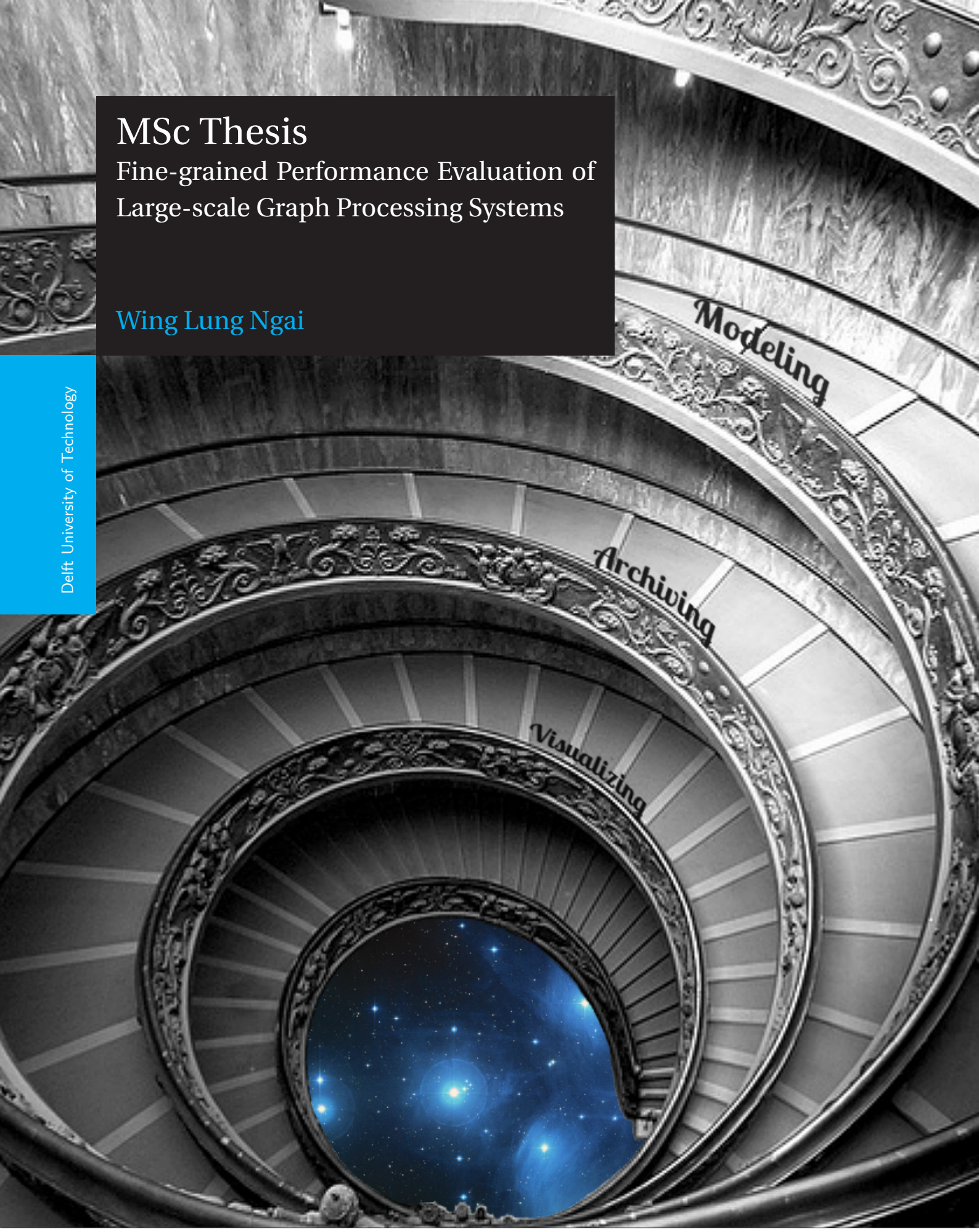
MSc Thesis

Fine-grained Performance Evaluation of Large-scale Graph Processing Systems

Wing Lung Ngai



Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

MSc Thesis

Fine-grained Performance Evaluation of Large-scale Graph Processing Systems

by

Wing Lung Ngai

in partial fulfillment of the requirements for the degree of

Master of Science

in Computer Science

at the Delft University of Technology,

to be defended publicly on Tuesday June 30, 2015 at 02:00 PM.

Supervisor:	Ir. M. Capotă,	
	Dr. ir. A. Iosup	
Thesis committee:	Prof. dr. ir. D.H.J. Epema,	Delft University of Technology
	Dr. ir. A. Iosup,	Delft University of Technology
	Dr. ir. A.J.H. Hidders,	Delft University of Technology
	Ir. M. Capotă	Delft University of Technology

TU Delft Delft University of Technology

ABSTRACT

In the age of information our society generates data at an increasing and already alarming rate. To keep up with the rapid increase in the amount of available data, the academics have shown strong interests in the emerging research field of Big Data Processing (BDP), which explores technologies aiming at efficient processing of enormous amounts of data. As a result, many different types of BDP systems, for example systems specialized in large-scale graph processing, have been added into the Big Data ecosystem in the past decade.

The performance of BDP systems impacts both efficiency and direct operational costs significantly, and may be the primary driving force behind the development of these systems. Performance evaluation aims at providing better understanding of BDP systems, which is proven to be challenging. The coarse-grained "black-box" evaluation is commonly used by performance analysts, which considers a BDP system as a black-box and only observes the overall execution time taken by the jobs of the BDP systems. Such coarse-grained evaluation method fails to provide much insight of the internal operations of the BDP systems. In contrast, fine-grained performance evaluation can be challenging and time-consuming, as it requires in-depth knowledge of the BDP systems on the implementation level, and can involve much repetitive work to process large amounts of performance statistics. Moreover, it is under question how efficiently the users can make use of the results of performance studies, as many studies are often out-dated, have a limited scope, and are difficult to reproduce.

This thesis work consists of three main chapters. First, we propose *Granula*, a fine-grained performance evaluation framework which facilitates performance modeling, performance archiving and performance visualizing of Big Data Processing jobs. This framework is designed to extract fine-grained, comprehensive performance results from BDP systems and to facilitate performance-related knowledge exchanges between system developers, researchers, and end users. Second, using the Granula framework, we build comprehensive performance models for two graph processing systems i.e., Giraph and GraphX. The evaluation method used for the purpose of building these performance models can be reused by others for their custom workload, and these models can be extended by other performance analysts for more in-depth studies. Third, we conduct fine-grained performance evaluation on these two graph processing systems and validate the performance models we built. Among the important achievement of this study, we investigate and quantify the performance overhead and the performance bottleneck of these systems. Overall, we hope to have provided better understanding of the performance of BDP systems by improving on the process of performance evaluation, and by defining a method to integrate performance evaluation into the Big Data ecosystem.

PREFACE

I have always known that I want to study Computer Science since I was young, as the astonishing capability to process astronomical amounts of data always seems mysterious to me. For my master thesis, I have chosen to specialize in the field of Big Data Processing. In the past year, I have received incredible support from a great number of individuals during the process of completing my master thesis.

First of all, I would like to express my deep gratitude to dr. Alexandru Iosup for the excellent guidance during my thesis. Alexandru has always managed to motivate me to strive for better results, has placed great trust in the direction of my research work and has provided great support in helping me finish my thesis. I am also very grateful to have worked closely with ir. Mihai Capotă, my daily supervisor, whose guidance throughout the project I sincerely appreciate. While this project has gone through many iterations, it was Mihai who encouraged me to present the initial ideas to Hassan Chafi from the Oracle Lab, whose feedback motivated me to further develop this project to its current form. Furthermore, I want to thank the other members of the committee, prof. Dick Epema and dr. Jan Hidders, for their interest in my research project, and for their efforts in making the defense schedule possible.

It has been a great pleasure for me to work in the Parallel and Distributed Systems group, and surrounded by so many highly skilled professionals and researchers. I have enjoyed much the time working in the same office with Siqi and Yong in the past year. Special thanks to Sietse Au, who I have known for many years and have learned most of my software engineering skills from. Besides working, I also want to thank Otto, Bogdan, Ernst, Jianbin, Tim, Jie, Lipu, Jaap and Niels for arranging enjoyable social events almost every week. I would like to take the opportunity to thank my friends who I have known during my master, Joseph, Marian, Volker, Hans, Joey, Soran and many others, for all the enjoyable time we shared and for the inspiring discussions in group projects. Finally, I would like to thank my parents for their love and continued support.

Wing Lung Ngai
Delft, June 2015

CONTENTS

1 Introduction **1**
- 1.1 Big Data Processing systems . . . 1
- 1.2 Performance Evaluation of BDP systems . . . 2
- 1.3 Problem Statement . . . 3
- 1.4 Research Questions and Approach . . . 5
- 1.5 Main Contributions . . . 5
- 1.6 Thesis Outline . . . 6

2 Background **7**
- 2.1 Big Data Processing . . . 7
- 2.2 Large-scale Graph Processing . . . 9
- 2.3 Performance Evaluation of Large-scale graph processing . . . 10

3 Granula: a Performance Evaluation Framework for Big Data Processing Systems **13**
- 3.1 Overview . . . 13
- 3.2 Performance Modeling Module . . . 15
  - 3.2.1 The Language . . . 15
  - 3.2.2 The Method . . . 16
  - 3.2.3 The Outcome . . . 17
- 3.3 Performance Archiving Module . . . 18
  - 3.3.1 The Method . . . 18
  - 3.3.2 The Outcome . . . 20
- 3.4 Performance Visualizing Module . . . 21
  - 3.4.1 The Method . . . 21
  - 3.4.2 The Outcome . . . 23
- 3.5 Discussion . . . 23
  - 3.5.1 Requirement Fulfillment . . . 24
  - 3.5.2 Application Domain . . . 24
  - 3.5.3 Threats to Validity . . . 24
- 3.6 Summary . . . 25

4 Performance Modeling of Large-scale Graph Processing Systems **27**
- 4.1 Overview . . . 27
  - 4.1.1 Selection of Graph Processing Systems . . . 27
  - 4.1.2 Modeling Abstraction . . . 28
- 4.2 Giraph Model . . . 29
  - 4.2.1 Programming Model of Giraph . . . 29
  - 4.2.2 Operational Model of Giraph . . . 30
  - 4.2.3 Performance Model of Giraph . . . 31
- 4.3 GraphX Model . . . 31
  - 4.3.1 Programming Model of GraphX . . . 31
  - 4.3.2 Operational Model of GraphX . . . 32
  - 4.3.3 Performance Model of GraphX . . . 32
- 4.4 Discussion . . . 33
- 4.5 Summary . . . 34

5 Fine-grained Performance Evaluation of Large-scale Graph Processing Systems **35**
- 5.1 Overview . . . 35
- 5.2 Experimental Setup . . . 36
  - 5.2.1 Real-World Workload . . . 36
  - 5.2.2 Hardware and Software Specification . . . 37

5.2.3 Experiment List . . . 37
5.3 Quantitative Analysis of Graph Workload . . . 39
5.4 Quantitative Analysis of Overall Job Performance . . . 41
5.5 Quantitative Analysis of System Overhead . . . 42
5.6 Quantitative Analysis of Data Load/Offload Operations . . . 45
5.7 Quantitative Analysis of Data Processing Operations . . . 46
5.8 Discussion . . . 48
5.9 Summary . . . 48

6 Conclusion and Future Work **51**
6.1 Conclusion . . . 51
6.2 Future Work . . . 52

A List of Performance Archives **53**

B List of Queries **55**

Bibliography **59**

1

INTRODUCTION

To keep up with the exponential growth in data, both the academics and the industry have shown strong interest in the emerging research field of *Big Data Processing (BDP)*. This has already resulted in the development of various types of BDP systems in the past decade, such as large-scale graph processing systems. Although these BDP systems are currently focusing on better performance than their alternatives, understanding their performance remains a challenging task for many analysts. Therefore, automating the entire process of performance analysis is the goal of this thesis.

1.1. BIG DATA PROCESSING SYSTEMS

From the invention of paper to the new generation of electronic storage devices, technological advances allow the human society to produce and preserve increasingly more information. In early China, knowledgeable scholars could be described as those who possess knowledge equivalent to *five wagons of wooden slips*, the content of which is not more than a modern book of a few hundred pages, and much less than 1% data volume of a regular USB disk. Completing a virtuous circle, the rapid spread of knowledge also stimulates the advances in technology. In the past decade, the exponential growth of available data lead to the phenomenon of Big Data, which is characterized by the enormous data volume, data variety and data velocity, and by data-driven advances in all aspects of human life, of engineering, and of science.

Obtaining valuable knowledge from these massive amounts of data introduces new challenges, as this scale of data processing is beyond the maximum capacity of a single sequential computer. For decades, the processing power of CPU has doubled every 18 months (Moore's Law), but each year we are getting closer to the theoretical upper-bound of the maximum processing power of a single CPU. Yet, the processing power of CPU is not the only limitation here: to give a simple example, assuming the storage device has a read speed of 500 MBps, still more than 21 days are required to read even 1 petabyte of data. Therefore, the Big Data Processing systems (see Section 2.1.1) developed in the past decade are essentially parallel and distributed systems, which apply data parallelism techniques on a cluster of computation nodes to achieve speedup in data processing.

Many other types of BDP systems and applications appeared in the past decade, each targeting different application domains. Together, these BDP systems constitute the Big Data ecosystem (for more details, see Section 2.1.3). For example, after the release of Apache Hadoop, systems such as Giraph and GraphX have focused on large-scale graph processing, a new and challenging application domain of Big Data Processing. *Apache Hadoop*, an open source implementation of *Google's MapReduce*, was released in 2005. The *MapReduce* (see Section 2.1.2) programming model translates high-level data processing operations into highly parallelizable *Mapping* and *Reducing* operations, such that the workload can be concurrently distributed among multiple computation nodes. While *Hadoop* is the most widely used Big Data Processing system in the past decade, the *MapReduce* paradigm of *Hadoop* did not fulfill as the ultimate solution to the Big Data problem. The programming model of *MapReduce* is not expressive enough for problems in many application domains such as graph processing, and therefore results in enormous performance overhead.

1.2. PERFORMANCE EVALUATION OF BDP SYSTEMS

The primary purpose of developing Big Data technology is to reduce the overall processing time for extracting meaningful results from large amounts of data, as such BDP systems are by definition performance-sensitive. When not carefully designed and implemented, a BDP job can easily be order of magnitude slower than expected execution time. To better understand the actual performance of these systems, performance evaluation (see Section 2.3) are currently conducted by analysts in the form of performance studies.

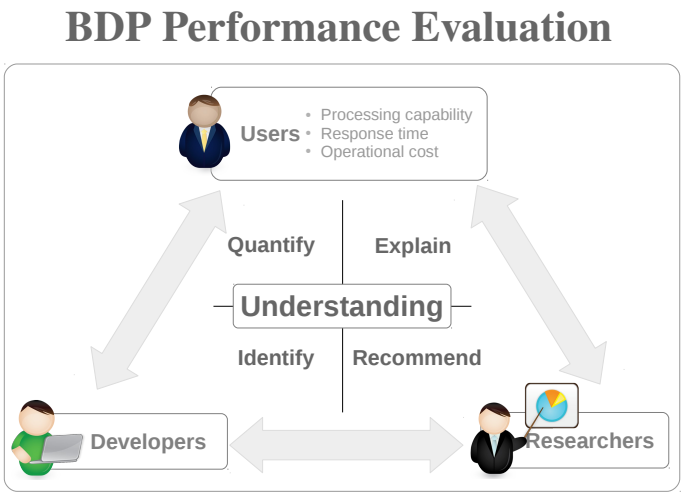


Figure 1.1: Performance evaluation of BDP systems.

MOTIVATION

Performance is a critical successful factor in the field of Big Data Processing. Performance is a broad concept, but it is often expressed in terms of processing capability, response time, and operational costs. A BDP system has better performance when, given the same budget or computational resources, a larger data volume can be processed within a certain time span, or the same amount of data can be processed faster.

Improvements in the processing capability lead to more comprehensive and accurate results; reductions in the processing time lead to higher responsiveness; and less demand on computational resources leads to lower operational costs. All these factors are critical for the competitiveness of a organization. Therefore, having a modern, data-driven comprehensive understanding of performance is crucial in developing and using BDP systems and applications.

GOALS

Performance evaluation is the process of studying BDP systems and deepening the **understanding** of those systems. The results of performance studies are published in different forms, for example blog posts, technical reports, conference articles, and journal articles. These performance studies fulfill purposes such as:

1. **Quantifying system performance** - numerous BDP systems are available in the Big Data ecosystem, which can be different in programming paradigms, software architecture and system implementation. Furthermore, in a BDP system, various types of optimization techniques can be used, or more efficient algorithms can be re-implemented. By conducting performance evaluation, analysts quantify how the performance of one BDP system differs from others on various workloads and resource configurations.

2. **Explaining performance differences** - why does one system perform better than the others? Which set of operations contributes to the overall execution time of a BDP job? Why is the effects of certain optimization technique not observable? Performance evaluation should provide deeper insight into the internal operations of BDP system and be able to explain the difference in performance.

3. **Identifying overheads and bottlenecks** - ideally, the performance of BDP systems should scale linearly with the available computation resources. However, in the reality, various system overheads and performance bottlenecks can occur which prevent optimal scalability of BDP systems. Performance evaluation should be able to pinpoint which internal operations are responsible for performance issues and assess the impact of such issues quantitatively.
4. **Recommending further improvements** - performance evaluation is only meaningful if the results of such evaluation can be effectively communicated to other performance analysts. With the insights gained from the performance study, developers should be able to identify potential improvements on the BDP system, and users should be able to choose the most appreciate BDP systems for their specific application.

