buffers. Each outbox buffer stores messages to a remote partition, and each inbox buffer collects messages from a remote partition. Thus, for each partition, the system has $|P| - 1$ outbox and $|P| - 1$ inbox buffers ($|P|$ is the number of partitions). This message aggregation

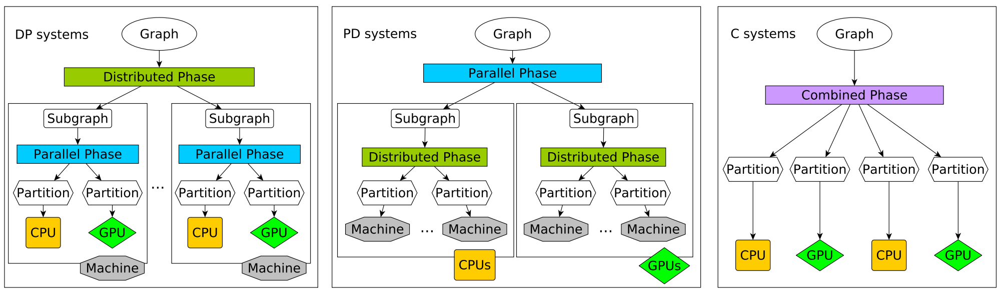


Figure 6.2: Three architectural families of distributed heterogeneous graph-processing systems: DP (left), PD (middle), and C (right) systems.

technique can significantly reduce communication [41].

For heterogeneous CPU+GPU systems, a crucial operation is partitioning the workload between CPUs and GPUs. For many graph processing algorithms, the computation workload can be heavily related to the number of edges of the input graph [41, 61, 150]. Thus, partitioning the workload is equivalent to determining the fraction of the edges to be put on the CPU(s), which both in TOTEM and in our system is denote by α, with the remainder to be put on the GPU(s). Existing studies have not proposed a method to select this value. In Section 6.3.5, we describe our method to calculate the value of α. This method is based on existing work on heterogeneous CPU+GPU systems [90, 117], which we extended and adapted to graph-processing workloads.

6.3 The Design of Distributed Heterogeneous Systems

In this section we discuss the design of our three families of distributed heterogeneous systems for graph processing. We further present the partitioning policies we have selected and/or designed for these systems, and we discuss the most challenging aspects of the implementation.

6.3.1 Three Families of Distributed Heterogeneous Systems

To extend single-machine graph-processing systems to a distributed architecture, graph partitioning is a key aspect for both functionality and performance. In this section, we focus on the architecture of graph partitioning in distributed heterogeneous systems.

Balanced partitions often lead to good system performance. To achieve balanced partitions in distributed heterogeneous systems, the graph partitioner (see Figure 6.1) must consider three main aspects: inter-machine workload distribution, intra-machine workload distribution (i.e., between the CPU and the GPU), and communication minimization. To explore these different aspects, we design three families of distributed heterogeneous graph-processing systems, with the architectures depicted in Figure 6.2. We describe these architectures further.

Distributed-Parallel (DP) systems: the partitioner takes two phases to partition the input graph on the processing units. First, in the *distributed phase*, the partitioner assigns vertices to different computing machines, similar to the graph partitioning approach used by many distributed graph-processing systems such as Pregel [92], Giraph [43], or GPS [112]. Next, in the *parallel phase*, each machine further splits the subgraph it received across its local CPUs and GPUs, similar to the actions taken in TOTEM.

Parallel-Distributed (PD) systems: in contrast to DP systems, PD systems reverse the sequence of the distributed phase and the parallel phase: the graph is first divided into two subgraphs, one to be processed by the CPUs and the other to be processed by the GPUs, and then each subgraph is further distributed across CPUs and GPUs.

Combined (C) systems: unlike DP and PD systems, the combined systems use a single-phase partitioning, the *combined phase*. The partitioner directly assigns vertices to processing units, considering both the CPUs and the GPUs of the entire system. To still achieve balanced partitions, the heterogeneity of processing units is the main challenge that must be tackled. We describe our approach to this challenge in Section 6.3.4. We note that our work is the first to consider a combined partitioning approach for heterogeneous systems.

For all three families of distributed heterogeneous systems, the graph partitioner operates on a single master machine. The partitioned data (vertices, edges, etc.) are then sent to the working machines. Besides being out of the scope of this chapter, a distributed partitioner is also non-trivial to implement, so we leave its design and implementation for future work.

6.3.2 Classification of Partitioning Policies

Graph-partitioning has been studied for many years and many policies, with different goals, have been proposed [53]. For example, the main target of the state-of-the-art graph partitioner *METIS* [72] is to reduce the number of edge cuts between partitions, which leads to less communication. The *total-degree balanced* policy (IO) used in the PGX.D graph-processing system [61] aims to balance the total degrees of all partitions, allowing each computing unit to have the same workload. Based on their goals and focuses, we identify four classes of graph-partitioning policies, and summarize them in Table 6.1. We discuss each of the four classes below.

The **computation-focused** policies focus on achieving balanced computation workloads across the processing units, with *no* consideration for the edge cuts between partitions. Policies in this class are based on the intuition that the computation workload of graph-processing algorithms can occur incrementally, and mainly along the edges of graphs [41, 61, 150]. Many computation-focused policies, such as the IO policy used by

Table 6.1: Four classes of graph-partitioning policies.

Class	**Examples**
Computation-focused	IO [61], HIGH [41], LOW [41]
Communication-focused	METIS [72], LDG [126]
Computation-communication	MW [146], MI [146]
Unfocused	hash [92], random [126], chunking [141]

the PGX.D system, are designed to balance the in-degree and/or out-degree of partitions. Considering the utilization of the cores of processing units, in particular for GPUs, the vertex-degree centric policies have been used in heterogeneous CPU+GPU systems. For example, the HIGH and LOW policies used by TOTEM fall into this class. For both of these policies, the vertices of a graph are first sorted by their out-degrees, and then they are split up into two parts. For the HIGH (LOW) policy, the part with higher (lower) out-degrees is assigned to the CPU and the remainder to the GPU.

The **communication-focused** policies are proposed mainly to minimize the communication between partitions. Many *traditional heuristics* adopt theoretical methods to achieve minimum communication, such as METIS and its family of partitioning policies [32]. Emerging *streaming partitioning policies*, which treat vertices as a stream and assign them one-by-one instead of in bulk, also include many policies to reduce the communication. For example, the LDG policy [126] places a vertex to a partition that already has most of the new vertex's neighbors. Some of the communication-focused policies also make an effort, albeit minimal, to balance computation workload, for example the LDG policy through a penalty function to avoid that too many vertices accumulate to one partition and thus lead to highly imbalanced computation.

The **computation-communication** policies consider *both* the computation and the communication workloads. For example, the *Min-Workload* (MW) policy [146] combines the two workloads, by greedily assigning vertices to partitions that incur minimum combined workload. The *Min-Increased* (MI) policy [146] places vertices with the least increase of workload.

The **unfocused** class includes policies designed for simplicity—either to reduce the implementation effort, or because partitioning is not believed to deliver a balanced workload. For example, the *hash* policy is commonly used by Pregel-like systems [43, 92]. The *random (R)* policy is another lightweight unfocused policy that randomly places vertices to different partitions. Non-random policies, such as chunking [141], take subsets of equal length from the input graph file and place them round-robin in different partitions.

6.3.3 Selection of Partitioning Policies

As discussed in Section 6.3.1, our three families of distributed heterogeneous graph-processing systems use one or two phases for partitioning graphs: a distributed and a parallel phase, or a combined phase. For each phase, any graph-partitioning policy could in principle be selected and used. However, because of policy complexity and the architecture of the graph-processing systems, some policies may perform poorly in specific partitioning phases. In this section we discuss the selection of suitable policies for each family of systems; we will evaluate these choices in Section 6.4.

The distributed phase For the distributed phase, we need to address the load-balancing in the distributed system. Because of message aggregation, our systems already reduce inter-machine communication. Moreover, the partitioning policies in both communication-focused and computation-communication-focused classes are very expensive—for example, METIS (communication-focused) takes more than 8.5 hours to partition a large graph (1.4 billion edges) with a typical machine that could be used by an SME (3.6 GHz CPU and 16 GB memory) [133]. Therefore, we select the partitioning policies for the distributed phase from the computation-focused class, as they will provide fast partitioning and good balancing of the computation workload, leading to reasonable load balancing.

To understand how to balance the workload, we observe that the computation can be divided into two parts: one for processing incoming messages, and another one for applying updates and creating outgoing messages. The partitioning policy should aim to balance the substantial part of the workload, requiring a decision to be made between balancing the in-degree or the out-degree of the partitions. Because of local message aggregation, the number of incoming messages for a partition is at most $(|P| - 1) \times |V_p|$, with V_p the set of vertices of partition p). Because $|V_p| << |E_p|$, with E_p the set of edges of partition p, this in-message processing part of the workload is significantly smaller than the updates and out-going message preparation part, which requires computation for each edge. With this knowledge, our policy for the distributed phase must focus on balancing the out-degree. We select the *out-degree balanced* policy (O), which is one of the partitioning policies proved successful by PGX.D [61]. By the O policy, vertices are assigned to the first $|P| - 1$ partitions until the sum of vertex out-degrees in each partition reaches $|E|/|P|$. The last partition takes all remaining vertices.

The parallel phase For the parallel phase, we need to address the heterogeneity of the CPU(s) and the GPU(s). We cannot use here a degree balanced policy (such as O or IO from PGX.D), because the GPU can process much faster than the CPU when they have similar computation workload. TOTEM has proposed for the parallel phase two policies—HIGH and LOW—to handle heterogeneity. Given the results from both [41] and [113], which indicate that HIGH and LOW can both be successful for different algorithms and datasets, we select both the HIGH and LOW policies for independent use in the parallel phase.

The combined phase There is no policy that addresses the heterogeneity of the CPU and the GPU in distributed environments. Thus, we design a new policy in Section 6.3.4.

The random policy Last, we also select the random policy from the unfocused class for the distributed and the combined phases. The random policy is lightweight and easy to implement. We aim to use it as a control policy for the O policy in the distributed phase, and for the newly designed policy in the combined phase. We do not use the random policy in the parallel phase, because previous studies on the performance of TOTEM [41, 113] have shown that in most cases, the random policy is the worst performing one.

6.3.4 The Design of a Profiling-Based Greedy Policy

In this section, we design a *profiling-based greedy* (PG) policy for the combined phase of combined systems. The PG policy belongs to the computation-focused class.