STAKEHOLDERS

Performance analysts are the stakeholders who show interest in the performance of BDP systems, and are involved in the performance evaluation of BDP systems. This work identifies three important types of performance analysts:

1. **Developers** are those who design and implement BDP systems. They have the most in-depth system knowledge, and therefore they have the capability to assess the validity of performance results, to explain in detail the design and the implementation of the system, and to hypothesize about abnormal performance. The developers are interested in enhancing the performance of their systems, and in proving that their BDP systems outperform others.
2. **Researchers** are those who carry out extensive performance studies of one or more BDP systems. They have broader knowledge about various types of BDP systems, are relatively less biased than developers, and are often the only stakeholder who can afford to spend the time and energy required by comprehensive performance evaluations. Researchers are interested in facilitating better understanding of the BDP system performance, and sometimes to propose fundamental improvement of BDP systems.
3. **Users** are those for whom BDP systems are built to fulfill their data processing demands. They rarely have sufficient knowledge and time to conduct in-depth performance evaluation. However, only users run BDP systems for practical, real-world applications, and it is important to note that unless the users understand how to fine-tune the BDP system, any performance improvement proposed by developers or researchers can remain theoretical and meaningless. The users are interested in selecting the appropriate BDP systems for their data processing tasks, and in fine-tuning the BDP systems for better performance.

The developers, the researchers, and the users all have different sets of knowledge and interests. Performance evaluation is only meaningful when there is efficient communication among various types of analysts, and that their knowledge is being used effectively.

1.3. PROBLEM STATEMENT

While understanding the performance of BDP systems is essential, conducting a performance evaluation study on BDP systems is challenging. We argue that the current state of performance studies (see Section 2.3) does not completely accomplish their goals listed in Section 1.2, and we observe a few common problems faced by performance analysts, such as the shortcomings of coarse-grained evaluation, difficulties in fine-grained performance evaluation, and the limited applicability of performance studies.

SHORTCOMINGS IN COARSE-GRAINED EVALUATION

In many performance studies, BDP systems are often considered as a "black-box". By varying the input parameters (e.g., data size, type of algorithms) and by observing how the output (e.g., the overall execution time) changes accordingly, analysts derive conclusions on the system performance. For example, by varying the number of computation nodes in a set of experiments, the horizontal scalability of a BDP system can be derived from the difference in overall execution time. Alternatively, by varying the number of computation cores per node, the vertical scalability of that system can be derived in a similar way.

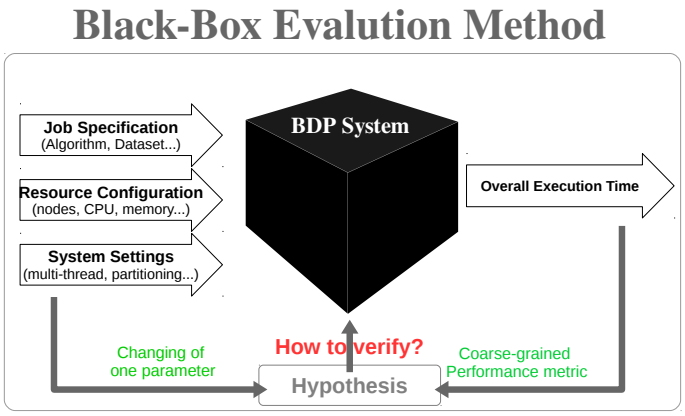


Figure 1.2: Black-box evaluation method.

Although the performance of a BDP system is reflected in its overall execution time, such coarse-grained evaluation method neglects the internal architecture of BDP systems, and therefore fails to explain **why** certain systems perform better than the others. For example, the overall execution time of a job is a commonly-used performance metric, which is in fact affected by the runtime of many hidden operations in a system. Deriving other performance metrics from the overall execution time can be highly inaccurate and biased, as many operations may not be contextually relevant and their performance can change significantly when the BDP system is deployed in a new computation environment. Lacking the means to examine the internal operations of a BDP system, analysts can only hypothesize which operations are truly responsible for the difference in performance.

DIFFICULTIES IN FINE-GRAINED EVALUATION

To gain a comprehensive understanding of the BDP system performance, fine-grained performance evaluation is necessary for analyzing and quantifying the performance of the internal operations in BDP jobs. However, the problem of conducting fine-grained performance evaluation is that such evaluations require deeper knowledge of the systems, costs more work to set up the experimental environment, and produces much more data to be understood.

In-depth evaluation on BDP system requires knowledge not only at the theoretical level (i.e., knowing the programming paradigm), but also at the implementation level (i.e., knowing the source code). Studying the source code of a complex software system is extremely time-consuming, and that applies to each BDP system added to the evaluation.

Furthermore, collecting, structuring, and analyzing performance statistics in a dynamic, distributed environment is tedious and time-consuming. As resource allocation is often dynamic, the same task may not be always allocated to the same worker when repeating an experiment; this means that jobs are rarely truly identical. Instead of simply running jobs on BDP systems, the evaluation process needs to be refined for each BDP system, as the internal software architecture is different across BDP systems.

LIMITED APPLICABILITY OF PERFORMANCE STUDIES

A performance study is meaningful when the results can be effectively communicated to other stakeholders. For example, if the developers are persuaded to re-implement certain performance critical modules, or if the users are convinced to migrate to another (better performing) BDP system. However, performance studies are often out-dated, as the Big Data ecosystem is rapidly evolving with new systems and updates. In addition, each performance study has a limited scope, with only a finite set of systems and workloads tested effectively. The results of these performance studies are difficult to reuse, as users cannot confirm that the conclusions drawn from such performance study also apply in their specific application.

Many performance studies only report briefly their experimental setup and results. With such level of detail, the performance study is difficult to reproduce, therefore the provenance problem arises. Unable to verify

the results under inefficient communication, developers and users cannot fully benefit from the results of performance evaluation.

1.4. RESEARCH QUESTIONS AND APPROACH

In this thesis work, we aim at improving the efficiency and effectiveness of performance evaluation on BDP systems. The following three main research questions are formulated.

RQ1: How to facilitate comprehensive performance evaluation of BDP systems? (addressed in Chapter 3)
While many challenges of performance evaluation are addressed, these challenges all direct to the problem of inefficiency, which is ultimately caused by the lack of reusability. Performance evaluation on BDP system should be a continuously evolving process that integrates into the Big Data ecosystem and results in continuously improving systems. We propose an incremental evaluation method which emphasizes on improving efficiency by substantial reusability of previous work. To support analysts in applying this method, we develop *Granula*, a performance evaluation framework which facilitates performance modeling, archiving and visualizing.

RQ2: How to understand the job performance of large-scale graph processing systems? (addressed in Chapter 4)
By using the Granula framework, we build job performance models for two large-scale graph processing systems, Giraph and GraphX. Each performance model recursively defines a graph-processing job, by modeling explicitly and increasingly more fine-grained internal operations. Each model explains how the programming model is implemented in each graph processing system, and is useful to derive and also to explain fine-grained job performance metrics from the internal operations. The evaluation steps used to develop the performance model are implemented programmatically and is reproducible by any other analysts running their customized jobs.

RQ3: How well do real-world large-scale graph processing systems perform, and Why? (addressed in Chapter 5)
Based on the performance models we built to answer RQ2, we conduct fine-grained performance evaluation on two graph processing systems, Giraph and GraphX. We evaluate these systems by running real-world workloads on these systems, collecting fine-grained job performance metrics and performing statistical analysis on the collected metrics. Using metrics defined in each performance model, we assess the performance characteristics of the graph workload. Furthermore, by decomposing a graph-processing job into data operation, processing operation, and system overhead, we analyze the performance of the internal operations of graph-processing jobs.

1.5. MAIN CONTRIBUTIONS

The main contributions of this work can be summarized as follows:

1. We propose an incremental performance evaluation method for Big Data Processing system which focuses on improving the efficiency and effectiveness of performance evaluation, and in particular makes results reusable. This method defines the complete workflow of performance evaluation: from the logging and the gathering of performance data, to the modeling and the analysis of performance statistics, and finally to the archiving and the visualization of experimental results. This work defines a comprehensive workflow of performance evaluation which, as we show in this work, can be applied to more than one BDP system.

2. We design *Granula*, a performance evaluation framework which facilitates performance modeling, archiving, and visualizing to support analysts in applying the incremental performance evaluation method step-by-step. The Granula framework is designed to be largely reusable by other analysts, for other BDP systems. By doing this, we hope to make performance evaluation of BDP system a continuously evolving process of the Big Data ecosystem.

3. Using *Granula*, We build two comprehensive performance models for large-scale graph-processing systems, Giraph and GraphX. As part of the contribution, we propose fine-grained performance metrics

for these systems. The evaluation steps used to build the performance models are implemented programmatically, which is reproducible by any other analysts running their customized jobs.

4. We implement the Granula framework and use this framework to conduct fine-grained performance evaluation on two graph-processing systems, Giraph and GraphX, by running real-world workloads on these systems. Using metrics defined in each performance model, we assess the performance characteristics of the graph workload. Furthermore, by decomposing a graph-processing job into data operations, processing operations, and system overhead, we analyze the performance of the internal operations of graph-processing jobs.

5. The *Granula* project is currently a central point in the collaboration with several other research projects. First, *Granula* is currently being integrated into the *Graphalytics* project, a benchmarking suite for graph-processing platforms developed by the PDS group of TU Delft. Instead of reporting only the job execution time, *Graphalytics* aims at producing more comprehensive benchmark results in the form of *Granula* performance archives. Furthermore, *Granula* is also being used for the *Lighthouse* project developed in VU University Amsterdam, which uses high-level Cypher queries to express graph pattern algorithms, and converts automatically such queries into optimized Giraph jobs. The fine-grained performance metrics in the Giraph performance model can provide a more accurate overview for the effects of their optimization. Finally, there is an ongoing discussion with the development team of *Parallel Graph Analytics* (PGX), a graph-processing system developed by Oracle Labs, San Francisco, USA, to use Granula for fine-grained performance modeling and reporting in their graph analytics framework.

6. The preliminary design of Granula was included in **A. Iosup, A. L. Varbanescu, M. Capotă, T. Hegeman, Y. Guo, W. L. Ngai, and M. Verstraaten, "Towards Benchmarking IaaS and PaaS Clouds for Graph Analytics", in Workshop on Big Data Benchmarking (WBDB), 2014**. For the complete results that derive from the completion of this thesis, we are currently working on the submission of a journal article in *Concurrency and Computation: Practice and Experience* (CCPE), for the design and implementation of the Granula framework, and for the fine-grained performance evaluation of larg-scale graph processing systems.

1.6. THESIS OUTLINE

The remainder of this thesis is structured as follows: Chapter 2 covers the background information related to Big Data Processing, large-scale graph processing and performance evaluation on BDP systems. Chapter 3 describes *Granula*, a fine-grained performance evaluation framework for BDP systems. Chapter 4 describes the performance models of two large-scale graph processing systems, Giraph and GraphX. Chapter 5 describes the fine-grained performance study conducted on Giraph and GraphX using the real-world environment provided by the DAS Dutch multi-cluster system. Finally, Chapter 6 summaries the conclusions of this thesis work and the future work.

2

BACKGROUND

In this chapter, we describe more elaborately the background information and the related work of this thesis work. We first introduce the basic concepts of Big Data Processing (BDP). Then we go deeper into the application domain of large-scale graph processing. Finally, we summarize recent work on performance evaluation of large-scale graph processing systems.

2.1. BIG DATA PROCESSING

Big Data Processing is an emerging research topic in distributed computing systems, which aims to process data that are too large and complex to be handled efficiently by traditional database systems. We introduce the concepts of Big Data, the Big Data programming paradigms, the multi-layered architecture of BDP systems and the Big Data ecosystem.

2.1.1. BIG DATA

Advances in technology allow our society to generate and to preserve data at an increasing rate. For example, information sensing devices such as mobile phone and cameras are getting cheaper and largely available to the general public. Development in web technology and social networks allow users to publish large amounts of data online. Storage devices are getting cheaper which allows more data to be permanently stored. The phenomenon of Big Data is commonly characterized by the volume, variety and velocity. Volume as in the enormous amounts of data that are available; variety as in the richness of data sources and differences in data structure; velocity as in the fast speed at which data are being generated.

Living in the age of knowledge, those who can extract knowledge from this enormous amounts of data may gain a huge advantage over others, and therefore many modern organizations are becoming data-driven in their decision-making process. However, processing so much data is extremely time-consuming, and the traditional database systems have difficulties in handling Big Data. Conventionally, a company may choose to buy more expensive hardware to increase the processing speed. This approach may not be cost-efficient, and eventually the theoretical upper bound of the processing power of a serial computer will be reached. Therefore, parallelization of the data processing operation is inevitable. By applying data parallelism techniques, BDP systems divide the data processing workloads into multiple computation nodes, such that the data can be processed concurrently.

2.1.2. PROGRAMMING PARADIGMS

Programming in a distributed environment is difficult for many users, because it requires a lot of domain-specific expertise. Programming paradigms are needed to create an abstraction of a Big Data Processing system running in a distributed environment, such that users can express their high-level application code without directly facing the complexity of the systems.

GENERAL-PURPOSE DATA PROCESSING

The MapReduce [1] paradigm (see Figure 2.1) was pioneered by Google, and popularized by the open source BDP system Apache Hadoop [2]. This paradigm makes use of functional programming concepts such as *Map* and *Reduce* to parallelize data processing operations. First, the input data are divided into multiple Mapper, which converts unstructured data into key-value pairs. Then the intermediate data are sorted by keys, sent over the network, and merged together at the reducers. Finally the merged data are processed by the reducer to produce the output data.

MapReduce is applicable for data processing in many areas, and by means of parallelization, MapReduce can achieve significant speedup compared to sequential data processing. However, this programming model is not always expressive enough for different application domains, and may introduce significant performance overhead.

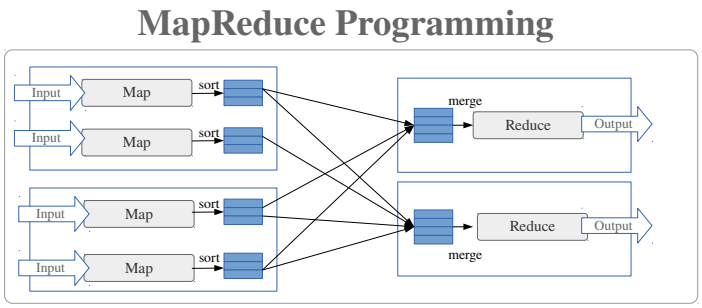


Figure 2.1: MapReduce programming paradigm.

LARGE-SCALE GRAPH PROCESSING

Large-scale graph processing is a specific domain within Big Data Processing. Graph processing is applicable in many different business/research areas such as social network analysis, logistics and bioinformatics. Although graph algorithms can be implemented in MapReduce, such implementations are usually highly inefficient. Therefore specific programming models such as Bulk Synchronous Processing (BSP) are developed for large-scale graph processing.

2.1.3. MULTI-LAYERED ARCHITECTURE AND ECOSYSTEM

Processing data in a distributed environment is challenging, especially for those who are not specialized in distributed computing. BDP systems are organized in a multi-layered architecture (see Figure 2.2), which can effectively decouple responsibility of each layer. The decoupled responsibility ensures that each BDP system only needs to focus on specific challenges, and new system can therefore be developed at much faster speed. Large variety of BDP systems have been released in the past decade, which constitute the Big Data ecosystem.

High level languages such as Pig, Hive and Impala provide a user-friendly interface that are similar to the SQL language used in traditional database systems. The application codes developed by users can be automatically converted into the language used in the programming models. Programming models such as MapReduce and BSP provide an abstraction of the complex operations of the BDP systems, which can be implemented or optimized differently in execution engines. Execution engines enable efficient data processing by applying data parallelism techniques in a distributed environment. Finally storage engines such as HDFS and S3 facilitate efficient and scalable data loading and offloading operations, which is essential in data-intensive operations.

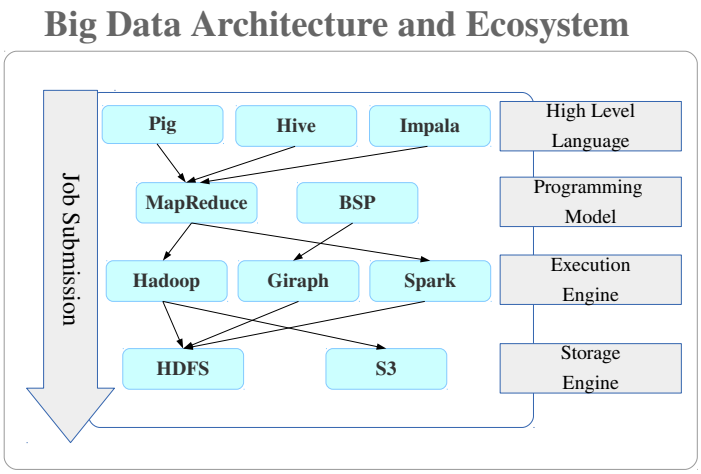


Figure 2.2: Big Data architecture and ecosystem.

2.2. LARGE-SCALE GRAPH PROCESSING

Large-scale graphs are applicable in many application domains such as social network analysis and bioinformatics. However processing large-scale graphs in distributed mode is challenging, consequentially many BDP systems are specialized in large-scale graph processing.

2.2.1. GRAPH THEORY

A graph is a mathematical structure that models entities in terms of vertices, and relationships in terms of edges. This form of data structure can naturally express problems in many different domains. However, as any vertex can be arbitrarily connected to any other vertices in the graph, graph data partitioning across a distributed environment becomes challenging, and communication may become a significant system overhead.

2.2.2. PARADIGMS OF GRAPH PROCESSING

There are many programming paradigms developed for graph processing. Here we discuss sequential graph processing, general-purpose parallel processing and graph-specific parallel processing.

- **Sequential graph processing** paradigms are very efficient in processing small amounts of data, as there are no parallelization overhead involved. Examples are graph databases such as Neo4j [3], or sequential graph processing engines such as GraphChi [4]. However these systems are not scalable, as the execution time can be severely extended if the amount of data increases beyond the processing capacity of a sequential computer.

- **General-purpose parallel processing** paradigms in BDP systems such as Hadoop [2] and Spark [5] also support implementation of graph algorithms. However general-purpose paradigms may not be expressive enough, which result in unreadable code. Furthermore, graph algorithms implemented in general-purpose data processing framework can be highly inefficient in terms of performance.

- **Graph-specific parallel processing** paradigms such as Bulk Synchronous Processing (BSP) [6] model proceeds in a series of synchronized global supersteps (Figure 2.3). In each superstep, all (active) vertices in the graph take messages as input, execute the computation on those received messages, and then send messages to some other vertices. When all vertices finish a superstep, they can advance the synchronization barrier and start with the next superstep. BSP paradigm is very efficient as graph data are usually kept in memory.

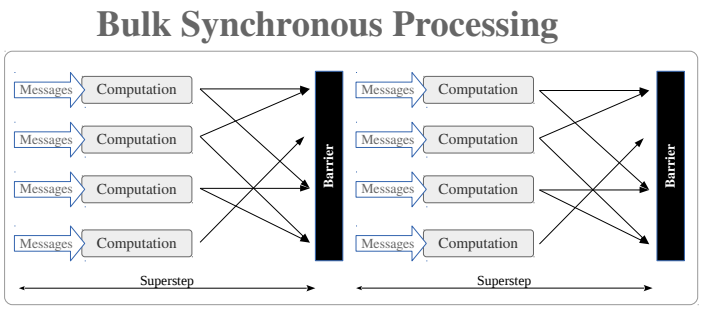


Figure 2.3: Bulk synchronous processing.

2.2.3. GRAPH PROCESSING SYSTEMS

There are many graph processing systems available. An extensive survey can be found in [7], which presents a taxonomy of over 80 graph processing systems, notably Apache Giraph [8], GraphX library [9] of Apache Spark, and GraphLab [10]. The variety of graph processing systems prevents users from having a comprehensive understanding of which graph processing systems are better performing and more suitable for their use scenarios. Therefore performance evaluation studies are conducted for the purpose of understanding their performance.

2.3. PERFORMANCE EVALUATION OF LARGE-SCALE GRAPH PROCESSING

Many performance studies have been published in the domain of large-scale graph processing. Although these studies often limited themselves in only evaluating one or two systems, several performance studies [11–15] have evaluated the performance of a broad set of graph processing systems (4-6 systems). These performance evaluation studies are similar in terms of research methods. By running a set of graph datasets and graph algorithms on the selected graph processing systems, analysts derive performance metrics from the experimental results for assessing the performance of these systems. Here we compare our performance evaluation method of BDP systems (see Chapter 3) with the evaluation methods used in these performance studies.

	Elser et al. [11]	**Guo et al.[12]**	**Han et al. [13]**	**Lu et al. [14]**	**Satish et al.[15]**
Systems	5	6	4	5	6
Algorithms	1	5	4	7	4
Datasets	7	7	5	6	8

Table 2.1: Number of graph processing systems, graph algorithms, and graph datasets per performance study.

Coarse-grained evaluation method is often used in these performance studies, in which graph processing systems are considered as a black-box, the difference in the internal architecture is completely neglected, and only coarse-grained performance metrics such as *overall execution time* is used to assess the job performance. Occasionally, these studies make use of performance metrics such as *setup time* and *computation time* [13], and *time per iteration* [15]. These metrics are not explicitly defined for each graph processing system, as there is a lack of comprehensive understanding of how these graph processing systems operate internally. In contrast, our evaluation method allows analysts to define a graph processing job recursively by its underlying operations in a performance model, which can be used to described the complex internal architecture of graph processing systems.

Performance evaluation of large-scale graph processing systems is a knowledge-intensive and labor-intensive process. The evaluation method used in these studies are difficult to reproduce, as many implementation details are missing in the description. Although a few studies also make their experiment scripts publicly available [12, 13], setting up the experiment environment is still challenging, and such scripts only support a limited set of predefined graph algorithms and graph datasets (see Table 2.1). In contrast, our evaluation

method are explicitly defined and can be completely integrated into the systems, such that users can reuse the same method to assess the performance of their own graph processing jobs.

These performance studies have produced many experiment results. However, due to space limitation, only the most interesting results are presented to the readers. Many intermediate results are not accessible. In addition, visualization techniques such as bar charts and line charts do not preserve the information with high accuracy. Although a few performance studies also publish their technical reports [12, 13], readers have to navigate through large amounts of experimental results presented in tens or hundreds of pages to find the relevant information. In contrast, our evaluation method preserves all intermediate evaluation results, describes explicitly how the final evaluation results are derived from their source information, and therefore allows other analysts to thoroughly examine the entire evaluation process.

3

GRANULA: A PERFORMANCE EVALUATION FRAMEWORK FOR BIG DATA PROCESSING SYSTEMS

Performance is a critical success factor for Big Data Processing (BDP) systems. However, in-depth performance evaluation of BDP systems is challenging for many reasons. In this chapter, we purpose *Granula*, a fine-grained performance evaluation framework which supports incremental, comprehensive performance evaluation on BDP systems.

3.1. OVERVIEW

Performance is critical for the success of a BDP system, and performance evaluation is the way to obtain a better understanding of the system performance. However, the comprehensive performance evaluation of BDP systems still faces many challenges (Section 1.3): a coarse-grained approach does not provide sufficient insights of the system performance; a fine-grained approach is too time-consuming and requires repetitive work; and the results of empirical performance studies are difficult to apply for practical scenarios.

We believe that the efficiency and effectiveness of performance evaluation can be significantly improved by redefining the evaluation process. We realize that performance evaluation should be a continuously evolving process that constantly adapts to the new developments in the Big Data ecosystem. However, evolution does not usually include leaps and bounds: improvement occurs gradually over time. Therefore in this work, we propose an incremental performance evaluation method for BDP systems. As a matter of course, in any incremental method it is required that the previous work must be substantially reusable, and this reusability is in fact the key to efficiency. Therefore, in this framework, we aim to facilitate:

- **The reuse of code:** we aim at liberating performance analysts from repetitive technical implementations that are required during performance evaluation.
- **The reuse of data:** we improve the quality of the published performance evaluation results, such that the data is self-contained, self-explanatory and self-proving.
- **The reuse of methods:** we apply an evaluation method that allows performance analysts to deepen their exploration systematically, without getting overwhelmed by information. This method allows both deep exploration on one BDP system, and broad exploration on various BDP systems.
- **The reuse of knowledge:** we encourage collaboration among performance analysts by facilitating an efficient sharing process, and by allowing performance analysts with different expertise to participate in the evaluation process.

To support the reusability demanded by this incremental evaluation method, we develop *Granula*, a fine-grained performance evaluation framework consisting of three main modules (Figure 3.1): the modeler, the

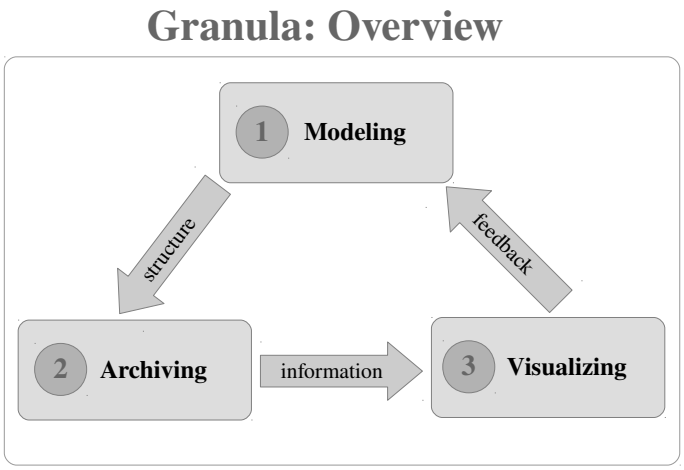


Figure 3.1: Overview of the Granula framework.

archiver and the visualizer, which map respectively to the process of performance modeling, performance archiving and performance visualizing.

- **Performance Modeling (see Section 3.2):** Performance evaluation on BDP systems is a time-consuming process: the performance analysts need to study both the high-level design and the low-level implementation of a BDP system, run jobs on the BDP system to extract performance information, analyze the gathered performance information, derive new insights from the performance results and finally elaborate the conclusion by means of text and diagrams. The knowledge on how to conduct performance evaluation requires much domain-specific expertise, which is worth preserving. However, the evaluation process, or better to say, the evaluation method, is usually described in only a few paragraphs, in an academic publication, or less formally, in a web page or a blog post. Such description leaves out lots of details which makes the evaluation process difficult to reproduce. The *Granula* modeler allows performance analysts to define their evaluation method for a BDP system explicitly, such that other performance analysts can follow each step in the evaluation process and examine the validity of the performance model.

- **Performance Archiving (see Section 3.3):** To take a step further, the performance model created by the *Granula* modeler is for the evaluation of each BDP job, a performance archive can be built programmatically with the *Granula* archiver, which serves as a snapshot of a BDP job that encapsulates the complete set of performance information in a structural way. The performance archive is self-contained (that is, without asking where the results can be found), self-explanatory (that is, without asking what the results stand for) and self-proving (i.e., without asking how the results are derived).

- **Performance Visualizing (see Section 3.4):** while a performance archive is sufficiently informative, it is not the most natural way of browsing through the performance results. The *Granula* visualizer presents the performance archive in a human-readable manner, allows efficient navigation of performance results, such that those results can be easily communicated among performance analysts.

In each iteration of the performance evaluation (see Figure 3.1), performance analysts first study the BDP system and update the performance model with the newly discovered insights. Then they proceed by running jobs on the BDP system and creating performance archives for those BDP jobs. By reviewing the performance results via the *Granula* visualizer, they gain new insights of the system which can be used to refine the performance model. After iterations, knowledge on the BDP system performance is built up in an incremental manner.

By using Granula, we are able to build comprehensive performance model for two large-scale graph processing systems i.e., Giraph and GraphX (as described in Section 4.2 and Section 4.3). And in the following three sections, we will use Giraph as an running example to elaborate on the process of performance modeling,

archiving and visualizing.

3.2. PERFORMANCE MODELING MODULE

Distributed systems such as BDP systems have a higher level of complexity than many other types of software systems. Being able to conduct fine-grained evaluation implies that the performance analysts have an in-depth understanding of the BDP system. In addition, performance evaluation on BDP systems is a time-consuming process which includes :

1. studying both the high-level design and the low-level implementation of a BDP system.
2. running jobs on the BDP system to extract performance information.
3. analyzing the gathered performance statistics to derive new insights.
4. and finally explaining the conclusion by means of text and diagrams.

The knowledge on the BDP system itself, and the knowledge on how to conduct performance evaluation on BDP systems require much domain-specific expertise, which is worth preserving. The Granula modeler allows performance analysts to build a comprehensive performance model for a BDP system, and more importantly, the Granula modeler allows analysts to explicitly define the evaluation method used for building such model, so that other analysts can examine, repeat, or even extend the evaluation process.

3.2.1. THE LANGUAGE

The *Granula* modeler makes use of three modeling concepts i.e., *Entities*, *Attributes*, and *Relations*. An *entity* represents a concrete or an abstract element in the BDP system i.e., Operation, Actors, Missions. Each *entity* contains a list of *attributes* that describe its properties or characteristics. A *relation* defines the relationship between two *entities*. In this chapter, we use Giraph as an example to explain how a Giraph job can be expressed in this modeling language.

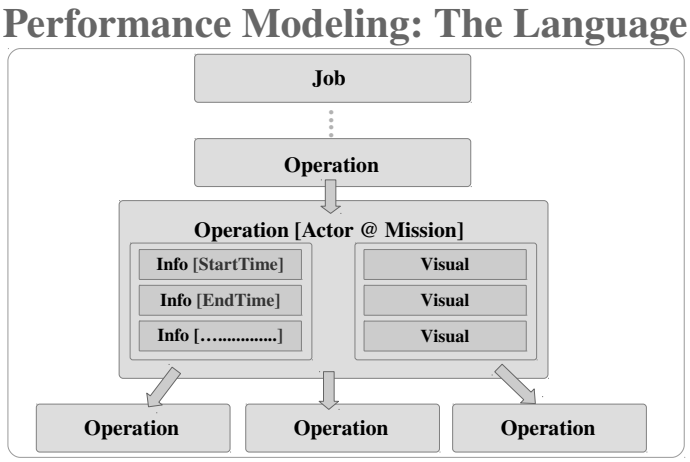


Figure 3.2: An operation-centric performance model.

Entities:

- *Job* is an entity at the top level of the performance model, consisting of multiple internal *operations*. For example, *GiraphJob* is at the top level of the Giraph model. While here the *job* is only regarded as the top-level element, it would be the only observable element of a BDP system when using the black-box approach.
- *Operation* is defined as a *mission* assigned to an *actor*, therefore an *operation* can be simply referred as [actor @ mission]. Each *operation* can be further divided into sub-operations. In this way, an *operation* can be recursively defined by its underlying sub-operations. With Giraph's example, an *operation* can

be [BspMaster @ GlobalSuperstep], which contains a list of [BspWorker @ VertexComputation] as its sub-operations.

- *Actor* is the one who is responsible for the execution of the *operation*. Responsibility is the keyword here, as often multiple *actors* are assigned with similar *missions*. With Giraph's example, assuming that n *BspWorkers* are responsible for the *VertexComputation* at *GlobalSuperstep-M*. In the case that *GlobalSuperstep-M* takes surprisingly long, then it would be interesting to know which *BspWorkers* are responsible for that.
- *Mission* is the task to be accomplished by an *actor* in an *operation*. When a *mission* is unique, and consequentially can only be assigned to an unique *actor*, in practice there is not much difference between a *mission* and its *operation*. In this case, an operation [actor @ mission] is simply referred as [mission].

Attributes:

- *Record* contains the performance information extracted from a Granula log, which is found in the system logs generated by the (modified) BDP system for each job. Empirically, *records* contain the source information from which all other information can be derived.
- *Info* is the information either extracted from a *record*, or derived from other *infos*. *Info* always belongs to an *entity* and is used to describe the *entity*. Such info can also be referred as a *performance metric* when it can be used to describe the performance of the *entity*.
- *Visual* defines how an *entity* can be presented in the Granula visualizer using its *infos*. Here the analysts have the chance to choose the form and the content of the visualization.

Relations:

- Each *operation* can only have one *actor* and one *mission*. On the other hand, an *actor* or a *mission* can be assigned to multiple operations. For example in Giraph, *BspMaster* can carry out m *GlobalSupersteps*, and *VertexComputation* can be done by n *BspWorkers* simultaneously. In such case, an *operation* is identified with [actor-id @ mission-id]. Each *operation* contains a set of *records*, *infos* and *visuals*.
- While the properties of an *operation* are described by its attributes, its context can only be defined by its position in the operation hierarchy. Each operation has one super-operation and a number of sub-operations. The *mission* assigned to each *operation* is meant to be accomplished by its sub-operations, therefore the super-operation describes the higher purpose, and the sub-operations carry out the actual execution. In this way, each *operations* can be recursively defined by its underlying *sub-operations*.

3.2.2. THE METHOD

Building a performance model is in fact the process of a performance analyst trying to explore, analyze and describe a BDP system. Our incremental method consists of the following steps.

- **Log Generation:** the analyst starts by running a job on a BDP system, such as Giraph. By looking into the source code (and possibly into the system logs), the analyst finds out which classes and functions are responsible for the job execution. With this knowledge, the analyst can inject *Granula records* into the logs by modifying the source code of the BDP system.
- **Data Collection:** the analyst collects job logs generated by the BDP system. These logs can be scattered in multiple machines in a cluster that needs to be gathered together and then sorted by job. For each collection of job logs, the analyst extracts the *Granula records* and use those as the source information.
- **Operation Subdivision:** the analyst constructs the top-level operation of the BDP job based on the *Granula records* describing its start time and end time. Adding these *records* is not difficult as the Granula *records* can be simply injected before and after the first line and the last line of the origin logs. In Giraph for example, the main function of the *AppMaster* class has an execution time that encapsulates the lifespan of the entire Giraph job.

  In the next iterations, the analyst can look into the code executed between the custom Granula *records*, and continues with adding sub-operations to the top-level operation. This process can go indefinitely down the hierarchal tree depending on the necessity and the available time.

- **Information Derivation:** for each operation of the BDP job, the analyst decides which pieces of information are valuable, and instructs the BDP system to log these information. The *StartTime* and *EndTime* are the most fundamental *infos* of an operation. In Giraph for example, the analysts may be interested in the *MsgVolume* of each *BspWorker* in a superstep, and such information can be added to the corresponding operation.

  The initial information reported by the system are not sufficient enough for comprehensive evaluation. However, the analyst can derive new *infos* from existing *infos* in the current operation or in its neighboring operations. For example, the *Duration* of an operation can be derived from the *StartTime* and the *EndTime*. Eventually this derivation step can lead to more conclusive performance metrics. For example in Giraph, all *MsgVolumes* of each *BspWorker* can be aggregated in *GlobalSuperstep* operation, which indicates the network traffic in that superstep.

- **Visual Description:** finally, the analyst can decides to describe each operation by defining a set of visuals. These visuals instructs Granula visualizer in what ways the *infos* of an operation should be presented to the viewers. More details on the visualizer can be found in section 3.4.