Requirements. Combined systems need to directly place vertices on CPUs and GPUs in a single combined phase (see Section 6.3.1). To balance the workload, the partitioning policy of combined systems needs to address the heterogeneity of CPUs and GPUs.

None of the existing partitioning policies is able to deal with this type of heterogeneity. For existing computation-focused policies, many of them are proposed to distribute graphs for CPU-based systems on homogeneous clusters, but these policies do not focus on GPUs. Other computation-focus policies, such as the HIGH and LOW policies used in TOTEM, are designed without considering distributed environments.

General idea of the new PG policy. To balance the assignment of vertices across both CPUs and GPUs, the policy needs to assess the relative computation abilities of the GPU and the CPU. We use the ratio of the GPU and CPU capabilities, r, to estimate a balanced workload—i.e., a workload ratio of $r : 1$ for the GPU versus the CPU constitutes a balanced workload. These capabilities can be obtained from an offline profiling process as introduced in Section 6.2 and discussed in more detail in Section 6.3.5. Our policy is inspired by streaming partitioning policies: we treat the vertex list as a stream, and we place the vertices from this stream, one-by-one, in the partition currently having the smallest computation workload. The PG policy simplifies the process of graph partitioning by considering only one vertex at a time, and achieves balanced partitions for CPUs and GPUs.

Technical details. We define the computation workload on the CPU as the sum of the vertex out-degrees of all vertices in the partition placed on the CPU. The computation workload on the GPU is the similar sum computed for vertices placed on the GPU, but divided by r to account for the computation ratio of the GPUs to CPUs. In the PG policy, we maintain an array, indexed by partition, of the computation workload of all partitions. For each next vertex, we search for the partition with the smallest workload, and place it there. We update the computation workload of this partition by adding to it the out-degree of the added vertex. If the partition is for a GPU and the required memory is too close to the GPU memory capacity, the partition is removed from the computation workload array and will not be further considered for the remainder of the partitioning process. When all GPU partitions are full, all remaining vertices go to the CPUs.

Limitations. The computation ratio of the GPUs to CPUs must be known for the PG partitioning policy to work. The profiling process to calculate this ratio is time consuming, requiring many experiments (see Section 6.3.5). The more experiments, the more accurate the computed ratio is. Because the partition quality of this policy strongly relies on the accuracy of this ratio, when hardware infrastructure changes, the profiling process should be re-executed for better accuracy.

Comparison with other policies. The complexity of the PG policy is low, because it only needs to maintain the computation workload array for all partitions and to search for the partition with least workload for each assignment. DP and PD systems use two-phase partitioning, combining two policies, which means they access each vertex at least twice, to decide its final partition. Moreover, although O and Random have low complexity, HIGH and LOW need time-consuming sorting of vertices. We expect PG to be faster than PD and DP. We further explore the partitioning time in Section 6.4.6.

6.3.5 Implementation Details

In this section, we describe the non-trivial elements of implementing the three families of distributed heterogeneous systems, and of their partitioning policies. We begin from the open-source code of TOTEM, which already implements the general single-machine model introduced in Section 6.2. To this code, we add a profiling component to determine r, the ability to operate as a *distributed* system with any of the three architectures, and the partitioning policies.

Profiling the relative computation ability of the CPU and of the GPU. Previous studies have shown that heterogeneous systems can outperform CPU-only or GPU-only systems in many application domains, including graph processing, but the performance gain is very sensitive to a good workload partitioning [41, 117]. Determining a good workload partitioning is equivalent, in the case of our systems, to computing the right α, i.e., the right workload fraction to be placed on the CPU (Section 6.2). This fraction depends on the relative ability of the CPU and GPU to process a given workload, i.e., how much slower is the CPU compared to the GPU. Previous studies have already shown that using hardware performance models is unfeasible [91], and using the theoretical bounds of the hardware platforms does not give accurate results [90, 117]. Therefore, we propose a profiling method to understand the computation heterogeneity between these processing units. In our offline profiling method, we let the CPU and the GPU compute the same workload, and calculate the ratio of the CPU run time to the GPU run time. Due to the irregular, data-dependent nature of graph processing, we repeat the experiment for multiple runs and compute an average (see Section 6.4.2 for details) to obtain an accurate execution profile describing r.

Ideally, the most accurate values of r are computed using the graphs and algorithms that are to be used at runtime. However, such a profiling method would be too expensive to use in practice, and would cancel out the partitioning speed of our selected partitioning policies. Therefore, we choose to trade-off accuracy for applicability, and implement our profiling method using a 4-step micro-benchmarking strategy.

Step1. We select a representative graph processing algorithm. Specifically, we use PageRank because it is stable in its performance (e.g., by contrast, BFS shows high performance variability depending on the root of the search).

Step2. We use five synthetic datasets: Scale-20 to Scale-24, created by the Graph500 generator [48].

Step3. We randomly partition each graph and set 10% to 50% (with a step of 10%) total edges on the CPU and the remaining (90% to 50%) on the GPU. Then, we reverse the workload of the CPU and the GPU for each partitioning. Thus, we can obtain 10 pairs of CPU and GPU run times for processing the same workloads. We calculate the computation ratio r as the CPU run time over the GPU run time. We repeat the process with 5 random seeds for partitioning, and derive a mean value of r for each workload.

Step4. We observe the correlation between r and the different graph sizes (because there is no single value of r for all graph sizes), and we determine how to select r for different graphs (Section 6.4.2). We then calculate the fraction α as $\alpha = \max\{\alpha_l, 1/(r+1)\}$, where α_l is derived from the limitation to ensure that the GPU is not out of memory. As we know the data structures for representing partitions on GPUs, α_l can be estimated using the vertex and edge counts of the partition.

Communication in the distributed system. To connect the CPUs and GPUs on multiple machines, we extend the communication part of the TOTEM system. We use MPI [102] and, where available, the Nvidia GPUDirect [47] technology to communicate between processing units in our distributed heterogeneous systems. GPUDirect eliminates the copy process between the CPU and the GPU(s), which means messages can be directly transferred between each pair of processing units with low overhead. The usage of GPUDirect improves the performance of delivering messages and simplifies the coding

Table 6.2: Graph-partitioning policies for partitioning phases.

Phase	Policies
Distributed phase	Out-degree balanced (O), Random (R)
Parallel phase	HIGH (H), LOW (L)
Combined phase	Profilling-based Greedy (PG), Random (R)

effort. We use MPI barriers to ensure that all messages are synchronously delivered.

Implemented Policies. Based on the selection and design of partitioning policies, we summarize in Table 6.2 the policies we consider and implement for the partitioning phases of the three families of systems.

Other distributed systems aspects. We deploy the graph partitioner on a master machine. For each family of distributed heterogeneous systems, we implement all the partitioning policies or policy combinations we have selected or created in Sections 6.3.3 and 6.3.4, respectively. After partitioning the input graph, the master sends all data to the working machines, partition by partition. Working machines simultaneously reconstruct (build) their partitions on each processing unit. We use the master to control the process of executing the graph algorithm. For each iteration in the execution of the graph algorithm, the master collects information from working machines, checks if all partitions have finished their execution, and determines if execution should be stopped. The master is also responsible for aggregating updates from all partitions to the original graph.

6.4 Experimental Results

In this section, we present the experiments conducted to evaluate the performance of our three families of distributed heterogeneous systems. We introduce our experimental setup in Section 6.4.1, and summarize it in Table 6.3. Our experiments include an evaluation of the profiling method (Section 6.4.2) and a thorough evaluation of our three families of distributed heterogeneous systems, using algorithm run time (Section 6.4.3), a breakdown of algorithm run time (Section 6.4.4), and scalability (Section 6.4.5). We further analyze the partitioning time for different policies (Section 6.4.6), and provide a performance comparison between our systems and other graph-processing systems (Section 6.4.7).

6.4.1 Experimental Setup

Hardware: We conduct our experiments on the DAS4 cluster [21]. All machines we used in our experiments are equipped with an Nvidia GeForce GTX 480 GPU (1.5 GB onboard memory) and an Intel Xeon E5620 2.4 GHz CPU (24 GB memory). The machines are connected by 24 Gbit/s InfiniBand. For the scalability test, we vary the number of working

Table 6.3: Experimental setup of the experiments in Section 6.4.

Section	Algorithms	Datasets	Metric	Machines
6.4.2	All	Scale-20 to Scale-25	Algorithm run time	1, 4
6.4.3	All	G1 to G5	Algorithm run time	4
6.4.4	PageRank	G4	Breakdown	4
6.4.5	All	G4, G5	Scalability	1-10
6.4.6	-	Scale-20 to Scale-25	Partitioning time	1-10
6.4.7	All	G1 to G5	Algorithm run time	1, 4

Table 6.4: Summary of datasets used in the experiments.

	Graph	$\|V\|$	$\|E\|$	**d**	$\bar{\mathbf{D}}$
G1	WikiTalk (D)	2,388,953	5,018,445	0.1	2
G2	DotaLeague (U)	61,171	101,740,632	2,719.0	1,663
G3	Datagen_p10m (D)	9,749,927	687,174,631	0.7	70
G4	Scale-25 (U)	17,062,472	1,047,207,019	0.4	61
G5	Friendster (U)	65,608,366	3,612,134,270	0.1	55

$|V|$ and $|E|$ are the vertex count and edge count of the graphs, **d** is the link density ($\times 10^{-5}$), and $\bar{\mathbf{D}}$ is the average vertex out-degree. (D) and (U) stand for the original directivity of the graph. For each original undirected graph, we transform it into a directed graph (see Section 6.4.1).

machines from 1 to 10. Our systems need one extra machine as the master.

Algorithms: Based on our literature survey on graph processing [51], we select 3 popular graph-processing algorithms. These are Breadth First Search (BFS), PageRank, and Weakly Connected Component (WCC). We use the same implementation as in our previous study [56]. For BFS on each graph, we use the same source vertex. For PageRank, we set the maximum number of iterations to 10 as the only termination condition. WCC does not have any specific configuration.

Datasets: We select 5 graphs with various characteristics, as seen in Table 6.4. We include two real-world graph from SNAP [121] (i.e., WikiTalk and Friendster), and one from the Game Trace Archive [54] (i.e., DotaLeague). We also use two synthetic graphs, Scale-25 and Datagen_p10m, from Graph500 [48] and the LDBC generators [81], respectively. For undirected graphs (G2, G4, and G5), we use two directed edges to represent an undirected edge, as required by the CSR format. The WCC algorithm decides two vertices are connected if there is an edge between them. Thus, for the WCC algorithm on directed graphs (G1 and G3), for each pair of vertices that are connected with only a single directed edge, we create a reverse. The new graphs are *G1WCC* and *G3WCC*, with edge counts 9,313,364 and 1,374,349,262, respectively.