This incremental evaluation method facilitates a continuously evolving process, which develops as the knowledge about the system increases. It can be applied to many types of BDP systems, and is especially efficient in exploring new and unfamiliar BDP systems. In each iteration, the analyst is able to **examine** the results of previous evaluation and verify the assumptions about the BDP system. For example, when exploring Giraph, the analyst can be convinced that certain functions are responsible for a heavy *DataLoad* operation, while by observation it is proven that the duration of that operation is too small. The analyst then realizes a wrong assumption was made and therefore needs to redefine the performance model based on the feedback. Similarly, the analyst is alerted when certain operations have a negligible runtime, and does not need to waste more time for further investigation. With this incremental method, the evaluation process can transverse down the operation hierarchy with indefinite granularity, depending on the necessity and the available time (hence the project name *Granula*).

3.2.3. THE OUTCOME

Theoretically, the evaluation method described above can be documented in extreme details even without *Granula* (unlikely), and be reproduced by another analyst manually (more unlikely). However, in practice, it is essential that this evaluation method can be reproduced **programmatically**, such that other analysts can examine, repeat, or even extend the evaluation process. The performance modeling process eventually results in two software deliverables: a *Logger* and an *Analyzer*, designed and implemented specifically for each BDP system (more precisely, each supporting major version of the BDP system).

- *Logger* incorporates the *log generation* process, which contains a software patch that encapsulates the modifications of the source code of the BDP system, such that after applying this patch, the BDP system can generate customized *Granula records* via its logging mechanism. Using the existing logging mechanism is the most convenient way to create the patch for the BDP system. However, if *Granula* is well-integrated into the BDP system, then a better way is to develop an internal mechanism specifically for storing, transferring and exporting performance data for *Granula's* use scenario.

- *Analyzer* incorporates the *operation subdivision*, *information derivation* and *visual generation* processes. It contains a set of modeling rules (see Table 3.1) which define explicitly how to interpret the gathered performance logs and how to use those *infos* to evaluate the performance of the BDP system.

Process	Rule	Description
Operation Subdivision	Assembling Rules	Group records belong to the same operation.
Operation Subdivision	Linking Rules	Link operation to its parent.
Information Derivation	Derivation Rules	Derive information from other information.
Visual Description	Visualization Rules	Define a visual with certain input information.

Table 3.1: Modeling rules.

By reusing these two software deliverables, other analysts can reproduce the same evaluation process programmatically with their customized workloads (Figure 3.3). First, the analyst needs to apply the *Logger* patch to the source code of the BDP system. Then the analyst can run a job on the modified BDP system to collect the performance logs. After that, the Granula archiver can use the modeling rules of the *Analyzer* to interpret the performance logs, and outputs a performance archive that contains the evaluation results of the job performance. The last step is in the *performance archiving* process, the details of which are explained more thoroughly in Section 3.3.

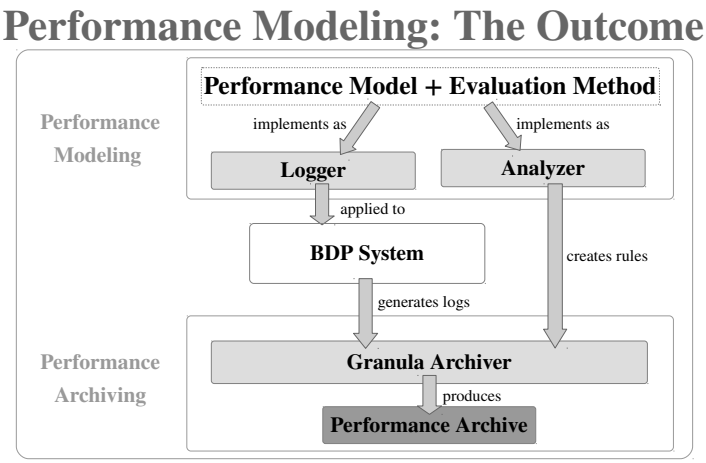


Figure 3.3: The outcome of performance modeling.

3.3. PERFORMANCE ARCHIVING MODULE

The *Granula archiver* aims at building comprehensive performance archives for BDP jobs. A performance archive is contextually similar to an academic publication, as both contain the results of performance evaluation. However, academic publication as a communication method has its restrictions: it is non-interactive (static text and diagrams) and its length is limited (10 - 15 A4 pages). A lot of important intermediate experiment results are missing in academic publications, which causes difficulties for another researchers examining results. In contrasts, the *Granula archive* serves as a snapshot of a BDP job that encapsulates the complete set of performance results in a structural way. The performance archive is self-contained (that is, without asking where the results can be found), self-explanatory (that is, without asking what the results stand for) and self-proving (that is, without asking how the results are derived).

Furthermore, fine-grained performance evaluation is challenging, as such evaluation require much domain-specific knowledge and lots of repetitive work. In Section 3.2, we have already explained at the conceptual level how the knowledge of performance evaluation is preserved by using our incremental evaluation method. Here we describe at the technical level how the entire evaluation process, from collecting the performance logs to building the performance archive, can be fully automated by using the *Granula archiver*. Here we do not use Giraph as an example, as the *Granula archiving* process is completely system-independent.

3.3.1. THE METHOD

The performance archiving consists of three intermediate processes, namely the *filtering*, the *assembling* and the *filling* processes. In each intermediate process, performance data are processed by the modeling rules defined by the analyst during performance modeling.

COLLECTING PROCESS

A BDP system generates logs for each running job, which may be stored in a centralized storage point, or more likely be scattered across multiple computation nodes. The original logs are injected with Granula *records* by

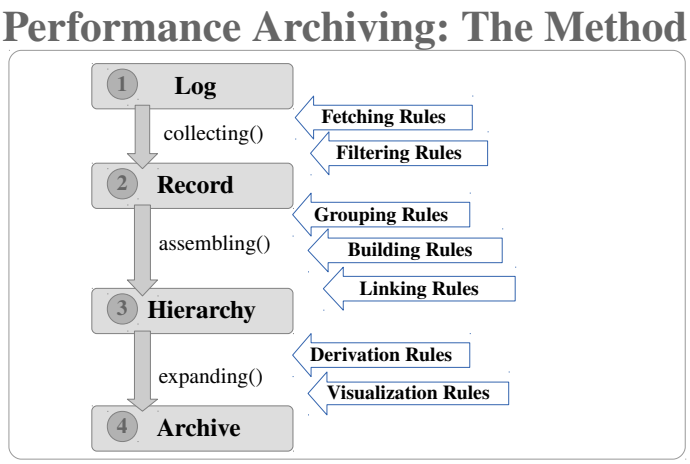


Figure 3.4: The method of performance archiving.

the patched BDP system (see Section 3.2). The *collecting* process gathers *logs* and extract *records* by *fetching rules* and *filtering rules*.

- **Fetching Rules:** defines how the logs are collected and stored, so that the archiver can later access the source information with ease. This process is dependent on the software architecture of the BDP system. The system may have a complete monitoring system that gathers performance data systematically, or each distributed executor may just generates logs independently. A simple fetching rule is to gather the system logs from the distributed environment, group logs by job directories, and save each directory in a compressed file.
- **Filtering Rules:** defines how to extract *records* from gathered *logs*. Usually this process is extremely simple: just add an unique identifier for each *Granula record* to filter out irrelevant logs.

However, if *Granula* is well-integrated into the BDP system, then a better way is to develop an internal mechanism specifically for storing, transferring and exporting *Granula's* record. Then the collecting process is no longer required.

ASSEMBLING PROCESS

As each job contains a collection of records, the assembling process groups the records by operation, establishes connections between operations, and creates an operation *hierarchy*. This process is not always straight-forward, as logs can be generated on different computation nodes without reference to each other. The assembling process makes uses of the grouping rules, the filling rules and the linking rules.

- **Grouping rules:** defines which records belong to the same operation. When an operation is unique, the grouping rule is extremely simple. Or if the combination of actor id and mission id is unique, then the grouping rule is still trivial. In other case, records can still be grouped together by analyzing their timestamps or by other means.
- **Linking rules:** defines how operations should be linked together. Each operation has a super-operation. A linking rule defines how an operation can find its corresponding super-operation from all other operations. The matching can be based on the id, or other specific logics.
- **Filling rules:** defines how to add abstract operations to a BDP job. There can be many low-level operations in a job which are difficult to manage. The filling rule allows high-level abstract operations to be added such that those low-level operations can be grouped into high-level operations.

While the assembling process sounds verbose, it is essential, as the analyst can be easily overwhelmed by thousands of records and hundreds of operations per job. By keeping the hierarchy in mind, the analyst can think in terms of operations and their connected neighbors when analyzing the performance.

FILLING PROCESS

When the operation hierarchy is ready, for each operation, the filling process analyzes records, derives new information, and creates the visuals. The filling process makes uses of derivation rules and visualization rules.

- **Derivation Rules:** defines how infos can be derived from records, or from other infos. Eventually, the goal is to derive meaningful performance metrics for a job after iterations of derivations. The intermediate infos are retained in the archive, as they can be examined later, or be reused for different performance metrics.
- **Visualization Rules:** defines how the infos of an operation can be visualized and presented in a view in the Granula visualizer.

The filling process defines how the analyst evaluates performance of a job using source information provided in the records. It is the last step of performance archiving.

3.3.2. THE OUTCOME

The outcome of the performance archiving process is a Granula performance archive for BDP job. The archive not only describes the overall performance of a job, but also the fine-grained information regarding internal operations in the job. It is self-contained, with both experimental results and intermediate results used during the evaluation method. All results are traceable without the needs for additional data sources.

It is important to note that the performance archive is both human-readable and machine-readable. The archive is stored in XML file format, which can be supported by many third-party software applications. It is machine readable because the intermediate results can be queried by other analysts to see if the whole model is consistent to their expectation. For example we can use the XQuery code in Figure 3.5 to get the results in Table 3.2:

```
xquery version "3.0";
for $archive in collection("/granula/?*.xml")
   for $job in $archive/Workload/Job
      let $dataInputPath := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="DataInputPath"]/string(@value)
      let $computationClass := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="ComputationClass"]/string(@value)
      let $numContainers := $job//Operation[Mission/@type="NumContainers"]/Infos/Info[@name="NumContainers"]/string(@value)

      let $bspTime := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="Duration"]/string(@value)

      return concat($computationClass,' ',$dataInputPath,' ',$numContainers,$bspTime)
```

Figure 3.5: Query to extract Performance Information.

Computation	DataInput	No. of Containers	BspTime
BFS	Citation_FCF	6	43901
CommunityDetection	amazon.302_FCF	6	41896
BFS	Friendster_FCF	6	801078
ConnectedComponent	KGS_0_FCF	6	39909
...	...	...	...

Table 3.2: Query results.

In this way, we can explicitly define which method was used to extract performance information from the archive. Therefore the extraction process is reproducible. By creating a performance archive, communicating

experiment results will be much more effective.

3.4. PERFORMANCE VISUALIZING MODULE

While in previous section it was discussed how a performance archive can be built which contains comprehensive information regarding the internal operations of a BDP job, such archive will only fulfill its purpose when the contents can be effectively understood by other analysts. As discussed in section 1.2, we consider three types of performance analysts:

- **Developers** have the most in-depth knowledge on the design and the implementation of the BDP system. They can assess the completeness of the operational model and verify the correctness of how performance information is extracted from the system.
- **Users** are those who are running the actual real-world jobs. While it is essential that they understand the performance of their running jobs, many users are just beginners in BDP. They need a thorough explanation of the performance model embedded in the performance archive.
- **Researchers** have broad knowledge of BDP systems. They fulfill the role of verifying the evaluation method which defines how performance metrics are derived from raw performance information. They could also be interested in extending the performance model to explore uncovered areas.

These analysts have different demands on the type of knowledge they need from the performance archive. The Granula visualizer is designed to translate the Granula archive into a human-readable format. The presentation of the archived information should be complete, and easy to understand.

3.4.1. THE METHOD

There is usually a trade-off completeness and the understandability are usually. Researchers are well-trained in being concise due to the limited length of academic publications. Conventional communication platforms such as articles, journals or technical reports are limited by their form of presentation. In this work, we refine the communication process and use the web as the communication medium.

NAVIGATION

A performance archive can contain hundreds of operations and thousands of performance information. How can other analysts navigate through all these data without being overloaded by information? As a BDP job is recursively defined by its internal operations, the operations of a BDP job are organized in a hierarchical structure.

A user starts by clicking on a URL which links to a workload archive that contains a list of job archives. By selecting a job, user navigates to the top operation of that job. In each operation, the sub-operations are presented, which can be selected, then the viewer can navigate to the sub-operations. Viewers can go back and forward with a hyperlink chain and navigate using relationship between operations.

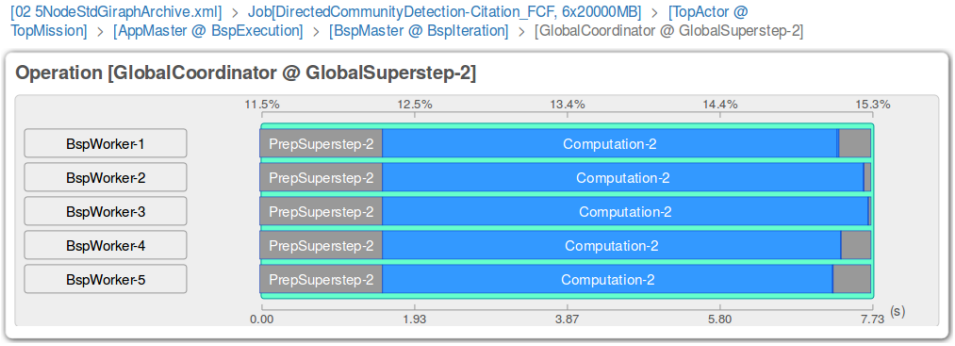


Figure 3.6: Operation Navigation Process

PRESENTATION

Each operation contains a list of *Visuals*, which presents the performance information to the viewers. These visuals are defined by the visualization rules specified during the modeling process. There are three main types of Visuals. Visuals are organized in vertical blocks.

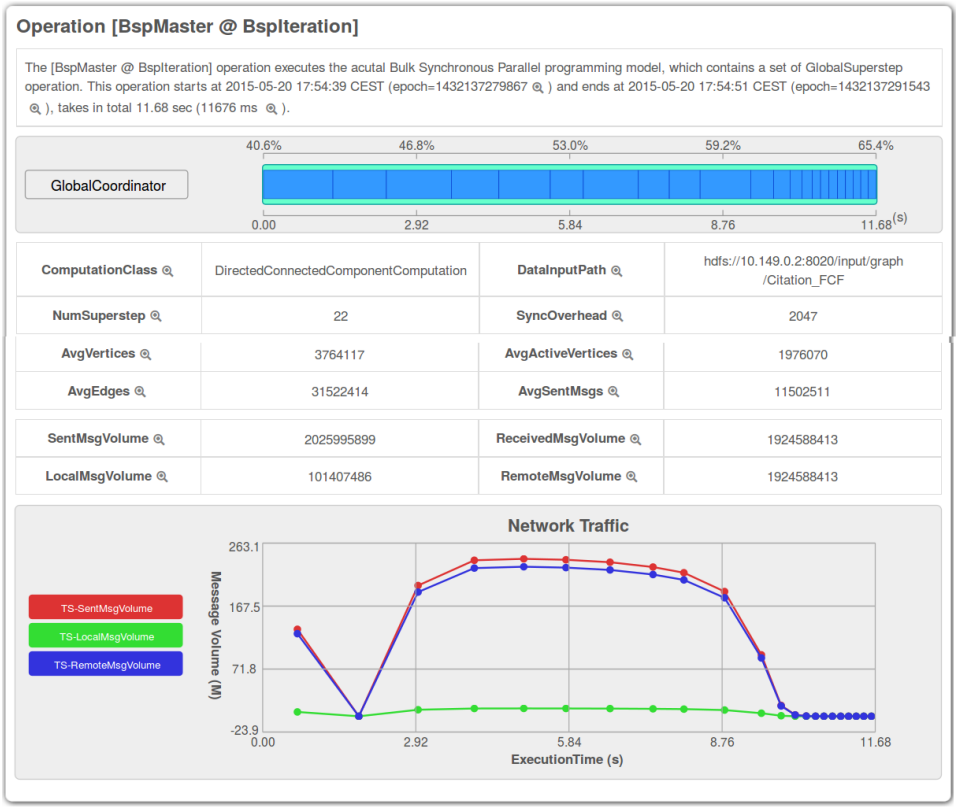


Figure 3.7: Types of Visuals.

- **SummaryVisual** shows the description of the operation.
- **NavigationVisual** shows the duration of an operation, which encapsulates its sub-operations in its domain. The sub-operations are organized by its actor on vertical-axis, and on the horizontal-axis by the duration of its mission.
- **TableVisual** contains a list of key performance information. Each information is identified with its name and it's value.
- **ChartVisual** visualizes a set of related information in a chart. During the modeling process, only a few parameters and the input information UUID needs to be specified.

PROVENANCE

In an empirical study, each piece of information should be derived from one or multiple sources. Yet the derivation process is often ambiguous, which causes problem in tracing provenance of the conclusions. However, in the Granula visualizer, performance information can be traced back to its sources.

For each piece of performance information in each operation , there is an associated *TraceButton* (examples can be seen in Figure 3.8). By clicking on the *TraceButton*, the viewer can investigate how the performance metric *BspRatio* is derived. In Figure 3.8, it can be seen that *BspRatio* is derived from the *BspTime* (and the *TotalTime*), which is derived from the duration of the *BspIteration* operation. Finally, the Duration of *BspIteration* operation is derived from its *StartTime* and *EndTime*. For each information, there is also a descriptive paragraph which explains how the information is calculated from the sources. Since the derivation step is simple, the description is sufficient for understanding of calculation.