Notation for system-policy configuration: We selected different policies for different phases of three families of systems (Table 6.2). We use the notation of "System-Policy/Policies" to denote the system-policy configuration. For example, C-PG stands for the combined system using the PG policy, and DP-OH indicates the DP system using the O and H policies. The DP and PD systems need α as an input parameter, and C-PG needs

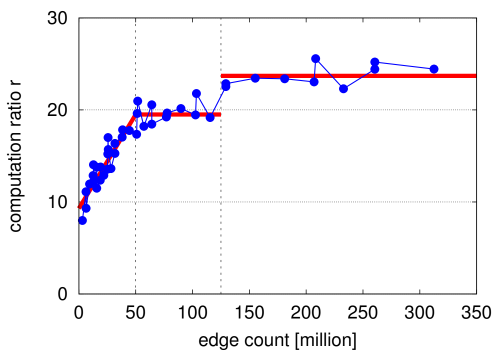


Figure 6.3: The relationship between the computation ratio and the edge count.

the computation ratio r. C-R does not need any input parameter.

Further configuration and settings: We use CUDA 5.5 as the GPU compiler, Intel TBB 4.1 [65] for sorting vertices by their out-degrees, and Open Mpi 1.8.2 for sending messages. We repeat each experiment 10 times and report the mean value. We do not show error bars because the results from different runs are stable, with the largest variance under 5%.

6.4.2 Calculating the Computation Workload Fraction

In this section we discuss the computation ratio r for our machines, and we derive the computation workload fraction α for CPUs for our graphs.

Key findings:

- The computation ratio r varies with the number of processed edges.
- The values of α obtained from PageRank can help BFS and WCC achieve good performance.

Following our micro-benchmarking strategy (Section 6.3.5), we obtain different values for r, all ranging between 8.0 to 25.0. Figure 6.3 shows how r relates to the number E_m of millions of processed edges. Using regression for the smallest graphs and approximating r as being constant for the other ranges, we find the following trends for r:

$$r = \begin{cases} 9.3 + 0.2 \times E_m, & 0 < E_m \leq 50 \\ 19.5, & 50 < E_m \leq 125 \\ 23.7, & 125 < E_m \leq 350 \end{cases}$$

When using multiple machines, each machine will have a value of r that corresponds to the number of edges it has to process. In our experiments, the edges are evenly distributed, because we use identical machines and a load-balancing driven policy. Thus, we can use the same value of r for all machines. Because of the GPU memory limitation, the

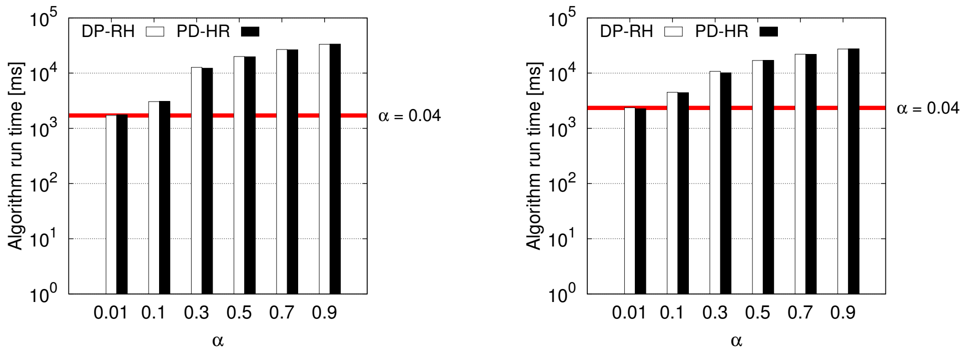


Figure 6.4: The algorithm run time for BFS (left) and WCC (right) for different values of α (vertical axes have a logarithmic scale).

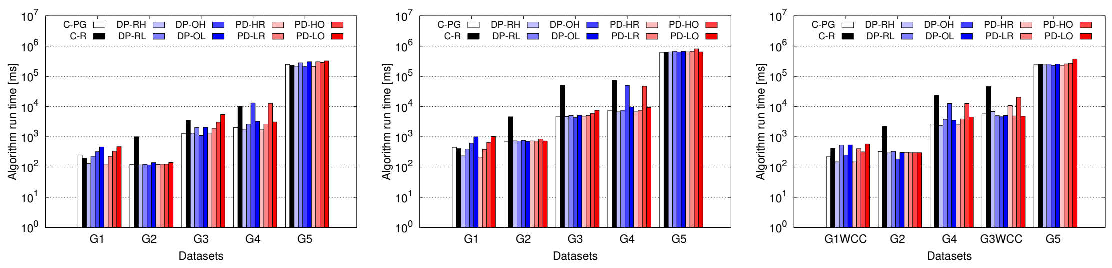


Figure 6.5: The algorithm run time of BFS (left), PageRank (middle), and WCC (right) on 5 datasets for all system-policy configurations (vertical axes have a logarithmic scale). Datasets are sorted in increasing order of their edge counts.

maximum value of E_m per working machine is about 350. All these values are likely to change for different machine configurations (i.e., different GPUs and CPUs).

We calculate α for each experiment in the following sections with different machine counts and graphs. For example, α for graphs G1 to G5 on 4 working machines is 0.09, 0.06, 0.04, 0.04, and 0.85, respectively.

Our micro-benchmarking strategy uses PageRank for determining α (Section 6.3.5). We preserve the same values for α for all 3 algorithms. To determine how suitable α is for the other algorithms, we run BFS and WCC on the G4 graph on 4 working machines using the DP-RH and PD-HR configurations with different values of α. We compare the algorithm run time of using these values with the one obtained using the calculated value of 0.04. Figure 6.4 shows these results (the horizontal line represents the algorithm run time for $\alpha = 0.04$ of DP-RH, which is very similar to PD-HR). For both BFS and WCC, the calculated α leads to the best performance, with the only exception when using the value of 0.01 of PD-HR.

6.4.3 Overview of the Performance of Three Families of Systems

In this section, we analyze algorithm run time, defined as the time for actually executing the graph algorithm; algorithm run time does not include the time spent on operations like initialization and result aggregation, and includes no system overhead [56].

Key findings:

- There is no overall winner, but C-R is in general the worst performing architecture.
- Our new PG policy for combined systems shows good performance.

Figure 6.5 shows the algorithm for all combinations of algorithms, systems, and datasets. The results are similar for all algorithms: no system-policy configuration outperforms the others in all cases. C-PG is typically in Top 3. C-R performs the worst because it has no consideration for the heterogeneity of the CPU and the GPU. When we fix the policy used for the distributed phase, and change the policy for the parallel phase, we can compare the influence of the HIGH and LOW policies. In almost all cases, the performance of the HIGH policy is better. We also notice that for G5, the performance of different system-policy configurations is very similar. This happens because $\alpha = 0.85$, and therefore the CPU dominates the overall algorithm run time. For G4, we need to set α to 0.35 for DP-OH and PD-HO to ensure that all partitions assigned to GPUs do not break the GPU memory limitation. We further analyze this setting of α for DP-OH in Section 6.4.4.

6.4.4 Breakdown of Algorithm Run Time

We further break down the algorithm run time into CPU- and GPU-computation time, and communication time, and discuss their impact on performance.

Key findings:

- The computation time is the dominant part of the algorithm run time.
- C-PG can achieve a balanced intra- and inter-machine computation workload.
- The O policy may lead to poor performance and needs to be tuned.

The breakdown of the algorithm run time per machine of PageRank on G4 is presented in Figure 6.6 for the C-PG and DP-OH configurations. The communication time is significantly shorter than the computation time (i.e., the maximum of the CPU time and the GPU time), which is empirical evidence for our discussion on message aggregation in Section 6.3.3. For C-PG, the computation times of the working machines are close to each other, and within each machine, the difference between the CPU time and the GPU time is also small. However, for DP-OH, the computation workload is not balanced across the CPU and the GPU, because DP-OH needs to set α to 0.35 in order to hold all partitions on

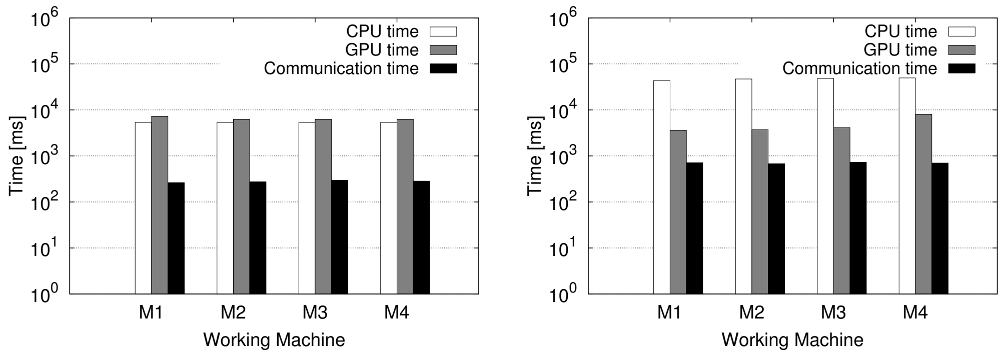


Figure 6.6: The breakdown of the algorithm run time of C-PG (left) and DP-OH (right, $\alpha = 0.35$) when running PageRank on graph G4 on each of the four working machines (vertical axes have a logarithmic scale).

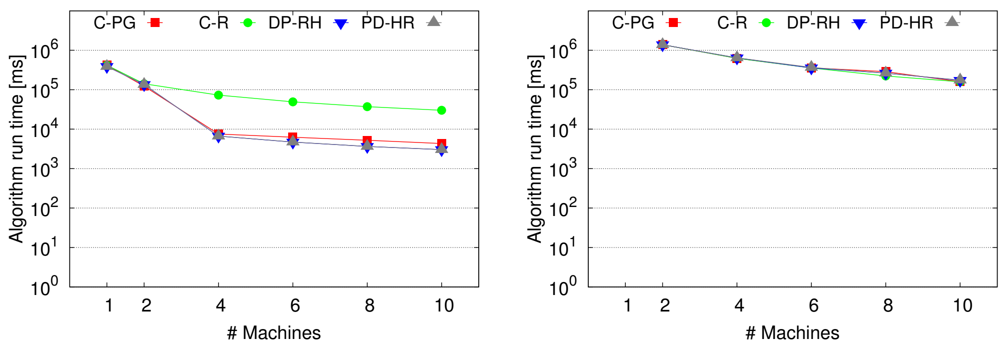


Figure 6.7: The scalability of running PageRank for G4 (left) and G5 (right) (vertical axes have a logarithmic scale). We failed to run G5 on 1 machine due to resources limitation.