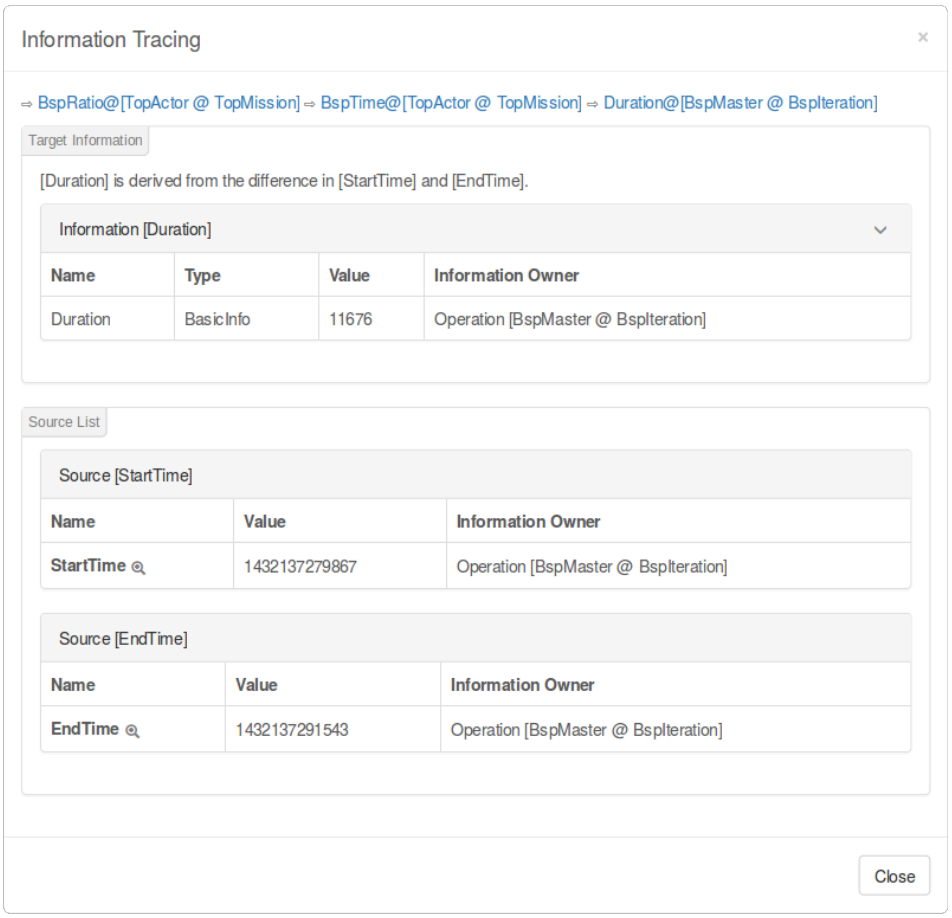


Figure 3.8: Information Tracing Process.

For analysts like developers or researchers, it is important to verify whether each step in the evaluation process is correct. By using the visualizer, each piece of information can be traced to its original sources, hence solve the provenance problem. Analysts like beginner users (everyone is at first a beginner users) can study how the performance evaluation is done and know the exact meaning of each performance metric.

3.4.2. THE OUTCOME

The Granula visualizer facilitates efficient communication and sharing among analysts by converting performance archives into human-readable reports. Viewers can easily explore the complex relationships between operations by navigating the operation hierarchy. To avoid ambiguity, every piece information is described explicitly how they are derived from which sources.

For further development, we intend to develop a platform where users can share the performance archives built for their BDP jobs, which allows other analyst to review the performance of their BDP jobs.

3.5. DISCUSSION

In this section, we evaluate the system design by elaborating how the system components meet the system requirement.

3.5.1. REQUIREMENT FULFILLMENT

Here we need to discuss how the list of requirements/goals can be fulfilled by the three Granula modules collaboratively, or independently.

- **The reuse of code:** can be found in the implementation of Modeler, Archiver and Visualizer. While the modeling process is system-dependent, models are built with rules, which can be further extended from a set of basic rules. Code injection can be based on a previous patch. The Granula archiver and visualizer are fully system-independent, which means analysts do not need to spend any implementation effort on those modules.
- **The reuse of data:** with Granula archiver, a comprehensive performance archive can be built which is self-contained, self-explanatory. The archive format is machine-readable, which means it's queryable i.e. using an XML querying language. With the Granula visualizer, the archive is also human-readable, and both useful for basic and advanced use scenarios.
- **The reuse of methods:** Granula models the performance of BDP job with a set of language elements which are generic. We always explore the performance of a BDP job with its operation hierarchy, following similar technique from creating a performance model, to archiving performance results to visualize the performance archive. In this way, analysts can get use to the way of working and pick up efficiently.
- **The reuse of knowledge:** Granula encourages collaboration among performance analysts by facilitating an efficient sharing process, and by allowing performance analysts with different expertise to participate in the evaluation process.

By improving on the reusability of performance evaluation process, we aim at putting performance evaluation in smaller cycle, adaptable to the fast changes in the environment and fully integrate performance evaluation in the Big Data ecosystem.

3.5.2. APPLICATION DOMAIN

Here we discuss from different perspectives the applicability of Granula on BDP system.

Granula is applicableto BDP systems in the category of high-level applications and execution engines. Granula is particularly applicable to batch-based, multi-stage BDP systems. For example, iterative MapReduce processing, or large-scale graph processing. It is less useful for transactional systems. In general, when the concept of a job is complex and time-consuming in a system, then Granula is applicable. When a job is short-lived and simple, with a duration depending on the resource management architecture of the system, then Granula is less applicable.

Granula is most effective between the high-level and the low-level job operations. For the high-level, a BDP job can be modeled by Granula, which is in the same level of abstraction as in "black-box" evaluation. Granula can explore the internal operations of a BDP job up to the low-level operations which are highly repetitive. In those cases, a counter is used to aggregate performance information.

3.5.3. THREATS TO VALIDITY

Here we discuss a few potential issues that may challenge the validity or the practicality of the Granula framework and the possible ways to mitigate such issues.

UNSYNCHRONIZED TIME AMONG COMPUTATION NODES

The time in each computation node always deviates slightly from the standard time. As Granula collects performance information from the logs generated by BDP systems, the collected information may contain unsynchronized timestamp. This inaccuracy may cause statistical errors in the results.

The time inaccuracy problem is unfortunately unavoidable, and performance analysts should always be alerted to statistical errors caused by this problem and try sync the time in their experiment environment.

An impractical solution is to set up a server for listening to incoming Granula logs in real-time, which consequentially introduces network overhead and more importantly adds network latency to the time inaccuracy problem.

However, we consider the impacts of the unsynchronized time problem to be rather limited. First of all, not all performance information is time-based. Second, time-based performance information derived from the timestamps in the same computation node is not affected by the unsynchronized time among different computation nodes. As our framework is completely transparent, all performance information can be traced back to its source to verify whether such information is indeed time-based.

What remains here is merely performance information regarding synchronization overhead. Based on the results from our experiments in chapter 5, and by using our visualization module, we can observe that synced operations usually start within a few milliseconds. Performance analysts can easily verify whether the BDP systems running in their environment achieve similar level of synchronization accuracy.

OVERHEAD OF THE LOGGING MECHANISM

The logging mechanism of BDP systems introduces additional performance overhead. As Granula relies on this logging mechanism to collect performance data, Granula inevitably introduces more overhead.

This issue is mitigated with several solutions. First of all, during the modeling phase, the design of a Granula patch should consider the number of repetitions that certain logs will cause. For a small number of repetitions, the logging overhead is limited. Analysts should be cautious not to inject logs in a function loop resulting in millions or billions of repetition, and use a counter in this scenario and log the aggregated result instead.

3.6. SUMMARY

In this chapter, we propose Granula, a fine-grained performance evaluation framework for BDP systems which facilitates an incremental, continuously-evolving evaluation method for BDP systems. The incremental method focuses on improving the efficiency of performance evaluation by allowing performance analysts to substantially reuse code, data, method and most importantly knowledge of previous work on the same BDP system.

Granula consists of three main module: the modeler, the archiver and the visualizer, which map respectively to the process of performance modeling, performance archiving and performance visualizing. The Granula modeler allows performance analysts to define their evaluation method for a BDP system explicitly in a performance model, which can be reproduced by other performance analysts step-by-step. The performance model created by the Granula modeler defined explicitly enough that a performance archive can be built programmatically with the Granula archiver for each BDP job, which serves as a snapshot that encapsulates various performance information regarding that particular BDP job in a structured way. While a performance archive is sufficiently informative, it is not the most natural way of browsing through the performance results. The Granula visualizer presents the performance archive in a human-readable manner, and allows efficient navigation of performance results, such that the performance results can be easily communicated among performance analysts.

By using Granula, the performance evaluation process can be integrated as an essential part of the Big Data ecosystem. This liberates performance analysts from repetitive operations, and allows even beginner users to follow exactly the same performance evaluation method for their tasks.

4

PERFORMANCE MODELING OF LARGE-SCALE GRAPH PROCESSING SYSTEMS

By using *Granula*, performance analysts are able to develop fine-grained performance models for BDP systems. To prove this point, in this chapter we present two fine-grained performance models built each for one of the two large-scale graph processing platforms we have focused on in this work, Giraph and GraphX.

4.1. OVERVIEW

In the previous chapter, we described an incremental performance evaluation method for Big Data Processing (BDP) systems, which is supported by the *Granula* framework. This incremental evaluation method is put into practice by using the *Granula* modeler to develop fined-grained performance models for two large-scale graph processing systems, namely Giraph and GraphX. The modeling process defines explicitly the steps of collecting, analyzing, and presenting the performance information of the jobs running on a BDP system. By building such performance models, we aim to enable other users to easily repeat the same evaluation process on their custom workload, and to allow other analysts to examine or even extend the performance models presented here, with possibly more in-depth models (for example, that link the observed performance with resource utilization information).

In this thesis report, we can only described the performance model at a high level due to space limitation and in the interest of presenting material interesting for the general reader. For example, we do not fully explain the low level operations and all performance information collected for each operation. Therefore, a *Granula* archive (see Appendix A) has been built as a demonstrator for each graph processing system we model, each archive describes the performance model comprehensively. By exploring the demo *Granula* archives, other analysts are able to study the performance model more efficiently than they could do by reading a comprehensive but too technical report.

4.1.1. SELECTION OF GRAPH PROCESSING SYSTEMS

We carry out fine-grained exploration on Apache Giraph, one of the large-scale graph processing system. To show that our evaluation method is not system-dependent, we also explore the performance of another graph processing system, one with a different system design than Giraph. We choose for this purpose the high-performance graph-processing system GraphX. To assess the usefulness of Granula when used to model a new system, we set a time limit of 1.5 weeks for creating a comprehensive performance model for GraphX. (The result is a less comprehensive model, compared to the comprehensive model we provide for Apache Giraph.) The two systems we select for this work are thus:

- **Giraph** [8] is among the first open-source BDP systems designed for large-scale graph processing. Giraph implements the Bulk Synchronous Parallel (BSP) [6] model, which proceeds in a series of synchronized global supersteps. In each superstep, all active vertices execute computation and deliver messages to other vertices. Giraph uses Yarn for resource provisioning and allocation and HDFS for distributed storage.
- **GraphX** [9] is a graph processing library that runs on Apache Spark, a general-purpose distributed workflow framework written in Scala. GraphX provides a Pregel-like abstraction which converts iterative superstep into Spark stages. GraphX also uses Yarn for resource provisioning and allocation. Graph datasets are stored in HDFS, which are first loaded into RDDs and subsequently processed by GraphX.

System	Version	Last Update	Archive in Appendix A
Giraph	Giraph 1.1.0	Oct 28, 2014	Archive-01
GraphX	Spark 1.1.1	Sep 11, 2014	Archive-02

Table 4.1: List of selected large-scale graph- processing systems.

4.1.2. MODELING ABSTRACTION

Software systems, especially distributed systems, are highly complex and difficult to understand. Therefore, models are useful in defining the terminology of a hardware and software system, and to explain the relationship between conceptual components. A system can be modeling on different levels of abstraction. Although a high-level model is easier to understand, due to the encapsulated complexity, a low-level model contains more (valuable) information of the system; this is particularly important for performance evaluation studies where system bottlenecks must be discovered. In *Granula*, we build performance models for Big Data Processing jobs, focusing on the three levels of modeling abstraction highlighted in Figure 4.1 4.1.

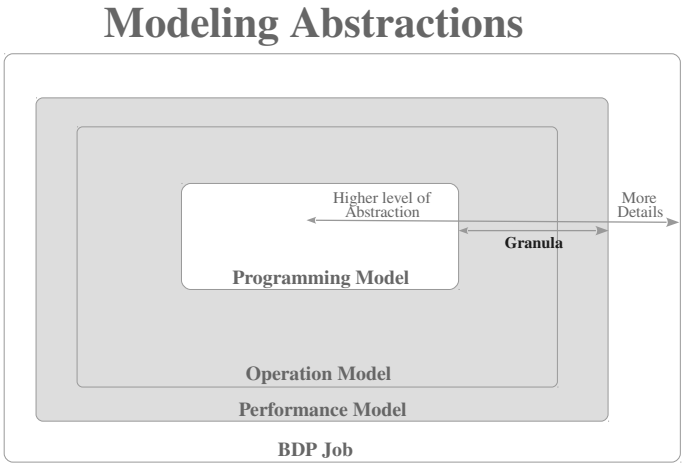


Figure 4.1: The modeling abstractions on which Granula is focusing: the programming model, and especially the operational and performance models.

PROGRAMMING MODEL

The programming model is often the first collection of concepts being explained when introducing to a new BDP system, because the programming model contains the minimum knowledge that a user needs to acquire in order to write application codes for that BDP system. The programming model usually assumes that the parallelized operations are uniform and the distributed environment is homogeneous. Operations such as resource allocation, scheduling, and deployment are entirely excluded from the programming model concepts, to reduce complexity. While this high-level abstraction reduces the learning curve for beginners, its

users cannot detect most of the performance-related issues, and as a consequence cannot find out directly how to improve the performance of their BDP jobs.

OPERATIONAL MODEL

The operational model is a superset of the programming model, which conceptualizes the internal operations of a job running on a BDP system. An operational model provides the context on which performance-related issues can be accurately mapped to the associated operations, and performance issues can be typically diagnosed. The BDP job is the highest level operation of each BDP system. In contrasts to the black-box evaluation method which considers a BDP job as the only observable entity, our method allows analysts to evaluate the performance of a BDP job recursively down to its smallest underlying sub-operations. Thus, a BDP system can be studied systematically, by increasingly refining the operation model with more low-level operations.

The depth of the operational model depends firstly on the available time for fine-grained performance evaluation and secondly on the interestingness of further investigation. For example, operations can be neglected when their duration is so short that in-depth optimization is meaningless. Other situations when it is more efficient to stop further refining the investigation are when an operation is already reduced to the point that it only contains numerous repetitive sub-operations, and when the system overheads have already been isolated from the main data processing operations.

PERFORMANCE MODEL

The performance model is a superset of the operational model, in that it attaches performance-related information to the internal operations defined in the operational model. While the operational model does not distinguish operations of the same type, the performance model captures the difference in performance between such operations, which is crucial in performance analysis. Of all performance information, the duration of an operations (derived from the start time and the end time) is usually the most important, and is thus recorded for each operation.

For the lower-level operations *Granula* usually records the raw (source) information reported by the BDP system. By analyzing the source information, analysts can derive new information for the operations at the higher levels. As a practical example, an analyst can define the duration of a high-level operation as the sum of durations of its constituent low-level operations; more complex derived information can be derived in *Granula*. In this way, performance information can be analyzed step-by-step from low-level operations to high-level operations. Eventually, performance metrics can be derived at the top operation, which describes the overall performance of the BDP job. As we show in the remainder of this chapter, this composition process is effective in describing complex performance models.

4.2. GIRAPH MODEL

Inspired by Google's Pregel [16], Apache Giraph[8] is one of the firstly developed open-source large-scale graph processing system, which is included in many performance studies of graph processing systems.

4.2.1. PROGRAMMING MODEL OF GIRAPH

Giraph implements the Bulk Synchronous Parallel (BSP) [6] model, which proceeds in a series of synchronized global supersteps. In each superstep, all active vertices in the graph receive incoming messages as parameters, execute user-defined computation, and then send messages to some other vertices. When all vertices finish in a superstep, they can then bypass the synchronization barrier and start with the next superstep.

The BSP model applies a vertex-centric approach to achieve parallelism, in which users can define their graph algorithms by modeling the behavior of a vertex. In Figure 4.2, a Giraph vertex program is written for the BFS algorithm. User can can define in the compute method how to interpret the received messages, modified the vertex value and send out new messages for each vertex.

Programming Model of Giraph

```
public void compute(Vertex<I, V, E> vertex, Iterable<M> messages) {
  if (getSuperstep() == 0) {
    vertex.setValue(new IntWritable(Integer.MAX_VALUE));
  }
  int minDist = isSource(vertex) ? 0 : Integer.MAX_VALUE;

  for (IntWritable message : messages) {
    minDist = Math.min(minDist, message.get());
  }

  if (minDist < vertex.getValue().get()) {
    vertex.setValue(new IntWritable(minDist));
    for (Edge<IntWritable, NullWritable> edge : vertex.getEdges()) {
      int distance = minDist + 1;
      sendMessage(edge.getTargetVertexId(), new IntWritable(distance));
    }
  }
  vertex.voteToHalt();
}
```

Figure 4.2: The programming model of Giraph (a Giraph job for BFS computation).

4.2.2. OPERATIONAL MODEL OF GIRAPH

We summarize the internal operations of Giraph in Figure 4.3.

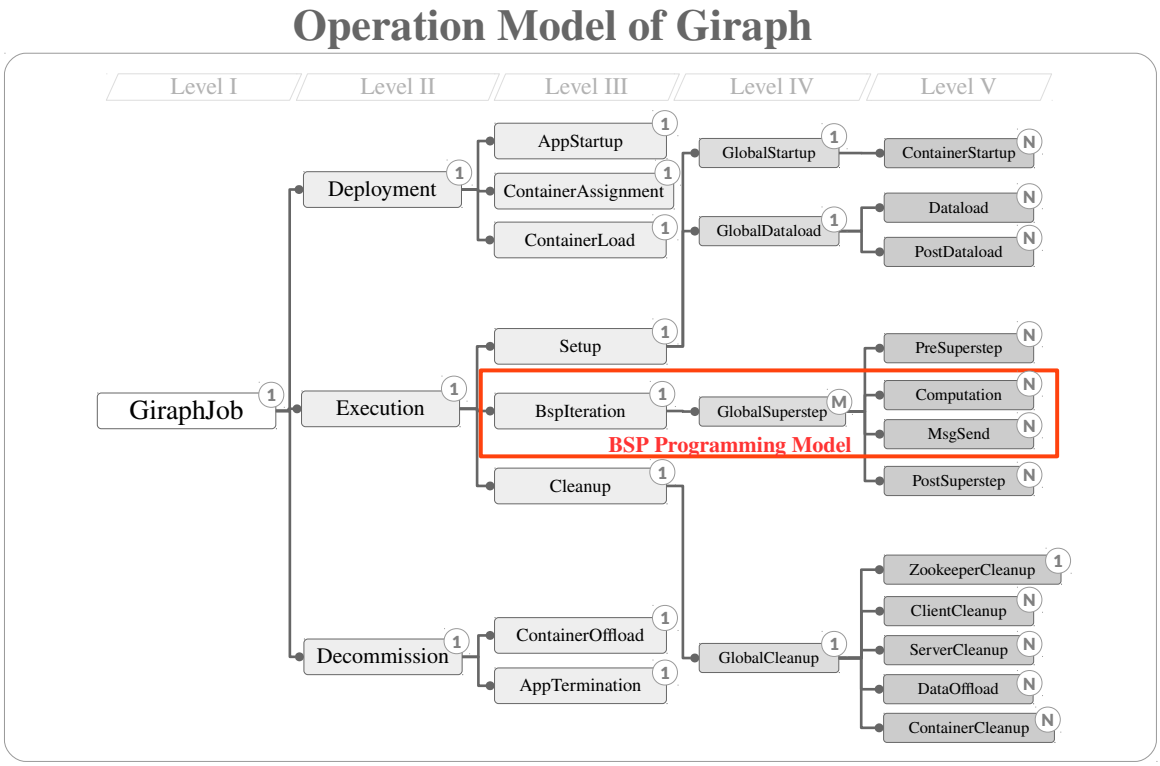


Figure 4.3: The Operation Model of Giraph.

The *GiraphJob* operation is at the top level (Level I) of the performance model, which consists of three sub-operations at Level II, namely *Deployment, Execution*, and *Decommission*. The Deployment operation ini-

tializes the Giraph application, requests containers (computation resources) from the resource manager, and then load the Giraph BSP program into each requested container. The Decommission operation waits for the requested containers to finish after the *BspExecution* and then terminates the Giraph application. Together, the Deployment and the Decommission operations are responsible for the resource provisioning and allocation process, and are often disregarded by performance studies regardless of their actual duration.

The Execution operation at Level II coordinates the execution of the BSP process, which consists of three child operations: *BspSetup*, *BspIteration*, and *BspCleanup*. The *BspSetup* operation prepares for the actual BSP supersteps by defining the role of each container (as either the BspMaster or the BspWorker), and then instructs the BspWorkers to load the input dataset into the system. The *BspCleanup* operation cleanup the (sub)systems after all supersteps are done and write the output data into the Distributed File System. Together the *BspSetup* and the *BspCleanup* operations set the distributed environment ready for BSP supersteps.

The *BspIteration* operation at Level III consists of a number of *GlobalSuperstep* operations. Of the entire *GiraphJob*, only this operation carries out the actual data processing, as defined by the application developer using the BSP programming model. In each *GlobalSuperstep* operation, each BspWorker prepares for the superstep in the *PreSuperstep* operation, does vertex-based computation in the *Computation* operation, waits for the completion of the message sending process through the *MsgSend* operation, and finally waits for slower BspWorkers to finish before starting another *GlobalSuperstep*.

4.2.3. PERFORMANCE MODEL OF GIRAPH

The overall execution time $T_{overall}$ of Giraph can be decomposed into *overhead time* ($T_{overhead}$), *IO time* (T_{io}), and *processing time* ($T_{processing}$).

$$T_{overall} = T_{overhead} + T_{io} + T_{processing}$$

- **overhead time** ($T_{overhead}$) is the system overhead to the overall execution time. In Giraph, $T_{overall}$ is the sum of *resource allocation time* (T_{alloc}) and *coordination time* (T_{coord}). T_{alloc} expresses the performance overhead caused by the Deployment and the Decommission operations in each Giraph job. T_{coord} expresses the performance overhead caused by the BspSetup and the BspCleanup operations, minus T_{io}.
- **IO time** (T_{io}) is the time required for data loading and offloading to the overall execution time. In Giraph, it is the sum of duration of the GlobalDataload and GlobalDataOffload operations.
- **processing time** ($T_{processing}$) is the time needed for the actual data processing. In Giraph, it is the duration of BspIteration.

By using this model and the empirical data presented in Section 5.5, we can conclude on the real-world system overhead presented in Giraph.

4.3. GRAPHX MODEL

GraphX [9] is a graph processing system that runs on the top of Apache Spark, a distributed workflow framework written in Scala. While many graph processing systems has a specialized system model comparing to that of general-purpose data processing systems such as Apache Hadoop and Apache Spark, there are several reasons for using Spark for graph processing. First, Spark uses Resilient Distributed Datasets (RDD), which avoid spilling to disk for iterative processing such as BSP. Further, graph processing and other type of processing can be mixed together without an additional and complicated extraction process, which is advantageous for many use cases and can lead to high-performance processing.

4.3.1. PROGRAMMING MODEL OF GRAPHX

Similarly to Giraph, GraphX also implements the Pregel engine, which proceeds in a series of synchronized global supersteps. However, the implementation of GraphX is more restrictive comparing to that of Giraph.

For each graph processing job, users define three functions: the vertex program, the send message program, and the merge program. In the vertex program, the user defines how to process the incoming messages and how to store the result as the vertex value. In the send message program, the user defines for each edge of the graph (but not arbitrary link between any two vertices), whether a message will be sent, in each direction, and what the value of the message is.

Programming Model of GraphX

```
def compute(graph : Graph[Boolean, Int]) =
    preprocess(graph).pregel(getInitialMessage, getMaxIterations)(vertexProgram, sendMsg, mergeMsg)

def vertexProgram = (vertexId : VertexId, oldValue : Long, message : Long) =>
    math.min(oldValue, message)

def sendMsg = (edgeData: EdgeTriplet[Long, Int]) =>
    if (edgeData.srcAttr < Long.MaxValue && edgeData.srcAttr + 1L < edgeData.dstAttr)
        Iterator((edgeData.dstId, edgeData.srcAttr + 1L))
    else
        Iterator.empty

def mergeMsg = (messageA : Long, messageB : Long) => math.min(messageA, messageB)

def getInitialMessage = Long.MaxValue
```

Figure 4.4: The Programming Model of Giraph.

4.3.2. OPERATIONAL MODEL OF GRAPHX

We summarize the internal operations of GraphX in Figure 4.5. To be able to compare with the operational model we have proposed for Giraph, and to keep within the time limitation we have auto-imposed as the duration of a modeling process usable in practice, we only model the Pregel Engine of GraphX comprehensively (the non-Pregel data processing operations in GraphX are grouped directly under higher level operation instead of under supersteps).

The *GraphXJob* operation is at the top level (Level I) of the performance model, and consists of three sub-operations at Level II, namely *Deployment*, *Execution* and *Decommission*. The Deployment operation initializes a Spark Driver, which requests containers (computation resources) from the Yarn resource manager, and then loads a Spark Executor into each requested container. The *Decommission* operation terminates the Spark Driver after the *Execution* is finished. Together, the *Deployment* and the *Decommission* operations are responsible for the resource provisioning and allocation process.

The Execution operation at Level II coordinates the execution of the BSP process, and consists of three child operations: *Setup*, *BspIteration* and *Cleanup*. The *Setup* operation prepares for the actual BSP supersteps loading the input dataset into the RDD. The *Cleanup* writes the output data back into the Distributed File System. Together, the *Setup* and the *Cleanup* operations make the distributed environment ready for BSP supersteps.

The *BspIteration* operation at Level III consists of a number of *GlobalSuperstep* operations. Of the entire *GraphXJob*, only this operation carries out the actual data processing defined by the application developed through the BSP programming model. Each *GlobalSuperstep* operation can be divided into *VertexUpdate*, *MsgSend* and *MsgCount* operations. *VertexUpdate* operation propagates the new vertex value from the VertexRDD to the EdgeRDD. The *MsgSend* operation checks the active vertices, creating messages in the EdgeRDD. And finally the *MsgCount* operation materializes the RDDs by checking the sent message.

4.3.3. PERFORMANCE MODEL OF GRAPHX

Similarly to our process for creating the performance model of Apache Giraph, for GraphX we focus on a performance model that includes only the duration of operations. The performance metrics we use here for GraphX is (almost) identical to that of Giraph, as the performance model allows us to encapsulate the

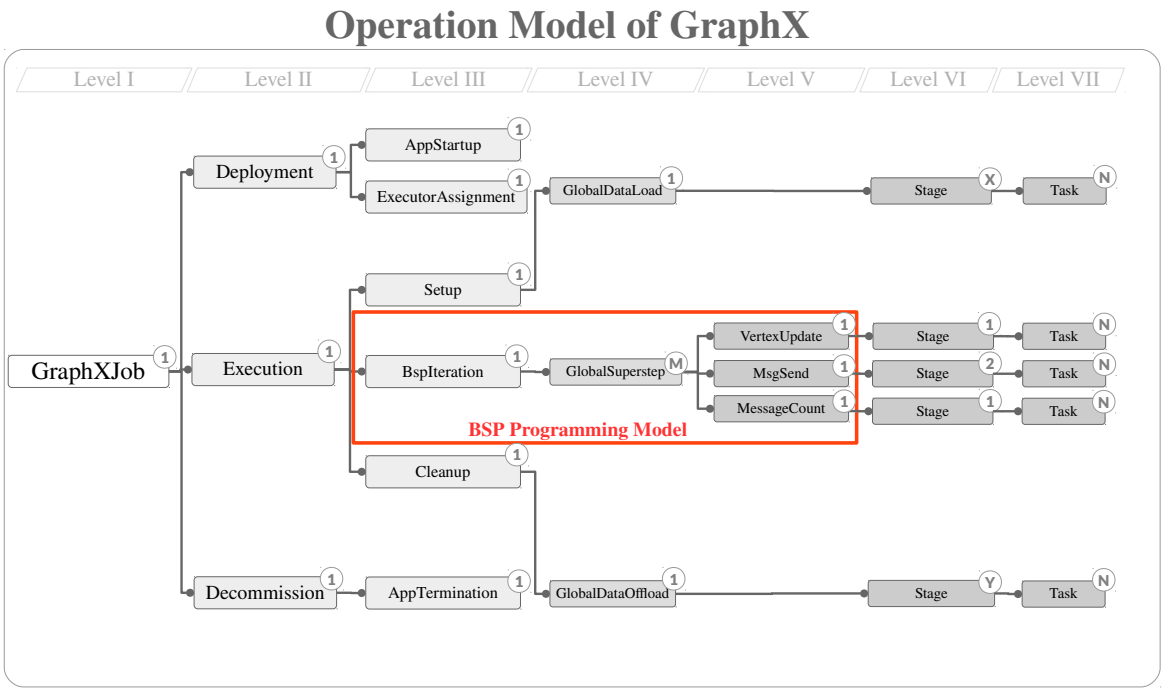


Figure 4.5: The Operation Model of GraphX.

low-level implementation differences in graph processing systems, such that we can reason about their performance at high level in similar way.

The overall execution time $T_{overall}$ of GraphX can be decomposed into *overhead time* ($T_{overhead}$), *IO time* (T_{io}), and *processing time* ($T_{processing}$).

$$T_{overall} = T_{overhead} + T_{io} + T_{processing}$$

- **overhead time** ($T_{overhead}$) is the system overhead to the overall execution time. In GraphX, $T_{overall}$ is the sum of *resource allocation time* (T_{alloc}) and *coordination time* (T_{coord}). T_{alloc} expresses the performance overhead caused by the Deployment and the Decommission operations in each GraphX job. T_{coord} expresses the performance overhead caused by the Setup and the Cleanup operations, minus T_{io}.
- **IO time** (T_{io}) is the time required for data loading and offloading to the overall execution time. In GraphX, it is the sum of duration of the GlobalDataload and GlobalDataOffload operations.
- **processing time** ($T_{processing}$) is the time needed for the actual data processing. In GraphX, it is the duration of BspIteration.

By using this model and the empirical data presented in Section 5.5, we can conclude on the real-world system overhead presented in GraphX.

4.4. DISCUSSION

In this work, we advocate for an incremental method of developing performance models for large-scale graph processing systems. The use of an the incremental method implies that the performance models are subjected to further refinement in each iteration. Here, we discuss the completeness of these performance models in terms of depth, coverage, and validity, and the factors that affect the development progress.

The Granula framework is highly efficient in exploring the internal operations of a BDP system. In our experience, an operational model with 5 - 6 levels can be developed within a week even for an unfamiliar BDP system with its high complexity. However, while a meaningful depth of the operational model can be reached rather easily, it is worth mentioning that in these systems there can be alternative implementation of similar functions, which may not be fully covered by the model yet. This requires more careful investigation in the design of the system. Performance metrics are added by the analyst (the Granula user) to the operational model, to create the performance model. It is not always obvious what is the exact definition of the gathered performance information, and finding and extracting these performance metrics are rather time-consuming. We leave as a future direction of research the automatic extraction of meaningful and necessary performance information.

Furthermore, in the future, the validity of these performance models should also be checked not only the researchers, but also by developers and users. Just like other software products, bugs cannot be totally eliminated. However, under enough eyeballs, bugs are typically eliminated. Developers have the deepest understanding of the system, and users can explore the models in their practical use cases. We see the process we propose with Granula as an important step in enabling model validation: the way the performance evaluation is structured in this work, it can be easily examined by many types of analysts.

4.5. SUMMARY

By using the Granula framework, we build fine-grained job performance models for two large-scale graph processing systems, Giraph and GraphX. Obtaining in-depth understanding of the performance of the jobs running on large-scale graph-processing systems is challenging for many users. Although these systems usually have a well-explained programming model, the programming model itself is too abstract and does not contain performance information. In this work, we introduce operational model, which recursively defines a graph processing job by its internal operations. By mapping performance data to the internal operations, analysts can create meaningful performance models for graph-processing jobs running on Giraph and on GraphX.

These fine-grained job performance models can be used to explain in details how a graph processing job operates internally and can lead to in-depth understanding of the performance of each operation. We created an exemplary performance archive for each of Giraph and GraphX. Using the Granula visualizer, our users can study the performance model embedded in these performance archives. Furthermore, the evaluation steps embedded in the performance model are integrated into the source code of the graph-processing systems, which means that the experiments we conduct in Chapter 5 are reproducible by other analysts.

5

FINE-GRAINED PERFORMANCE EVALUATION OF LARGE-SCALE GRAPH PROCESSING SYSTEMS

In this work, we conduct a fine-grained performance evaluation on large-scale graph-processing systems, which extends non-trivially on our previous work [12] on benchmarking graph-processing systems. By using the *Granula* framework (proposed in Chapter 3), the performance models are built in Chapter 4 for Giraph and GraphX. We assess the performance of these two systems with fine-grained performance metrics, and study the impact of various workloads and of resource availability on performance of these BDP systems.

5.1. OVERVIEW

Large-scale graphs are increasingly used in many application domains, such as social network analysis, business intelligence, and bioinformatics. Since Google published their seminal paper on Pregel [16], many different large-scale graph processing systems have been proposed, notably Apache Giraph [8], the GraphX library [9] of Apache Spark, and GraphLab [10]. The diversity of the available graph processing systems makes it challenging for users to select the most suitable system for their application domain.

Although several performance studies have already been done on (large-scale) graph-processing systems, there are yet unresolved challenges preventing us from gaining a comprehensive understanding on the performance of these systems (see Section 1.3). Therefore, in Chapter 3, we proposed an increment evaluation method for performance evaluation on Big Data Processing (BDP) systems, which is supported by the Granula framework. In Chapter 4, we define two performance models, one for each of Giraph and GraphX. In this chapter, we use the Granula method to study the performance of two graph processing systems, namely Apache Giraph [8] and GraphX [9].

We propose an evaluation method for an in-depth performance evaluation of graph processing systems. This evaluation method consists of the following steps:

- **Quantitative Analysis of Graph Workload**
  In many performance studies, a workload consists of graph algorithms and graph datasets. Usually, the performance characteristics of workload is ambiguously defined. For example, dataset properties such as number of vertices and edges are stated quantitatively. However, due to the dynamicity of graph processing, the dataset size alone cannot explain the stress incurred by the workload. Algorithm properties are only described qualitatively. In contrast to prior work, in this chapter, we perform quantitative analysis of graph workloads using the performance metrics we defined in the performance model. This analysis is reusable by other analysts for their own specific workload.
- **Quantitative Analysis of Overall Job Performance**

We quantify the overall job performance of graph-processing jobs. This is similar to a traditional black-box evaluation method. This step provides context for further fine-grained evaluation, and also serves as a basis for comparing our results with the results of previous studies using black-box models.

- **Quantitative Analysis of System Overhead**
  We quantify the performance overhead of graph processing jobs. As defined (in Chapter 4) by the performance decomposition metrics of the performance models, $T_{overall}$ can be decomposed into *overhead time* ($T_{overhead}$), *IO time* (T_{io}) and *processing time* ($T_{processing}$). In this work, we try to access the impact of system overhead on the overall performance of Giraph and GraphX jobs. By pointing out elements of poor performance that may be eliminated without affecting needed processing, this type of quantitative analysis enables future research on tuning and possibly even optimizing BDP systems.