GPUs. Although the edge counts on the GPUs are balanced by DP-OH, the fourth GPU partition has 200 times more vertices than the first GPU partition. This imbalance of the vertex counts is caused by the behavior of the O policy and the input graph G4. In G4, high-degree vertices have small vertex IDs and are assigned to the first machine (and then to the first GPU) by the O policy. The O policy needs tuning to avoid such imbalance.

6.4.5 Scalability

In this section, we discuss the scalability of our systems using a number of working machines from 1 to 10.

Key finding:

- Our three families of systems show good scalability.

From Section 6.4.3, we select the best performing system-policy configurations. These are C-PG for the Combined systems, DP-RH for the Distributed-Parallel systems, and PD-HR for the Parallel-Distributed systems. We also select C-R for comparison. Figure 6.7 depicts the algorithm run time of PageRank for the G4 and G5 graphs (the results

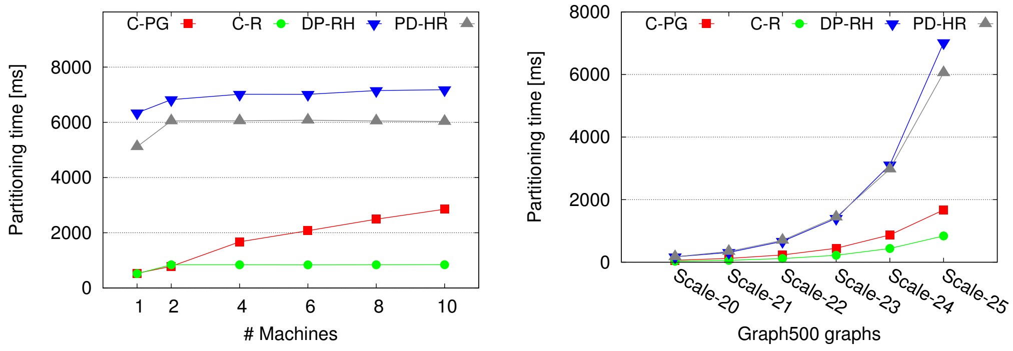


Figure 6.8: The time spent on partitioning the Scale-25 graph into different numbers of partitions (left) and on partitioning Graph500 graphs into 8 partitions on 4 machines (right).

we obtained for BFS and WCC are similar, and therefore not included). For G4, using up to 4 machines leads to excellent scalability. The values of α for 1, 2, and 4 machines are 0.8, 0.5, and 0.04, respectively, meaning that increasingly more workload is placed on the GPUs, and is therefore accelerated. However, when using more than 4 machines, the performance gain is not significant, simply because G4 is not large enough to stress larger clusters. For the graph G5, such a scalability limitation is not visible when using up to 10 machines. Even for 10 machines, the algorithm run time is still heavily dominated by the execution on the CPUs.

6.4.6 Partitioning Time

In this section, we analyze the time spent on partitioning graphs for different system-policy configurations.

Key findings:

- C-PG has shorter partitioning time than DP-RH and PD-HR. Its partitioning time increases with partition count.
- The size of graphs can significantly influence the partitioning time, especially for DP and PD systems.

We run two sets of experiments, one for partitioning Scale-25 with increasing partition count (i.e., using an increasing number machines), see Figure 6.8 (left), and one for partitioning 6 Graph500 graphs (Scale-20 to Scale-25) into 8 partitions on 4 machines (4 CPU and 4 GPU partitions), see Figure 6.8 (right). Since the results of different configurations of DP and PD systems are similar, we only present the results of DP-RH and PD-HR. The time for partitioning Scale-25 of C-PG increases linearly with the partition count, as for each vertex the operation of searching for the partition with the smallest workload has to be performed. As the time required for this operation increases with the partition count,

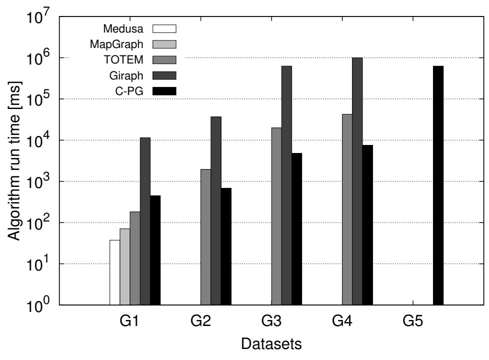


Figure 6.9: The algorithm run time of PageRank on 5 datasets with different graph processing systems (the vertical axis has a logarithmic scale, missing bars are explained in the text).

more time is consumed by C-PG. However, compared with the time-consuming sorting operation in the HIGH and LOW policies used in DP and PD systems, the partitioning time of C-PG is much shorter. Moreover, the gap will increase as the graphs grow larger, see Figure 6.8 (right).

6.4.7 Comparison with Other Graph Processing Systems

In this section, we compare the performance of our systems with other graph-processing systems, including GPU-enabled systems (Medusa [152], MapGraph [39], and TOTEM [41]), and a distributed CPU-based system (Giraph [43]).

Key finding:

- Our system can process all 5 datasets and achieves good performance compared with the other systems.

We select C-PG, and we deploy both our system and Giraph on 4 working machines. The other three systems work on one machine (altough MapGraph claims to be usable on GPUs of multiple machines, the latest publicly available version tested in this section can only work on a single machine). For TOTEM, we set α for each graph according to the rules introduced in [41]. We use the HIGH policy in TOTEM, as it outperforms the LOW policy for our selected datasets.

We run the PageRank algorithm on each system for graphs G1 to G5 and show the algorithm run time in Figure 6.9. Although Medusa and MapGraph can process the smallest graph G1 much faster, they both fail to run already on the medium-sized graph G2 due to the limited GPU memory. TOTEM fails to run on the graph G5 during the graph partitioning step. In contrast, our system can process all graphs. On the graph G4, our system using 4 machines is about 6 times as fast as the single-machine TOTEM. This super-linear speed-up is due to getting much more acceleration from the GPUs. Finally, our system outperforms Giraph significantly (by a factor of more than 50). The reasons include the acceleration of using GPUs and the reduced communication workload in our system. Giraph fails to run for G5 with 4 machines, but successfully executes with 20 machines, with an algorithm run time closes to that of C-PG with 4 machines. Similar results are observed when running BFS and WCC.

In our previous work [56], we found that TOTEM takes around 7,000 ms to finish PageRank on G4 with 8 GPUs (and no CPUs) in a single machine. In contrast, as shown in Figure 6.7, C-PG takes only about 5,300 ms for the same job on a distributed system with 8 machines with one CPU and one GPU each. We attribute this performance gain to two reasons: (1) the CPUs play an important role in the performance of our distributed systems, and (2) the inter-machine communication is not really a bottleneck.

6.5 Related Work

There are three directions of research that contribute to the success of our work: designing graph-processing systems, graph partitioning, and workload partitioning for heterogeneous systems. In this section we place our work in the context of each of these research directions.

Graph-processing systems. There are tens of graph processing systems developed in the past 10 years, each one designed with specific requirements in mind. Among these requirements, support for large-scale graph and efficient use of existing hardware infrastructure are often the most important ones. For example, Pregel [92], Giraph [43], or PGX.D [61] are distributed CPU-based systems that offer a simple, high-level programming model and focus on processing very large graphs with reasonable performance and very good scalability. Other systems, like TOTEM [41], Medusa [152], Gunrock [137], focus on offering users efficient ways to accelerate their graph processing using GPUs on a single machine. Despite their high performance, these systems cannot handle large-scale graphs efficiently. In this chapter, we combine the advantages of both worlds: we are the first to design and evaluate three families of distributed heterogeneous graph-processing systems.

Graph partitioning. Many graph-partitioning policies and methods have been proposed in various research areas. In our previous work [53], we have summarized the characteristics of existing partitioning policies and classified them into different classes from different perspectives: edge-cut [92] and vertex-cut [45] , static [61] and dynamic [112], and traditional heuristics [32] and streaming policies [126]. In this chapter, we combine existing policies for parallel and distributed systems to address the 2-layer systems we have designed (DP and PD systems). We further propose a novel partitioning policy, inspired by the streaming policies, to tackle both the heterogeneity and the scale of GPU-enabled distributed systems.

Heterogeneous systems. A lot of work has been recently dedicated to the efficient use of heterogeneous, CPU+GPU systems [60, 118, 123]. Most of this work focuses on workload partitioning - static or dynamic - between the different processing units in the system. In this chapter, we draw inspiration from static workload partitioning, which uses an estimation of the relative compute capabilities of the processing units - CPUs and GPUs - to compute an efficient partitioning before runtime. We adapt and extend the state-of-the-art profiling-based approach from [117] to a method that determines the right fraction of edges per processing unit. This fraction is an important parameter for our graph partitioner.

6.6 Summary

In this chapter, we bridge the gap between large-scale systems and accelerated systems for graph processing by designing three families of distributed heterogeneous systems. Each family focuses on a different partitioning architecture—Distributed-Parallel, Parallel-Distributed, or Combined. We combine promising policies for the DP and PD systems, and propose a new policy for the C sytems. To tackle heterogeneity while partitioning, we adapt and extend a profiling-based method to compute the workload fractions for the CPU(s) and the GPU(s).

For the implementation of systems, we address several technical challenges, such as implementing communication of CPUs and GPUs on multiple machines and building partitions independently on each processing unit.

To evaluate performance, we conduct experiments for all three families of systems, using different partitioning policies. Our results demonstrate the feasibility of distributed heterogeneous systems for graph processing. Performance-wise, the systems are competitive with the state-of-the-art.

Chapter 7

Conclusion and Future Work

In this chapter, we first summarize the contributions and findings of this thesis in Section 7.1. We then propose several directions for future research in Section 7.2.

Processing graphs, especially at large scale, is an increasingly useful activity in a variety of business, engineering, and scientific domains, such as online social networks, online retail, and online gaming. To process large-scale graphs and complex graph algorithms, many graph-processing systems, using or not using accelerators, have been designed and developed. Graph processing has become a very popular research topic in recent years. There are many challenges in this area, such as data collection and sharing, system performance evaluation and comparison, and system design and tuning.