- **Quantitative Analysis of Data Load/Offloading Operations**
  Big Data Processing is data-intensive, and the duration of data loading and offloading operations (T_{io}) are not static, as they are impacted by the size of the data, the underlying DFS, and the data loading and offloading method. We quantify their duration through an in-depth study enabled by Granula.

- **Quantitative Analysis of Data Processing Operations**
  Finally, we evaluate the time need for the actual data processing operations ($T_{processing}$). Without the $T_{overhead}$ and T_{io}, which we analyze before reaching this part of the study, this analysis of data processing operations is more accurate than what can be achieved by the traditional (black-box-based) analysis. We compare quantitatively the performance of data processing operations in Giraph and GraphX.

5.2. EXPERIMENTAL SETUP

In this section, we list the real-world graph workloads used in this performance study; we describe the hardware and software configuration of our experimental environment; and we assess how Granula supports fine-grained performance evaluation in this performance study.

5.2.1. REAL-WORLD WORKLOAD

This performance study uses real-world workload to assess the performance of graph processing systems. In our previous work by Guo et al. [12], a taxonomy of commonly-used graph algorithms (Table 5.1) and graph datasets (Table 5.2) was defined, which will also be used in this study.

- **Graph Algorithm** We select a list of commonly used graph algorithms (Table 5.1) based on the survey conducted by Guo et al. [12], which are representative in their functionality group. Here we use the graph algorithm implementations of both Giraph and GraphX provided by the Graphalytics project [17].

- **Graph Dataset** We reuse the real-world graph datasets (Table 5.2) selected by Guo et al. [12]. These datasets are converted into the vertex-list format: each line starts with the vertex id, and continues with tab-separated list of neighboring vertex ids.

ID	Name	Group
NCC	Network Clustering Coefficient	General Statistics
BFS	Breadth-first Search	Graph Traversal
CC	Connected Component	Connectivity
CD	Community Detection	Clustering

Table 5.1: Graph algorithms.

By combining all graph algorithms and graph datasets described in this section, we define a real-world workload consisting of 24 graph processing jobs.

ID	Vertices	Edges	Data Volume
Amazon [18]	262,000	1,234,000	17 MB
WikiTalk [18]	2,388,000	5,018,000	66 MB
KGS [19]	293,290	16,558,839	210 MB
Citation [18]	3,764,000	16,511,000	280 MB
DotaLeague [20]	61,171	50,870,316	655 MB
Friendster [18]	65,608,366	1,806,067,135	31 GB

Table 5.2: Graph datasets.

5.2.2. HARDWARE AND SOFTWARE SPECIFICATION

We deploy the graph-processing systems (Table5.3) on the Distributed ASCI Supercomputer 4 (DAS4) [21], which is designed by the Advanced School for Computing and Imaging to provide a common computational infrastructure for ASCI researchers in the Netherlands.

System	Version	Last Update
Giraph	Giraph 1.1.0	Oct 28, 2014
GraphX	Spark 1.1.1	Sep 11, 2014

Table 5.3: List of selected large-scale graph- processing systems.

Each computation node in DAS4 contains an Intel Xeon E5620 2.4 GHz CPU (dual quad-core, 12 MB cache) and 23.0 GiB of memory. DAS4 nodes used for these experiments also contain additional storage such as HDD disk array (2 SATA disks in RAID0) configured with the XFS file system. All these computation nodes are connected by Infiniband. For this performance evaluation, we use 5 - 20 computational nodes, each assigned with a 20000MB Yarn container. CentOS 6.5 kernel version 2.6.32 is installed as the operation system on each node. Java(TM) SE Runtime Environment (build 1.7.0-b147) is installed as the Java Virtual Environment.

Processor	Intel Xeon E5620 (Dual Quad-Core, 2.4 GHz, 12 MB Cache)
Memory	23.0 GiB
Storage	HDD
Network	Infiniband

Table 5.4: Hardware specification.

Operation System	CentOS 6.5 kernel version 2.6.32
Java Virtual Environment	Java(TM) SE Runtime Environment (build 1.7.0-b147)
Resource manager	Yarn 2.4.1
Distributed File System	HDFS 2.4.1 (blocksize = 64MB, repetition = 3)

Table 5.5: Software specification.

5.2.3. EXPERIMENT LIST

We run the 24 graph processing jobs defined in Section 5.2.1 on Giraph and GraphX. We conduct 3 separate experiments, in which 5, 10, and 20 computational nodes are used. The performance results are assembled into 6 Granula archives, which are listed in Appendix A. By using queries defined in Appendix B, we can easily extract the experiment results from these performance archives. In Table 5.6, we explain how the performance results described in Section 5.4, 5.5 and 5.6, and 5.7 can be extracted.

Furthermore, we evaluate the accuracy of our experimental results by repeating each experiment 3 times. Due to time limitation we cannot repeat the experiments more frequently. In Figure 5.1, we can observed that for Giraph, the relative standard deviation (standard deviation / mean) is between 0 to 15 %, and for

Section	Goal	Environment	Job	Archive	Query
§ 5.3	Extract V, V_T, $V_{T,act}$	20 DAS4 nodes	24 Giraph jobs	Arc_5	Qry_{A1}
§ 5.3	Extract E, E_T, $M_{T,rec}$	20 DAS4 nodes	24 Giraph jobs	Arc_5	Qry_{A2}
§ 5.4	Extract $T_{overall}$	20 DAS4 nodes	24 Giraph jobs	Arc_5	Qry_{B1}
§ 5.4	Extract $T_{overall}$	20 DAS4 nodes	24 GraphX jobs	Arc_8	Qry_{B2}
§ 5.5	Extract $T_{overhead}$	5-20 DAS4 nodes	72 Giraph jobs	Arc_3, Arc_4, Arc_5	Qry_{C1}
§ 5.5	Extract $T_{overhead}$	5-20 DAS4 nodes	72 GraphX jobs	Arc_6, Arc_7, Arc_8	Qry_{C2}
§ 5.5	Extract T_{alloc}, T_{coord}	5-20 DAS4 nodes	72 Giraph jobs	Arc_3, Arc_4, Arc_5	Qry_{C3}
§ 5.5	Extract T_{alloc}, T_{coord}	5-20 DAS4 nodes	72 GraphX jobs	Arc_6, Arc_7, Arc_8	Qry_{C4}
§ 5.6	Extract T_{io}	5-20 DAS4 nodes	72 Giraph jobs	Arc_3, Arc_4, Arc_5	Qry_{D1}
§ 5.6	Extract T_{io}	5-20 DAS4 nodes	72 GraphX jobs	Arc_6, Arc_7, Arc_8	Qry_{D2}
§ 5.6	Extract T_{io}, $T_{processing}$	5-20 DAS4 nodes	72 Giraph jobs	Arc_3, Arc_4, Arc_5	Qry_{D3}
§ 5.6	Extract T_{io}, $T_{processing}$	5-20 DAS4 nodes	72 GraphX jobs	Arc_6, Arc_7, Arc_8	Qry_{D4}
§ 5.7	Extract $T_{processing}$	20 DAS4 nodes	24 Giraph jobs	Arc_5	Qry_{E2}
§ 5.7	Extract $T_{processing}$	20 DAS4 nodes	24 GraphX jobs	Arc_8	Qry_{E2}

Table 5.6: List of experiments results.

GraphX the relative standard deviation is between 0 to 19 %. This observation indicates that the variance in overall execution time of these graph processing jobs is relatively small.

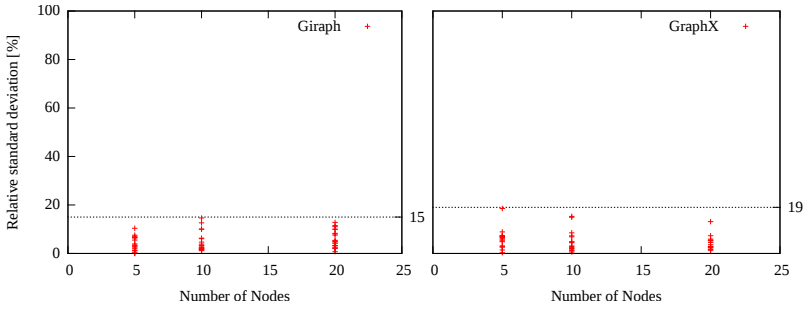


Figure 5.1: Relative standard deviation of the overall execution time of Giraph and GraphX jobs with 3 repetitions.

We also evaluate the overhead of *Granula* logs on the graph processing systems. We repeating each experiment 3 times, with and without code modification from using *Granula*. In Figure 5.2, we can observe that for Giraph, the performance overhead of *Granula* code modification is between -17 to 13 % and for GraphX the performance overhead is between -16 to 16 %. These observation indicates that Granula does not introduce significant performance overhead, and considering the observable variance in the overall execution time, the actual performance overhead can be expected to be even less.

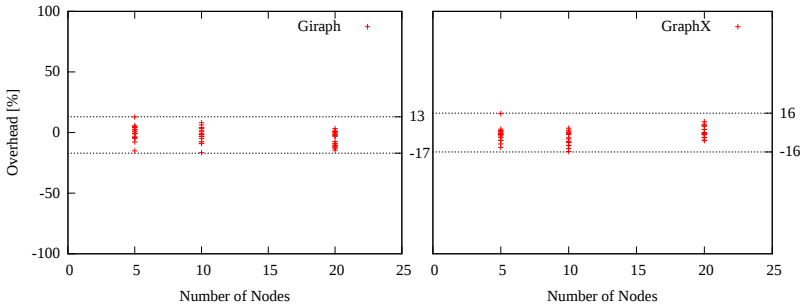


Figure 5.2: Overhead of the Granula framework on Giraph.

5.3. QUANTITATIVE ANALYSIS OF GRAPH WORKLOAD

Graph processing is different from other types of Big Data processing in the irregularity of the graph data structure, which allows arbitrary linking between any two vertices. Due to this irregularity, the performance of a graph processing job is more difficult to predict than that of a general-purpose data processing job (i.e. MapReduce job). Besides the properties of the graph data, many other factors such as the complexity of the graph algorithm and the implementation of the graph programming model also have their impacts on the performance. In performance studies, the graph workload under experimentation consists of a list of graph processing jobs, each in combination of a graph dataset and a graph algorithm. In many studies, the graph workload is usually not sufficiently quantitatively described. Users cannot understand the actual performance stress of the graph processing jobs on the system based on primitive performance information such as file size, number of vertices, number of edges etc.

We are able to collect many fine-grained performance information by the performance model we built for Giraph (thus not GraphX). To take a step further, we are able to derived more indicative performance metrics from more primitive performance information. We propose a few descriptive performance metrics that are indicative to the actual demand on each computation resources i.e. computation and network.

COMPUTATION-RELATED PERFORMANCE METRICS

- *vertices* (V) is the number of vertices in the initial graph dataset.
- *total vertices* (V_T) is the total number of vertices aggregated from all supersteps.
- *total active vertices* ($V_{T,act}$) is the total number of active vertices aggregated from all supersteps.

In a vertex-centric graph processing system such as Giraph, each vertex represents the execution of a vertex program. Therefore, the number of vertices V in the graph dataset can be used to estimate the demand on the computation resources. However, note that only the active vertices are processed in each superstep, therefore the total number of active vertices in all supersteps $V_{T,act}$ is a better indicator for the computation intensity.

Figure 5.3 shows the **vertex activation**: the ratio of $V_{T,act}$ to V_T for each Giraph job. For this metric, the value of zero indicates that the vertex program was not executed at all, and the value of one indicates that the vertex program was executed for all vertices in all supersteps (this further indicates that the algorithm is compute-intensive, relative to the situation when the ratio is close to a value of zero). It can be observed that the BFS algorithm has a ratio of 0.05 - 0.53, and the CC algorithm has a ratio of 0.40 - 0.80, which indicates that both BFS and CC are not compute-intensive. The CD algorithm and the NCC algorithm have a ratio of 1.00, which indicates that they are both compute-intensive, as all vertices are active in each superstep.

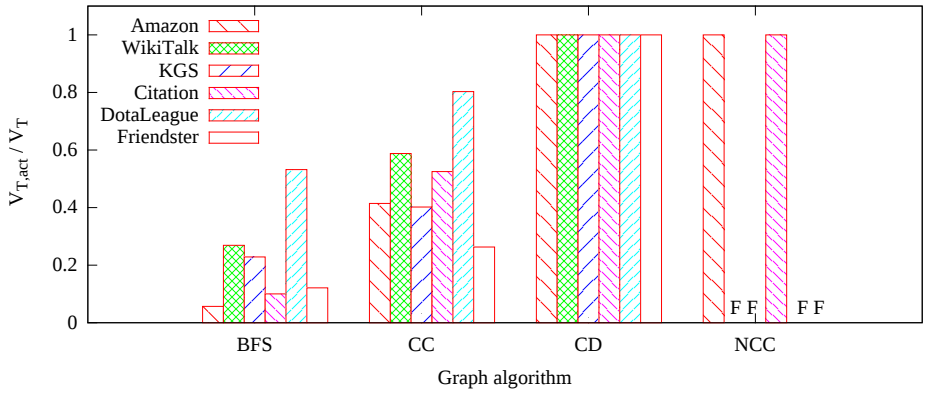


Figure 5.3: The ratio of the total number of active vertices ($V_{T,act}$) to the total number of vertices (V_T) of each graph processing job in the Giraph workload, grouped by graph algorithms.

Figure 5.4 shows the **vertex repetitiveness**: ratio of $V_{T,act}$ to V for each Giraph job. This metric indicates in average how many times a vertex program is executed for each vertex. It can be observed that BFS and CC algorithms need less computation time than the CD algorithm, as there are less vertex programs to be

executed. The CD algorithm scores high in this metric, as it is instructed to redo the same execution for 10 iterations. By contrasting Figure 5.3 and Figure 5.4, it can be observed that while both the NCC and the CD algorithm are relatively compute-intensive, the execution time of the NCC algorithm is expected to be much shorter than the CD algorithm.

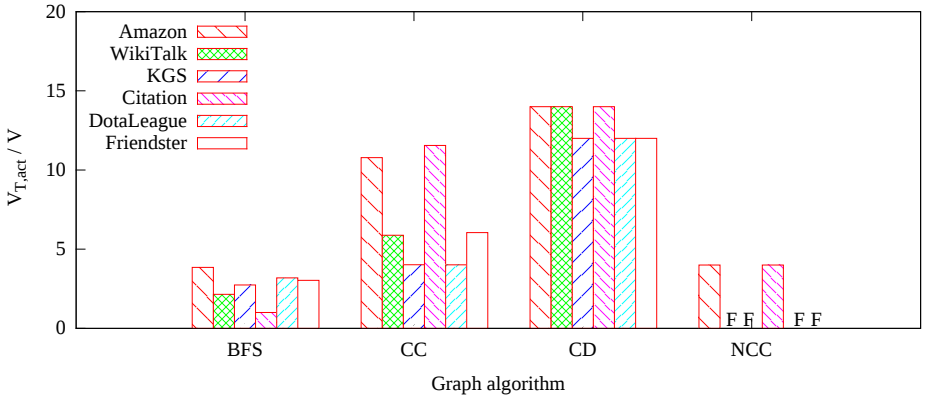


Figure 5.4: The ratio of the total number of active vertices ($V_{T,act}$) to the *initial* number of vertices (V) of each graph processing job in the Giraph workload, grouped by graph algorithms.

While the vertex activation and the vertex repetitiveness provide better insights on the computation intensity of a Giraph job than the number of vertices, these metrics is still an estimation, as an assumption is made implicitly that all vertex program has the same execution time. However, a more accurate estimation is difficult, as the execution time of a vertex program is dependent heavily on the user's implementation which cannot be easily generalized for all applications.

NETWORK-RELATED PERFORMANCE METRICS

- *edges* (E) is the number of edges in the initial graph dataset.
- *total edges* (E_T) is the total number of edges aggregated from all supersteps.
- *total received message volume* ($M_{T,rec}$) is the total volume of messages received.

In the Bulk Synchronous Model, communication is done by means of sending and receiving messages. The network intensity of a Giraph job can be measured by the total volume of messages sent in all supersteps ($\sum M_{sent}$). However a message can be sent locally, which should not be included in the network traffic. Therefore, the total volume of messages received remotely ($M_{T,rec}$) is better performance metric to indicate network intensity.

Figure 5.5 shows **edge volume**: the ratio of $M_{T,rec}$ to E. This metrics indicates in average, how many bytes of messages are received per edge over time. It can be observed that the BFS algorithm received 7 - 10 bytes per edge, and the CC algorithm received 18 - 61 bytes of messages per edge. Both BFS and CC do not produce much messages. On the other hand, the CD and NCC algorithm can produce up to 270 bytes per edges.

Figure 5.6 shows the **edge flow**: the ratio of $M_{T,rec}$ to E_T. This metrics indicates how many bytes of messages are sent per edge per superstep. For BFS and CC, the average message sent per edge is between 0 - 4 bytes in each superstep. For CD, the average message is betwen 14 - 20. However, for NCC, the average message per edge can be up to 66 bytes. This could explain why many giraph job running NCC fail, even when the dataset size is small.

KEY FINDINGS

- The *vertex activation* shows the percentage of active vertices, which can be used to estimate computation intensity.
- The *vertex repetitiveness* shows how frequently a vertex program is executed, which can be used to estimate the computation usage.