In this thesis, we have designed and maintained the Game Trace Archive, which provides many graphs from online games. We have proposed an empirical method and conducted comprehensive experiments to understand the performance of CPU-based graph-processing systems. We have extended this empirical method and run experiments to evaluate the performance of GPU-enabled graph-processing systems. We have noticed that graph partitioning is essential to the performance of distributed graph-processing systems, and we have designed new partitioning policies with good performance. We have designed three families of distributed heterogeneous graph-processing systems and compared their performance with existing graph-processing systems.

7.1 Conclusion

We summarize our main contributions and findings as follows:

1. **RQ-1: How to build a virtual meeting space for sharing, exchanging, and analyzing graphs?** We design the Game Trace Archive (GTA) to be a virtual meeting space for the community. We identify five main requirements to build an archive

for game traces, and address them with the GTA. We propose a unified format for game traces. Gaming graphs, which are commonly exist in various games, can be presented in this format. We introduce a number of tools associated with the format. With these tools, we collect, process, and analyze 9 game traces. We collect in the GTA traces corresponding to more than 8 million real players and more than 200 million information items, spanning over 14 operational years. We also show that the GTA can be extended to include a variety of real game trace types. Analyzing gaming graphs may guide the multi-disciplinary field of game design by providing new understanding of player behavior. For example, we find that a correlation between the interactivity of match-based games and the retention of players over both long and short term, that friendship does not always help to perform better in games, and that in match-based games each player explores tens of different play strategies over time.

2. **RQ-2: How well do CPU-based graph-processing systems perform?** We identify methodological and practical challenges of benchmarking graph-processing systems. To address these challenges, we design an empirical method and use it to evaluate six popular CPU-based graph-processing systems, including Hadooop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. Our method defines the processes of performance evaluation, and how to select performance metrics, datasets, and algorithms. We report the performance on four aspects: raw performance, resource utilization, scalability, and overhead. We obtain interesting findings, for example, there is no overall winner-system, Hadoop always performs worst, several distributed systems are unable to process all datasets for all algorithms due to the complexity of algorithms and the scale of graphs.

3. **RQ-3: How well do GPU-enabled graph-processing systems perform?** We extend the empirical method first proposed for solving RQ-2 to measure the performance of GPU-enabled graph-processing systems by identifying new performance aspects and metrics, and by selecting and including new datasets and algorithms. We implement the extended method on three GPU-enabled graph processing systems, including TOTEM, Medusa, and MapGraph. For GPU-enabled graph-processing systems, we focus on reporting five performance aspects: raw processing power, performance breakdown, scalability, and, new for GPU-enabled systems, the impact on performance of system-specific optimization techniques and of the GPU generation. Our comprehensive results show that single-machine GPU-enabled graph-processing systems can perform much faster than distributed CPU-based systems and that the ability of processing large-scale graphs of single-machine GPU-enabled systems is limited.

4. **RQ-4: How to design low-overhead graph-partitioning policies for distributed graph-processing systems?** We design new graph-partitioning policies for real world graph-processing systems. We model the run time of graph-processing systems. Our policies are designed to minimize the run time of graph-processing systems, which is different from the goals of many previous policies, such as minimizing communication and balancing the number of vertices on each working machine. We propose a method to identify important performance issues and run-time-influencing graph characteristics, based on which we design policies to obtain balanced partitions. We conduct comprehensive experiments to evaluate and compare the performance of our new policies and previous alternatives, by processing various graph applications and reporting a set of performance metrics. Our results demonstrate that our newly designed partitioning policies perform well, while existing streaming policies exhibit poor performance.

5. **RQ-5: How to design a distributed and heterogeneous graph-processing system?** We design and implement three families of distributed heterogeneous graph-processing systems, which bridge the gap between distributed CPU-based systems and non-distributed GPU-enabled systems. We design these systems with different partitioning architectures, including Distributed-Parallel, Parallel-Distributed, and Combined. We select and/or design new partitioning policies for each family of systems. To address the heterogeneity of the CPU and the GPU while partitioning, we adapt and extend a profiling-based method to assign different fraction of workload to CPU(s) and GPU(s). We evaluate the performance of these three families of systems with various combination of graph-partitioning policies and compare the performance of our systems with existing graph-processing systems. From our results, we see that our systems can achieve better performance than other graph-processing systems, such as Giraph and TOTEM.

7.2 Future Work

As we present in Figure 1.2, this thesis consists of three main parts, graph-processing applications, knowledge gaining about system performance, and system design. We propose future research directions for each part, respectively.

7.2.1 Using the Game Trace Archive

Beside graph processing, many research directions may benefit from the comprehensive game information stored in the GTA. In this section, we discuss the potential usage of the GTA.

1. *Game resource management.* Resource management is critical for game operators and players. Inadequate management can lead to service shut-down, player departure, etc. Knowledge about the evolution of the player graph, from the simple player count to the complex guild information, can enable accurate prediction of the needed resources. The GTA can also help in understanding the change of game workloads with different time patterns (diurnal, weekly, etc.). Idle resources of one game may be used to support other games or applications during a specific period in a day or week [13].

2. *Quality of Experience for players.* Using the information from the GTA can help in improving the Quality of Experience for players in many aspects, for example, building reputation systems [69]. Reputation systems are important for successful online games. The reputation of a player (ratee) is calculated from ratings given by the other players (raters). During the calculation, each rating could have its own weight, which may be assigned according to many metrics derived from the GTA data, such as the level and online time of rater, the relationship of rater and ratee in the gaming graph, etc.

3. *In-game advertisement.* Advertisement is an important source of revenue for game operators. Successful game advertisements should meet at least the following requirements: for advertisers, advertisements should attract more users than their expectation; for game operators, they should obtain more income from advertisement than the revenue loss due to the effect of user experience. To achieve this requirement, the content of advertisements should be well designed and the placement of advertisements should be well selected. If an advertisement interrupts player immersion, especially during important and time-limited tasks, the advertisement may fail. What is worse, this may even result in player departure. Analyzing the type and frequency of player in-game operations may help advertisers and game operators to integrate seamless advertisements.

7.2.2 Benchmarking Graph Processing Systems

In Chapters 3 and 4, we have introduced our empirical method to evaluate and compare the performance of graph-processing systems. In this section, we suggest two directions to enhance our method and to spread our results.

1. *A comprehensive benchmark for graph-processing systems.* To further address the methodological and practical challenges we have envisioned in Chapter 3, we have to extend our empirical method and to design a comprehensive benchmark for graph-processing systems. We can design the benchmark by including more algorithms and datasets, by designing a unified benchmarking process that simplifies

the engineering effort, and by visualizing performance results. Our research group is actually working on this benchmark, with cooperative effort from both academia and industry.

2. *New common knowledge for the community.* We have discovered many interesting results through our experiments. For example, Hadoop, overall, is the worst performer compared with existing popular graph-processing systems. These results have been noticed and accepted by researches in the graph-processing area, and helped the community avoid redundant work on evaluating the performance of Hadoop on graph processing and on designing Hadoop-based graph-processing systems. By conducting more experiments, we may be able to obtain new common knowledge, such as what is the best graph-processing systems for specific graph-processing algorithms.

7.2.3 Designing Graph Processing Systems

The performance of graph-processing systems can be improved by many techniques and tuning methods. In this section, we focus on the design of graph-partitioning policies and of distributed heterogeneous graph-processing systems.

1. *Graph-partitioning policies considering the heterogeneity of clusters and algorithmic variety in real-world graph processing.* Graph-processing systems are deployed on clusters with different hardware, such as machines with different processors, accelerators, and amount of memory, and networks with different types and topologies. For heterogeneous clusters, it may be necessary to consider modeling the capability of the *entire* hardware infrastructure and to design specific graph-partitioning polices. Graph algorithms, such as Breadth-First Search, may exhibit diverse behavior in each iteration. It is challenging to predict and balance the workload in each iteration, because we do not know what are the active vertices. Dynamic repartitioning polices may help solve this balancing problem.

2. *Enhanced distributed heterogeneous graph-processing systems.* We have designed and implemented three families of distributed heterogeneous graph-processing systems. To make them more practical and to improve their performance, we can extend our systems by covering more heterogeneous hardware infrastructures, designing a distributed graph partitioner with many partitioning policies, and exploring CPU and GPU optimization techniques and their impact on the performance.

3. *Support for mutating graphs.* Graphs are growing and changing over time. Many graph-processing systems and graph-partitioning polices are designed to address static graphs. It is time and resource consuming to repartition and reprocess old

data for mutating graphs, especially for graphs with large scale. Efficient support for mutating graphs is a very interesting and challenging topic for the design of partitioning policies and graph-processing systems.

4. *Support for property graphs.* Modern graphs have rich information (*properties*) associated with vertices and edges. Many existing graph-processing systems can only support processing graphs with few properties, such as weights and timestamps. To extract more interesting and useful information, analyzing graph with multiple properties is required, but very challenging.

Bibliography

[1] K. Andreev and H. Racke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[2] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. *Journal of the ACM*, 56(2):5:1–5:37, 2009.

[3] D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and St-Connectivity on the Cray MTA-2. In *International Conference on Parallel Processing*, pages 523–530, 2006.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. 2014.

[5] M. Balint, V. Posea, A. Dimitriu, and A. Iosup. User Behavior, Social Networking, and Playing Style in Online and Face to Face Bridge Communities. In *Annual Workshop on Network and Systems Support for Games*, pages 13:1–13:2, 2010.

[6] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[8] K. P. Burnham. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociol. Methods & Research*, 33(2), 2004.

[9] M. Capobianco and O. Frank. Graph Evolution by Stochastic Additions of Points and Lines. *Discrete Mathematics*, 46(2):133 – 143, 1983.

[10] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz. Graphalytics: a Big Data Benchmark for Graph-Processing Platforms. In *Graph Data-management Experiences & Systems*, pages 7:1–7:6, 2015.

[11] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. Multithreaded Clustering for Multi-level Hypergraph Partitioning. In *International Parallel and Distributed Processing Symposium*, 2012.

[12] Ü. V. Çatalyürek, K. Kaya, A. E. Sariyüce, and E. Saule. Shattering and Compressing Networks for Betweenness Centrality. In *International Conference on Data Mining*, pages 686–694, 2013.

[13] C. Chambers, W. Feng, S. Sahu, D. Saha, and D. Brandt. Characterizing Online Games. *Transactions on Networking*, 18(3):899–910, 2010.

[14] A. Chan, F. Dehne, and R. Taylor. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *International Journal of High Performance Computing Applications*, 19(1):81–97, 2005.

[15] F. Checconi and F. Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *International Parallel and Distributed Processing Symposium*, pages 425–434, 2014.

[16] K. Chen, P. Huang, and C. Lei. Game Traffic Analysis: an MMORPG Perspective. *Computer Networks*, 50(16):3002–3023, 2006.

[17] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan. GraphHP: A Hybrid Platform for Iterative Graph Processing. 2014.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms. *MIT Press*, 2009.

[19] CUDPP, 2015. http://cudpp.github.io/.

[20] P. Danzig, J. Mogul, V. Paxson, and M. Schwartz. The Internet Traffic Archive, 2008. http://ita.ee.lbl.gov/.

[21] DAS4, 2014. http://www.cs.vu.nl/das4/.

[22] K. Dempsey, K. Duraisamy, H. Ali, and S. Bhowmick. A Parallel Graph Sampling Algorithm for Analyzing Gene Correlation Networks. *Procedia Computer Science*, 4:136 – 145, 2011.

[23] A. Denault, C. Canas, J. Kienzle, and B. Kemme. Triangle-Based Obstacle-Aware Load Balancing for Massively Multiplayer Games. In *Annual Workshop on Network and Systems Support for Games*, pages 4:1–4:6, 2011.

[24] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *International Parallel and Distributed Processing Symposium*, pages 124–124, 2006.

[25] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazán, and J. Larriba-Pey. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. *Web-Age Information Management*, pages 37–48, 2010.

[26] N. Ducheneaut, N. Yee, E. Nickell, and R. Moore. The Life and Death of Online Gaming Communities: A Look at Guilds in World of Warcraft. In *the SIGCHI Conference on Human Factors in Computing Systems*, pages 839–848. ACM, 2007.

[27] N. Ducheneaut, N. Yee, E. Nickell, and R. J. Moore. Alone Together?: Exploring the Social Dynamics of Massively Multiplayer Online Games. In *the SIGCHI Conference on Human Factors in Computing Systems*, pages 407–416, 2006.

[28] B. Elser and A. Montresor. An Evaluation Study of BigData Frameworks for Graph Processing. In *IEEE International Conference on BigData*, pages 60–67, 2013.

[29] European Commission Annual Reports. In *Ecorys*, 2013.

[30] Facebook, 2015. https://www.facebook.com/.

[31] T. Falkowski, A. Barth, and M. Spiliopoulou. Dengraph: A Density-Based Community Detection Algorithm. In *International Conference on Web Intelligence*, pages 112–115, 2007.

[32] Family of Graph and Hypergraph Partitioning Software, 2015. http://glaros.dtc.umn.edu/gkhome/views/metis.

[33] W. Feng, D. Brandt, and D. Saha. A Long-Term Study of a Popular MMORPG. In *Annual Workshop on Network and Systems Support for Games*, pages 19–24, 2007.

[34] W. Feng and W. Feng. On the Geographic Distribution of On-Line Game Servers and Players. In *Annual Workshop on Network and Systems Support for Games*, pages 173–179, 2003.

[35] M. Ferdman, et al. Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2012.

[36] Friendster, 2014. http://www.friendster.com/.

[37] J. Fritsch, B. Voigt, and J. Schiller. The Next Generation of Competitive Online Game Organization. In *Annual Workshop on Network and Systems Support for Games*, 2007.

[38] Z. Fu, H. Dasari, B. Bebee, M. Berzins, and B. Thompson. Parallel Breadth First Search on GPU Clusters. In *IEEE International Conference on Big Data*, pages 110–118, 2014.

[39] Z. Fu, M. Personick, and B. Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Graph Data-management Experiences & Systems*, pages 2:1–2:6, 2014.

[40] Ganglia Monitoring System, 2014. http://ganglia.sourceforge.net/.

[41] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. *CoRR*, abs/1312.3018, 2013.

[42] B. Ghit, N. Yigitbasi, and D. Epema. Resource Management for Dynamic MapReduce Clusters in Multicluster Systems. In *Workshop on Many-Task Computing on Grids and Supercomputers*, 2012.

[43] Giraph, 2014. http://giraph.apache.org/.

[44] L. Golab, M. Hadjieleftheriou, H. Karloff, and B. Saha. Distributed data placement to minimize communication costs via graph partitioning. In *International Conference on Scientific and Statistical Database Management*, pages 20:1–20:12. ACM, 2014.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Conference on Operating Systems Design and Implementation*, pages 599–613, 2014.

[47] GPUDirect, 2015. https://developer.nvidia.com/gpudirect/.

[48] Graph500, 2014. http://www.graph500.org/.

[49] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. *Parallel Object-Oriented Scientific Computing*, 2005.

[50] A. Guerrieri and A. Montresor. Distributed Edge Partitioning for Graph Processing. *arXiv:1403.6270*, 2014.

[51] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report. Technical Report PDS-2013-004, Delft University of Technology, 2013.

[52] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *International Parallel and Distributed Processing Symposium*, pages 395–404, 2014.

[53] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema. Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies: A Technical Report. Technical Report PDS-2015-002, Delft University of Technology, 2015.

[54] Y. Guo and A. Iosup. The Game Trace Archive. In *Annual Workshop on Network and Systems Support for Games*, pages 4:1–4:6, 2012.

[55] Y. Guo, S. Shen, O. Visser, and A. Iosup. An Analysis of Online Match-Based Games. In *International Workshop on Massively Multiuser Virtual Environments*, pages 134–139, 2012.

[56] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema. An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems. In *International Symposium on Cluster, Cloud and Grid Computing*, pages 423–432, 2015.

[57] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking Graph-Processing Platforms: a Vision. In *International Conference on Performance Engineering*, pages 289–292, 2014.

[58] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-Like Graph Processing Systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[59] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International Conference on High Performance Computing*, pages 197–208. 2007.

[60] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *International Euro-Par Conference on Parallel Processing*, pages 235–246, 2010.

[61] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[62] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. *ACM SIGPLAN Notices*, 46(8):267–276, 2011.

[63] J. Hsieh and C. Sun. Building a Player Strategy Model by Analyzing Replays of Real-Time Strategy Games. In *Neural Networks*, pages 3106–3111, 2008.

[64] T. Ideker, O. Ozier, B. Schwikowski, and A. F. Siegel. Discovering Regulatory and Signalling Circuits in Molecular Interaction Networks. *Bioinformatics*, pages S233–S240, 2002.

[65] Intel TBB, 2015. https://www.threadingbuildingblocks.org/.

[66] B. Iordanov. HyperGraphDB: A Generalized Graph Database. *Web-Age Information Management*, pages 25–36, 2010.

[67] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. Epema. The Grid Workloads Archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[68] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[69] E. Kaiser and W. Feng. PlayerRating: A Reputation System for Multiplayer Online Games. In *Annual Workshop on Network and Systems Support for Games*, pages 1–6, 2009.

[70] K. Kambatla, G. Kollias, and A. Grama. Efficient Large-Scale Graph Analysis in MapReduce. In *International Workshop on Parallel Matrix Algorithms and Applications*, 2012.

[71] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *International Conference on Data Mining*, pages 229–238, 2009.

[72] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *International Conference on Parallel Processing*, pages 113–122, 1995.

[73] G. Karypis and V. Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[74] G. Karypis, K. Schloegel, and V. Kumar. ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. 1997.

[75] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-Scale Graph Processing. In *European Conference on Computer Systems*, pages 169–182, 2013.

[76] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *International Symposium on High-performance Parallel and Distributed Computing*, pages 239–252, 2014.

[77] J. Kinicki and M. Claypool. Traffic Analysis of Avatars in Second Life. In *International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 69–74, 2008.

[78] T. G. Kolda, A. Pinar, and C. Seshadhri. Triadic Measures on Graphs: The Power of Wedge Sampling. In *International Conference on Data Mining*, pages 10–18, 2013.

[79] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *International Conference on World Wide Web*, pages 591–600, 2010.

[80] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX Conference on Operating Systems Design and Implementation*, pages 31–46, 2012.

[81] LDBC, 2015. http://ldbcouncil.org/.

[82] Y.-T. Lee, K.-T. Chen, Y.-M. Cheng, and C.-L. Lei. World of Warcraft Avatar History Dataset. In *Annual ACM Conference on Multimedia Systems*, pages 123–128, 2011.

[83] J. Leskovec and C. Faloutsos. Sampling from Large Graphs. In *International Conference on Knowledge Discovery in Data Mining*, pages 631–636, 2006.

[84] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *International Conference on Knowledge Discovery in Data Mining*, pages 177–187, 2005.

[85] I. X. Leung, P. Hui, P. Liò, and J. Crowcroft. Towards Real-Time Community Detection in Large Networks. *Physical Review E*, 2009.

[86] LinkedIn, 2015. https://gb.linkedin.com/.

[87] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. 5(8):716–727, 2012.

[88] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.

[89] A. Lugowski, D. M. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *International Conference on Data Mining*, pages 930–941, 2012.

[90] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *International Symposium on Microarchitecture*, pages 45–55, 2009.

[91] S. Madougou, A. L. Varbanescu, C. de Laat, and R. van Nieuwpoort. An Empirical Evaluation of GPGPU Performance Models. In *International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, 2014.

[92] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *International Conference on Management of Data*, pages 135–146, 2010.

[93] MapGraph, 2015. http://mapgraph.io/.

[94] W. A. Mason and A. Clauset. Friends FTW! Friendship and competition in Halo: Reach. *CoRR*, abs/1203.2268, 2012.

[95] 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 Survey*, 48(2):25:1–25:39, 2015.

[96] D. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU Graph Traversal. In *Symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[97] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel Graph Partitioning for Complex Networks. *arXiv:1404.4797*, 2014.

[98] D. C. Montgomery, E. A. Peck, and G. G. Vining. *Introduction to Linear Regression Analysis*, volume 821. John Wiley & Sons, 2012.

[99] V. Nae, A. Iosup, and R. Prodan. Dynamic Resource Provisioning in Massively Multiplayer Online Games. *Transactions on Parallel and Distributed Systems*, 22(3):380–395, 2010.

[100] A. Narayanan and V. Shmatikov. De-Anonymizing Social Networks. In *Security and Privacy*, pages 173–187, 2009.

[101] Neo4j, 2014. http://www.neo4j.org/.

[102] Open MPI, 2015. http://www.open-mpi.org/.

[103] OrientDB, 2016. http://orientdb.com/.

[104] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1999.

[105] U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 2007.

[106] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In *International Symposium on Parallel and Distributed Processing*, pages 203–214, 2013.

[107] E. J. Riedy, D. A. Bader, and H. Meyerhenke. Scalable Multi-Threaded Community Detection in Social Networks. In *Workshop on Multithreaded Architectures and Applications*, 2012.

[108] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody Ever Got Fired for Using Hadoop on a Cluster. In *International Workshop on Hot Topics in Cloud Data Processing*, pages 2:1–2:5, 2012.

[109] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Symposium on Operating Systems Principles*, pages 472–488, 2013.

[110] I. Safro, P. Sanders, and C. Schulz. Advanced Coarsening Schemes for Graph Partitioning. *Journal of Experimental Algorithmics*, 2015.

[111] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Zhao. Sharing Graphs Using Differentially Private Graph Models. In *Internet Measurement Conference*, pages 81–98, 2011.

[112] S. Salihoglu and J. Widom. GPS: A Graph Processing System. Technical report, 2012.

[113] S. Sallinen, D. Borges, A. Gharaibeh, and M. Ripeanu. Exploring Hybrid Hardware and Data Placement Strategies for the Graph 500 Challenge. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[114] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An Efficient Matrix Computation with the MapReduce Framework. In *Cloud Computing Technology and Science*, pages 721–726, 2010.

[115] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *International Conference on Data Engineering*, pages 553–564, 2013.

[116] B. Shao, H. Wang, and Y. Li. The Trinity Graph Engine. Technical report, Technical Report 161291, Microsoft Research, 2012.

[117] J. Shen, A. L. Varbanescu, and H. Sips. Look Before You Leap: Using the Right Hardware Resources to Accelerate Applications. In *International Conference on High Performance Computing and Communications*, pages 383–391, 2014.

[118] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *International Conference on Computing Frontiers*, pages 14:1–14:10, 2013.

[119] S. Shen and A. Iosup. The XFire Online Meta-Gaming Network: Observation and High-Level Analysis. In *International Workshop on Massively Multiuser Virtual Environments*, 2011.

[120] SMEs, 2015. http://ec.europa.eu/growth/smes/.

[121] SNAP, 2014. http://snap.stanford.edu/index.html/.

[122] E. Solomonik, A. Buluç, and J. Demmel. Minimizing Communication in All-Pairs Shortest Paths. In *International Symposium on Parallel and Distributed Processing*, pages 548–559, 2013.

[123] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems. In *International Conference on Supercomputing*, pages 365–376, 2012.

[124] Sparksee, 2016. http://sparsity-technologies.com/.

[125] F. Spitzer and A. Mathematician. *Principles of Random Walk*. Springer, 1964.

[126] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230, 2012.

[127] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. CRC press, 1994.

[128] P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *International Semantic Web Conference on The Semantic Web*, pages 764–780. 2010.

[129] M. Suznjevic, I. Stupar, and M. Matijasevic. MMORPG Player Behavior Model Based on Player Action Categories. In *Annual Workshop on Network and Systems Support for Games*, pages 6:1–6:6, 2011.

[130] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka. Performance Characteristics of Graph500 on Large-Scale Distributed Environment. In *International Semantic Web Conference on The Semantic Web*, pages 149–158, 2011.

[131] P.-Y. Tarng, K.-T. Chen, and P. Huang. On Prophesying Online Gamer Departure. In *Annual Workshop on Network and Systems Support for Games*, pages 16:1–16:2, 2009.

[132] The Parallel Workloads Archive Team. Parallel Workloads Archive, 2007. http://www.cs.huji.ac.il/labs/parallel/workload/.

[133] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *International Conference on Web Search and Data Mining*, pages 333–342, 2014.

[134] M. Tsvetovat, J. Reminga, and K. Carley. DyNetML: Interchange Format for Rich Social Network Data. Technical Report CMU-ISRI-04-105, Carnegie Mellon University, 2004.

[135] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[136] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Annual Southeast Regional Conference*, pages 42:1–42:6, 2010.

[137] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Symposium on Principles and Practice of Parallel Programming*, 2015.

[138] D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *Workshop on Many-Task Computing on Grids and Supercomputers*, pages 8:1–8:10, 2009.

[139] D. J. Watts and S. H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393(6684):440–442, 1998.

[140] B. Weber and M. Mateas. A Data Mining Approach to Strategy Prediction. In *International Conference on Computational Intelligence and Games*, pages 140–147, 2009.

[141] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[142] Wikimedia, 2015. https://wikimediafoundation.org/wiki/FAQ/en/.

[143] R. J. Wilson. *Introduction to Graph Theory*. Academic Press, 1972.

[144] B. Wu and Y. Du. Cloud-Based Connected Component Algorithm. In *International Conference on Artificial Intelligence and Computational Intelligence*, pages 122–126, 2010.

[145] XFire, 2012. http://xfire.com/.

[146] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao. Heterogeneous Environment Aware Streaming Graph Partitioning. *Transactions on Knowledge and Data Engineering*, 27(6):1560–1572, 2015.

[147] YARN, 2014. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[148] J. Yeo, D. Kotz, and T. Henderson. CRAWDAD: A Community Resource for Archiving Wireless Data at Dartmouth. *SIGCOMM Computer Communication Review*, 36(2):21–22, 2006.

[149] B. Zhang, A. Iosup, and D. Epema. The Peer-to-Peer Trace Archive: Design and Comparative Trace Analysis. Technical Report PDS-2010-003, Delft University of Technology, 2010.

[150] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang. Efficient Graph Computation on Hybrid CPU and GPU Systems. *Journal of Supercomputing*, 71(4):1563–1586, 2015.

[151] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate Large-Scale Iterative Computation through Asynchronous Accumulative Updates. In *Workshop on Scientific Cloud Computing Date*, pages 13–22, 2012.

[152] J. Zhong and B. He. Medusa: Simplified Graph Processing on GPUs. *Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[153] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *USENIX Conference on Usenix Annual Technical Conference*, pages 375–386, 2015.

[154] X. Zhuang, A. Bharambe, J. Pang, and S. Seshan. Player Dynamics in Massively Multiplayer Online Games. *CMU-CS-07-158*, 2007.

[155] Zynga. CityVille, 2012. http://company.zynga.com/games/cityville.

Summary

Distributed Heterogeneous Systems for Large-Scale Graph Processing

Graph processing is increasingly popular in a variety of scientific and engineering domains. Consequently, graphs and graph-processing algorithms have become increasingly more diverse. Following the big data trend in every computer-related domain, graphs have also become increasingly larger. Processing graphs is requiring more sophisticated computer systems. Important for this thesis, graph-processing systems now need to combine scalability (a grand challenge in computer science) and raw processing power (an endless race), with efficiency especially in cost and energy requirements (a difficult to define and ensure non-functional property for computer systems). New trade-offs between these elements are offered by two important trends in computer systems. First, distributed systems have grown in popularity and cost-efficiency. Second, GPUs offer an excellent performance-energy ratio and are included in most modern computers. By combining distributed CPU-based systems and non-distributed GPU-enabled systems into distributed heterogeneous systems, large-scale graph processing may become possible and efficient. However, many challenges still exist in the area of graph processing before distributed heterogeneous systems can be well understood. In this thesis, we conduct fundamental and applied research to address three major challenges in three research directions of graph processing: application (understanding new data characteristics and sharing graphs), knowledge (evaluating and comparing the performance of various graph-processing systems), and design (designing new partitioning policies and entire graph-processing systems that can use both CPUs and GPUs on multiple machines).

In the application direction (Chapter 2), we design the Game Trace Archive (GTA), which is a virtual meeting space for exchanging and sharing gaming data and graphs. The GTA addresses five main requirements on building a trace archive in the online gaming domain. To facilitate the exchange and usage of game traces, within the GTA, we design a unified format to cover diverse types of game traces and graphs. We create many tools related to this format. By using these tools, we collect over 10 game traces from different gaming genres in the GTA, and share them through an online archive (http://gta.st.ewi.tudelft.nl/). These traces cover more than 8 million real players and more than 200 million gaming elements, spanning over 14 operational years. We also conduct an

analysis of game traces, with a focus on online match-based games. We obtain interesting and valuable information that may benefit the game operators to improve their services and to design future games.

In the knowledge direction (Chapter 3 and 4), we propose an empirical method on evaluating the performance of graph-processing systems, including both CPU-based and GPU-enabled systems. We envision seven methodological and practical challenges on benchmarking graph-processing systems. We define a four-step method to address these challenges. We select and define interesting performance metrics, implement a benchmarking suite with representative graph-processing algorithms and datasets, deploy the suite on many popular graph-processing systems, and report comprehensive performance results of these systems. Some of our findings have become the common knowledge of the graph community. For example, Hadoop, which is a widely-used generic data-processing system, has very poor performance on processing graphs. Based on our experience of using the graph-processing systems we tested, we also discuss the highlights and the limitations of these systems.

In the design direction (Chapter 5 and 6), we design graph-partitioning policies for distributed graph-processing systems and also three families of distributed heterogeneous graph-processing systems. To design new partitioning policies, we model the run time of of graph-processing systems. We propose a method to identify the graph characteristics that are closely related to the run time of graph-processing systems, based on the run time models of systems and the behavior of graph-processing algorithms. To balance these graph characteristics, we design a new low-overhead graph-partitioning policies and tune existing policies. To design distributed heterogeneous systems, we focus on exploring the design space of systems with different partitioning architectures. For each family of systems, we select and/or design graph-partitioning policies. To balance the workload on CPUs and GPUs, we profile the relative computation ability of the CPU and the GPU using a four-step micro-benchmarking strategy. We evaluate the performance of systems with different combinations of partitioning policies. We also compare the performance of our systems with other existing graph-processing systems. Experimental results show that our system can achieve better performance, and in some cases even be the only system that can complete the work without crashing.

To conclude, we design and maintain the first comprehensive Game Trace Archive to benefit graph and gaming researchers. We are the first to comprehensively evaluate and compare the performance of six popular CPU-based graph-processing systems and three popular GPU-enabled graph-processing systems. We propose a method to facilitate the design of graph-partitioning policies for different graph-processing systems. We are the first to design families of distributed heterogeneous graph-processing systems to bridge the gap between existing CPU-based systems and GPU-enabled systems.

Our work has already done a step toward understanding distributed heterogeneous systems for large-scale graph processing. However, more work remains to be done. In the future work, we plan to analyze the graphs stored in the GTA and extract more useful information, to design a comprehensive benchmark and evaluate more graph-processing systems, to enhance our current distributed heterogeneous systems, and to design graph-processing systems that can support mutating graphs and property graphs.

Samenvatting

Gedistribueerde Heterogene Systemen voor Grootschalige Graafverwerking

Graafverwerking wordt steeds populairder binnen verschillende wetenschappelijke en toegepaste domeinen. Hierdoor worden grafen en graafalgoritmen steeds diverser. Als gevolg van ‘Big Data’, een trend die vrijwel elk aspect van de informatica raakt, worden deze grafen ook steeds groter. Het verwerken van grafen vereist dus meer en meer geavanceerde computersystemen. Centraal in dit proefschrift staat het volgende probleem: heden ten dagen moeten graafverwerkingssystemen schaalbaarheid (een grote uitdaging in de informatica) en harde verwerkingscapaciteit (een eindeloze opdracht) combineren met efficiëntie in termen van kosten en energieverbruik (een eigenschap van computersystemen die moeilijk is te definiëren en te garanderen). Twee belangrijke ontwikkelingen binnen de informatica hebben ervoor gezorgd dat er hiertussen nieuwe afwegingen zijn ontstaan. Ten eerste, gedistribueerde computersystemen groeien in populariteit en kostenefficiëntie. Ten tweede, GPU’s (grafische verwerkingseenheden) zijn beschikbaar op de meeste moderne computers en bieden een zeer goede verhouding tussen prestaties en energieverbruik. Door gedistribueerde CPU-gebaseerde systemen te combineren met niet-gedistribueerde GPU-gebaseerde systemen zal grootschalige graafverwerking misschien uiteindelijk zowel mogelijk als efficiënt worden. Desalniettemin, er zijn nog vele uitdagingen op het gebied van graafverwerking voordat gedistribueerde heterogene systemen volledig begrepen worden. In dit proefschrift onderzoeken we drie belangrijke uitdagingen in drie verschillende onderzoeksrichtingen aan de hand van toegepast en fundamenteel onderzoek: toepassing (het begrijpen van nieuwe data-karakteristieken en het uitwisselen van grafen met anderen), kennis (het evalueren en vergelijken van de prestaties van verschillende graafverwerkingssystemen), en ontwerp (het ontwerpen van nieuwe partitioneringsstrategieën en het ontwikkelen van nieuwe graafverwerkingssystemen die gebruik maken van zowel CPU’s als GPU’s op verschillende computers).

Op het gebied van toepassing (Hoofdstuk 2), ontwerpen we de Game Trace Archive (GTA): een virtuele ontmoetingsplek voor het uitwisselen en delen van game-gerelateerde data en grafen. De GTA richt zich op vijf belangrijke eisen voor het maken van een archief voor game traces op het gebied van online games. Om de uitwisseling van game traces mogelijk te maken hebben we binnen de GTA een uniform opslagformaat ontworpen dat

de traces van verschillende soorten games ondersteunt. Ook hebben we verschillende hulpprogramma's ontwikkeld die werken met dit formaat. Met behulp van deze hulpprogramma's hebben we meer dan 10 game traces verzamelt van verschillende soorten games binnen de GTA en deze beschikbaar gemaakt door middel van een online archief (`http://gta.st.ewi.tudelft.nl/`). Deze traces bevatten de gegevens van meer dan 8 miljoen online spelers en meer dan 200 miljoen game elementen over een periode van 14 jaar. We hebben deze game traces geanalyseerd waarbij de nadruk lag op online wedstrijd-gebaseerde games. Hierbij vonden we interessante en waardevolle informatie die beheerders van deze games kunnen gebruiken bij het verbeteren van hun service en het ontwerpen van nieuwe games.

Op het gebied van kennis (Hoofdstuk 3 en 4), presenteren we een empirische methode voor het evalueren van de prestaties van graafverwerkingssystemen. Dit betreft zowel CPU- als GPU-gebaseerde systemen. We bekijken zeven methodologische en praktische uitdagingen die komen kijken bij het benchmarken van een graafverwerkingssysteem. We presenteren een methode die deze uitdagingen aanpakt in vier stappen. We selecteren en definiëren interessante metrieken voor het meten van de prestaties, implementeren een benchmark suite die bestaat uit algoritmen en data sets die representatief zijn voor graafverwerking, voeren deze benchmark suite uit op verschillende populaire graafverwerkingssystemen en beschrijven uitgebreid de resultaten van deze systemen. Sommige van onze bevindingen zijn onderhand uitgegroeid tot algemene kennis binnen het domein van de graafverwerking. Bijvoorbeeld, Hadoop is een zeer populair systeem voor het verwerken van algemene data, maar blijkt zeer slechte prestaties te leveren bij het verwerken van grafen en is dus niet geschikt voor dit type data. Aan de hand van onze ervaringen met verschillende graafverwerkingssystemen, bespreken we de positieve eigenschappen en de beperkingen van deze systemen.

Op het gebied van ontwerp (Hoofdstuk 5 en 6), presenteren we strategieën om grafen te partitioneren voor gedistribueerde graafverwerkingssystemen en voor drie soorten gedistribueerde heterogene graafverwerkingssystemen. Om deze partitioneringsstrategieën te kunnen ontwerpen hebben we de totale looptijd van verschillende systemen gemodelleerd. We presenteren een methode om de karakteristieken van een graaf te bepalen die een grote invloed hebben op de looptijd van een systeem aan de hand van deze modellen en het gedrag van verschillende graafalgoritmen. Op basis van de belangrijkste karakteristieken van grafen, presenteren we een nieuwe partitioneringsstrategie en stellen we de bestaande partitioneringsstrategieën beter af. Om een nieuw gedistribueerd heterogeen systeem te ontwerpen, verkennen we de ontwerpruimte van verschillende systemen aan de hand van verschillende partitioneringsstrategieën. Voor elk type systeem, selecteren of ontwerpen we een geschikte partitioneringsstrategie. Om het werk gelijkmatig over CPU's en GPU's te verdelen, onderzoeken we de relatieve verwerkingssnelheid van de CPU en GPU aan de hand van een micro-benchmark die bestaat uit vier stappen. We onderzoeken de prestaties van graafverwerkingssystemen met verschillende combinaties van partitioneringsstrategieën. We vergelijken ook de prestaties van ons systeem met andere bestaande systemen. Experimentele resultaten laten zien dat ons systeem beter presteert dan de anderen, voor sommige taken is ons systeem de enige die het werk kan voltooien zonder te crashen.

Tenslotte presenteren we het eerste uitgebreide archief voor game traces (Game Trace Archive) waar zowel onderzoekers op het gebied van grafen als op het gebied van games baat bij hebben. We zijn de eerste die een uitgebreide evaluatie uitvoeren en de prestaties vergelijken van zes populaire CPU- en drie GPU-gebaseerde graafverwerkingssystemen. We presenteren een methode om ondersteuning te bieden bij het ontwerpen van graaf-partitioneringsstrategieën voor verschillende soort graafverwerkingssystemen. We zijn de eerste die een gedistribueerd heterogeen graafverwerkingssystemen ontwerpen en hiermee het gat tussen CPU- en GPU-gebaseerde systemen kleiner maken.

Ons werk is een belangrijkste stap voorwaarts in het begrijpen van gedistribueerde heterogene systemen voor grootschalige graafverwerking. Echter, er is nog veel werk dat gedaan moet worden. In de toekomst willen wij ons richten op: het analyseren van de grafen in de GTA en hieruit zinvolle informatie halen, het ontwerpen van een uitgebreide benchmark voor het evalueren van graafverwerkingssystemen, het verbeteren van onze huidige gedistribueerde heterogene systeem en het ontwerpen van een graafverwerkingssysteem dat kan omgaan met grafen die veranderen over de tijd en grafen met data op de knopen en de links.

Biography

Yong Guo was born in Jingzhou, China, on December 25, 1988. Yong received his Bachelor degree in Computer Science and Technology and graduated as Excellent Graduate Student (top 2%) from National University of Defense Technology (NUDT), China, in 2009. Then, Yong started his master study in Computer Science and Technology at NUDT, without sitting the entrance examination. After one year and a half, due to his excellent performance, Yong was admitted as a PhD student at NUDT, one year ahead of his master graduation time. In September 2011, Yong became a PhD student in the Parallel and Distributed Systems group at Delft University of Technology (TUD), the Netherlands, with a four-year scholarship by the China Scholarship Council. In 2015, Yong did a three-month internship on graph processing and partitioning at Oracle Labs, in the USA. In 2013, 2014, and 2016, Yong was a teaching assistant for the master course of Distributed Computing Systems at TUD. Yong's research work focuses on designing large-scale graph-processing systems, on evaluating the performance of graph-processing systems, and on analyzing player behavior and evolution of online games.

Publications

1. **Yong Guo**, Sungpack Hong, Hassan Chafi, Alexandru Iosup, and Dick Epema, "Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies", *Journal of Parallel and Distributed Computing (JPDC)*, http://dx.doi.org/10.1016/j.jpdc.2016.02.003, 2016.

2. **Yong Guo**, Ana Lucia Varbanescu, Dick Epema, and Alexandru Iosup "Design and Experimental Evaluation of Distributed Heterogeneous Graph-Processing Systems", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

3. **Yong Guo**, Ana Lucia Varbanescu, Alexandru Iosup, and Dick Epema, "An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

4. Alexandru Iosup , Mihai Capotă, Tim Hegeman, **Yong Guo**, Wing Lung Ngai, Ana Lucia Varbanescu, Merijn Verstraaten, "Towards Benchmarking IaaS and PaaS Clouds for Graph Analytics", *Workshop on Big Data Benchmarking (WBDB)*, 2014.

5. Alexandru Iosup, Siqi Shen, **Yong Guo**, Stefan Hugtenburg, Jesse Donkervliet, Radu Prodan, "Massivizing Online Games Using Cloud Computing: A Vision", *IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, 2014.

6. **Yong Guo**, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke, "How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis", *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2014.

7. **Yong Guo**, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke, "Benchmarking Graph-Processing Platforms: A Vision", *ACM/SPEC international conference on Performance engineering (ICPE)*, 2014.

8. **Yong Guo** and Alexandru Iosup, "The Game Trace Archive", *Annual Workshop on Network and Systems Support for Games (NetGames)*, 2012.

9. **Yong Guo**, Siqi Shen, Otto Visser, and Alexandru Iosup, "An Analysis of Online Match-Based Games", *International Workshop on Massively Multiuser Virtual Environments (MMVE)*, 2012.