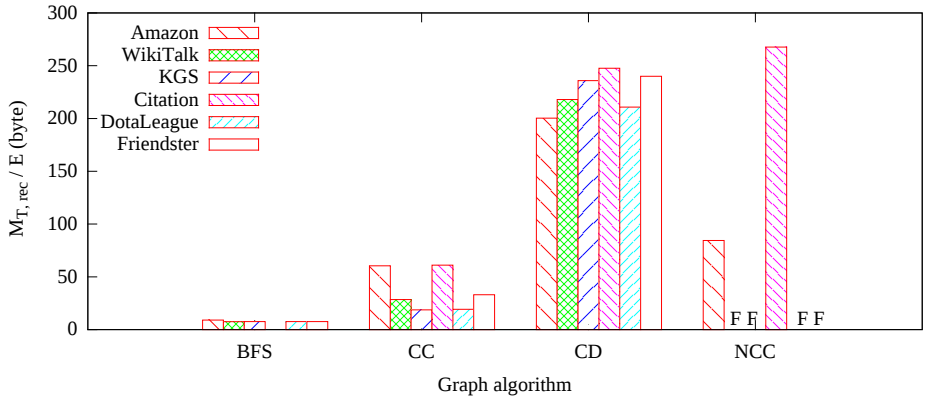


Figure 5.5: The ratio of the total volume of messages ($M_{T,rec}$) in all supersteps to initial number of edges (E) in all supersteps, for each job in the Giraph workload grouped by graph algorithms.

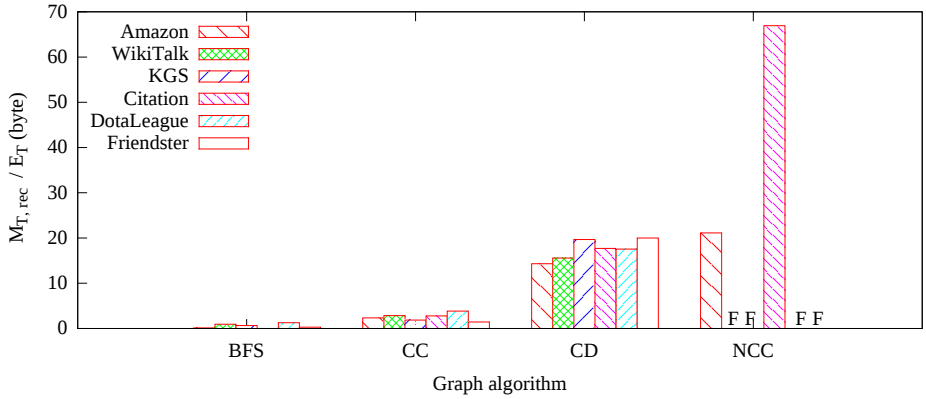


Figure 5.6: The ratio of the total volume of messages ($M_{T,rec}$) in all supersteps to the total number of edges (E_T) in all supersteps, for each job in the Giraph workload grouped by graph algorithms.

- The *edge volume* shows in average how many messages are being sent per edges, which can be used to estimate the network usage.
- The edge flow shows in average how many messages are sent per superstep per edge, which can be used to estimate the network intensity.

5.4. QUANTITATIVE ANALYSIS OF OVERALL JOB PERFORMANCE

In this section, we aim at quantifying the overall job performance of Giraph and GraphX by using the performance metric *overall time* $T_{overall}$. For both Giraph and GraphX, 24 jobs (4 algorithm × 6 datasets) are executed on 20 DAS4 computation nodes.

EXPERIMENT RESULTS

$T_{overall}$ of both Giraph and GraphX jobs are depicted in Figure 5.7 and Figure 5.8. 20 of 24 Giraph job succeed, and 16 of 25 GraphX job succeed. Jobs denoted with the F symbol are either faulty or automatically terminated if still running after 2 hours.

By just looking at the $T_{overall}$ of the Giraph jobs, it seems that most Giraph jobs finish within 30 seconds to 2 minutes, regardless of the variations in graph algorithms and datasets. Two exceptions are the jobs using Friendster dataset and the jobs running NCC algorithm. The Friendster dataset is proportionally much larger in size, which causes observable difference in $T_{overall}$ of its jobs. NCC algorithm generates a lot of messages, which causes its job to run out-of-memory.

For the GraphX jobs, the same observation applies: most jobs complete within 30 seconds to 2 minutes. However, the average GraphX job running the CD algorithm takes longer than 15 minutes, whereas the average corresponding Giraph job takes only 2 minutes (7 times less!). The GraphX jobs using Friendster dataset do not complete after 2 hours and therefore killed.

It seems that GraphX has a longer $T_{overall}$ execution time than Giraph for most job. However, using only the $T_{overall}$, it cannot be explained why many jobs with much variations in algorithms and datasets all complete in 30 seconds to 2 minutes. Certain types of system overhead must be hidden in the overall execution time.

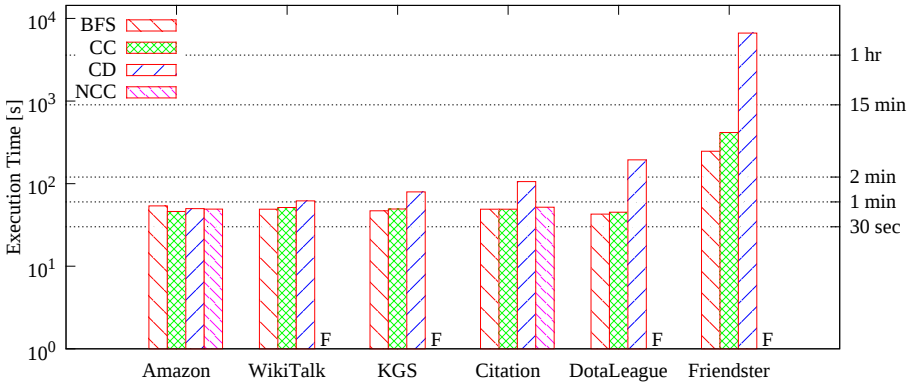


Figure 5.7: $T_{overall}$ for Giraph jobs with $20 \times 2000MB$ containers.

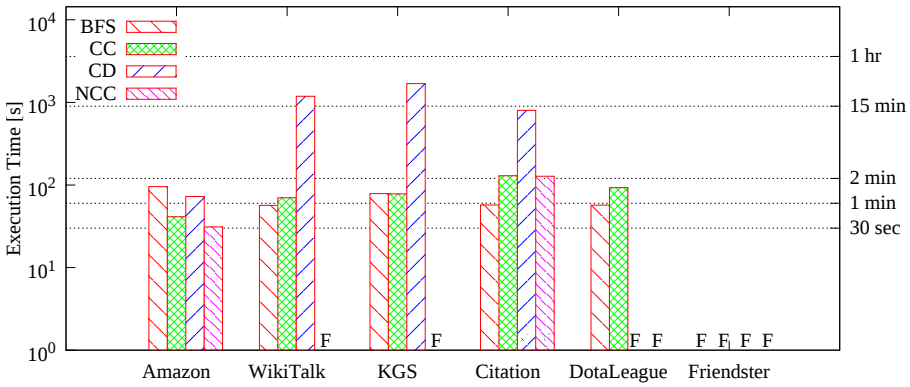


Figure 5.8: $T_{overall}$ for GraphX jobs with $20 \times 2000MB$ containers.

KEY FINDINGS

- For the graph workload we use, many Giraph and GraphX jobs have a $T_{overall}$ between 30 seconds to 2 minutes.
- Similar $T_{overall}$ despite the variations in algorithms and datasets indicates hidden system overhead in $T_{overall}$, which is further investigated in next section.

5.5. QUANTITATIVE ANALYSIS OF SYSTEM OVERHEAD

In this section, we aim at quantifying the performance overhead of both Giraph and GraphX job by using the performance metrics i.e. *overhead time* $T_{overhead}$, *resource allocation time* T_{alloc} and *coordination time* T_{coord}. For both Giraph and GraphX, 24 jobs (4 algorithm * 6 datasets) are executed on 5, 10, and 20 DAS4 computation nodes.

EXPERIMENT RESULTS

$T_{overhead}$ of both Giraph and GraphX jobs are depicted in Figure 5.9 and Figure 5.10. 57 of 72 Giraph job succeed, and 48 of 72 GraphX job succeed.

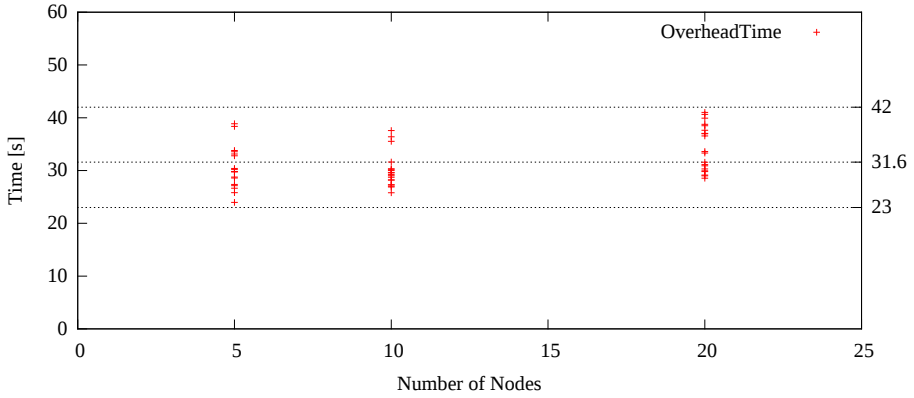


Figure 5.9: $T_{overhead}$ for Giraph jobs.

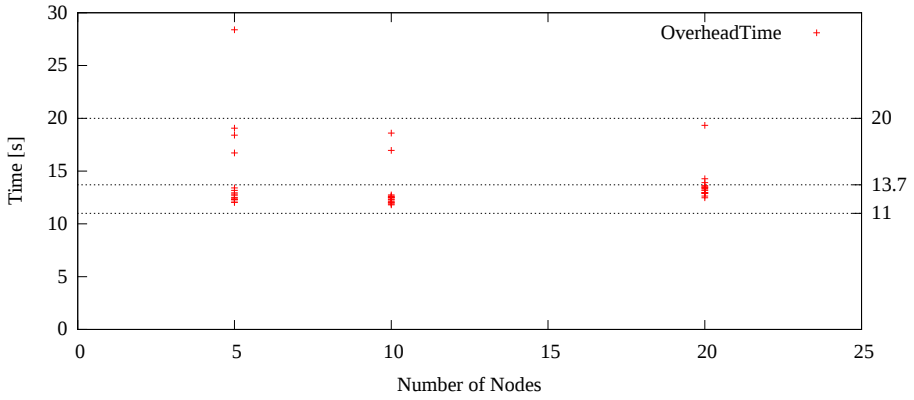


Figure 5.10: $T_{overhead}$ for GraphX jobs.

In Figure 5.9, it can be observed that for Giraph jobs, $T_{overhead}$ is appr. between 23 to 42 seconds. $T_{overhead}$ of Giraph is averaged at 31.6 seconds (with relative standard deviation of 13.6%). This value does not seem to be impacted by the number of nodes being used.

In Figure 5.10, it can also be observed that for GraphX jobs, $T_{overhead}$ is appr. between 11 to 20 seconds. $T_{overhead}$ of GraphX is averaged at 13.7 seconds (with elative standard deviation of 21.1%). This value does not seem to be impacted by the number of nodes being used.

It can be concluded from the results that for both Giraph and GraphX, there is a constant system overhead on the graph processing jobs, and $T_{overhead}$ of Giraph is in average 31.6 / 13.7 = 2.3 times more than GraphX. To further investigate the difference in system overhead, we break down $T_{overhead}$ into T_{alloc} and T_{coord}, the results of which are shown in Figure 5.11 and Figure 5.12.

In Figure 5.11, it can also be observed that for Giraph jobs, T_{alloc} is appr. between 9 to 17 seconds (averaged at 11.1 seconds). T_{coord} of Giraph is appr. between 13 to 30 seconds (averaged at 20.5 seconds).

In Figure 5.12, it can also be observed that for GraphX jobs, T_{alloc} is appr. between 10 to 18 seconds (averaged at 12.0 seconds). T_{coord} of GraphX is appr. between 0 to 5 seconds (averaged at 1.8 seconds).

The results show that *resource allocation time* T_{alloc} is similar for both Giraph and GraphX, which is the time required for resource allocation (i.e. initializing the application, requesting Yarn containers, waiting for responses from distributed and finally terminating the application.) The operation to request Yarn containers is at least responsible for half of the 11.1 - 12.0 seconds.

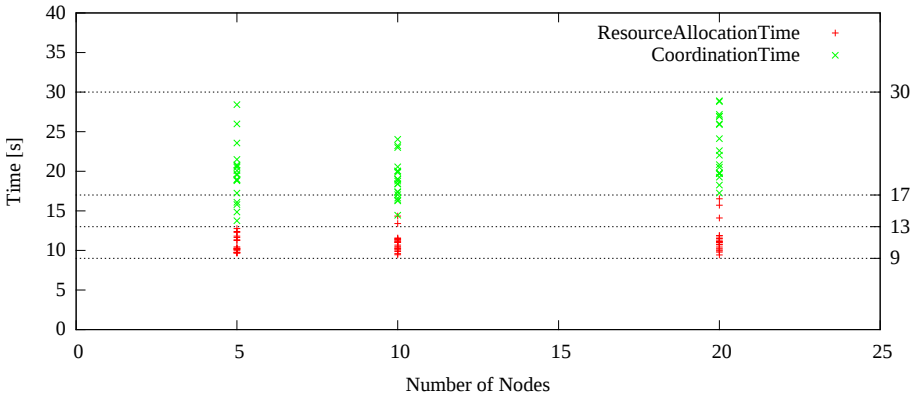


Figure 5.11: T_{alloc} and T_{coord} for Giraph jobs.

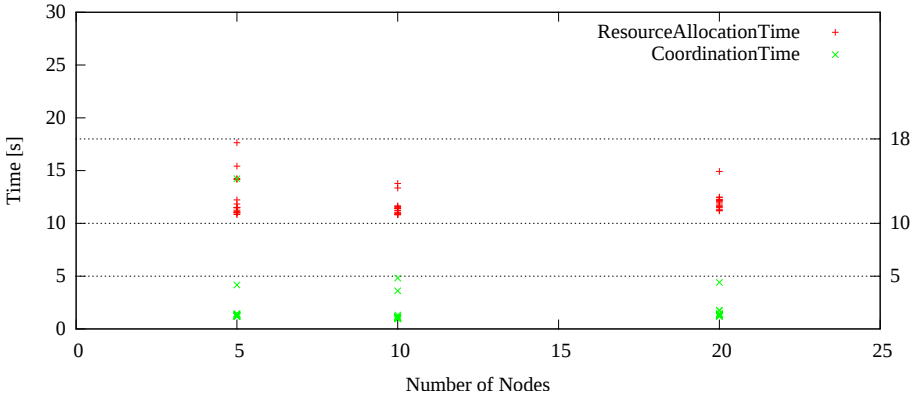


Figure 5.12: T_{alloc} and T_{coord} for GraphX jobs.

The results also show that in average, the *coordination time* T_{coord} in Giraph is 20.5 / 1.8 = 11.4 times longer than that of GraphX. Most of Giraph's T_{coord} is in setting up Zookeeper, cleaning up servers, clients and Zookeeper for each disbtributed worker. On the other hand, GraphX does not require that much time in coordination and just uses the DAGScheduler and TaskScheduler in the driver for assigning jobs to executors.

KEY FINDINGS

- The *overhead time* $T_{overhead}$ of Giraph is averaged at 31.6 seconds, which is 2.3 times more than the $T_{overhead}$ of 13.7 seconds of GraphX. For both Giraph and GraphX, $T_{overhead}$ seems to be a constant value unaffected by the number of nodes used.
- The *resource allocation time* T_{alloc} of both Giraph and GraphX is similar, averaged at 11.1 - 12.0 seconds. Communication with Yarn is responsible for half of the T_{alloc} overhead.
- The *coordination time* T_{coord} of Giraph is averaged at 20.5 seconds, which is 11.4 times more than the T_{coord} of 1.8 seconds of GraphX. Communication with Zookeeper, and cleaning up the server and clients mainly causes the T_{coord} overhead.

From this empirical data, we derive empirical models of the system overhead of both Giraph and GraphX. Let F_{env} be a constant performance factor (CPU, network) describing a static, homogeneous distributed environment,

then for Giraph:

$$T_{overall} = T_{overhead} + T_{io} + T_{processing}$$

$$T_{overall} = 31.6s \times F_{env} + T_{io} + T_{processing}$$

and for GraphX:

$$T_{overall} = T_{overhead} + T_{io} + T_{processing}$$
$$T_{overall} = 13.7s \times F_{env} + T_{io} + T_{processing}$$

From our empirical data, the value of $T_{overhead}$ is unaffected by the number of nodes, the dataset size or the algorithm type. However, do notice that this work suggests rather a first estimation than a final conclusion on the system overhead. More empirical data is needed to test the general applicability and validity of such hypothesis.

5.6. QUANTITATIVE ANALYSIS OF DATA LOAD/OFFLOAD OPERATIONS

In this section, we aim at quantifying the performance of data loading and offloading operations for both Giraph and GraphX jobs by using the performance metrics IO time T_{io} as a function of different file size and number of compute nodes. For both Giraph and GraphX, 24 jobs (4 algorithm × 6 datasets) are executed on 5, 10, and 20 DAS4 computation nodes.

EXPERIMENT RESULTS

T_{io} of both Giraph and GraphX jobs are depicted in Figure 5.13 and Figure 5.14. 57 of 72 Giraph jobs succeed, and 48 of 72 GraphX jobs succeed. For both Giraph and GraphX jobs, it can be observed that jobs using the same dataset with the same number of compute nodes have similar T_{io}. This is as expected as the types of graph algorithm are irrelevant to T_{io}. However, for GraphX jobs running CD algorithm, a heavy preprocessing operation is applied during the data loading operation, for consistency reasons those jobs are excluded from Figure 5.14.

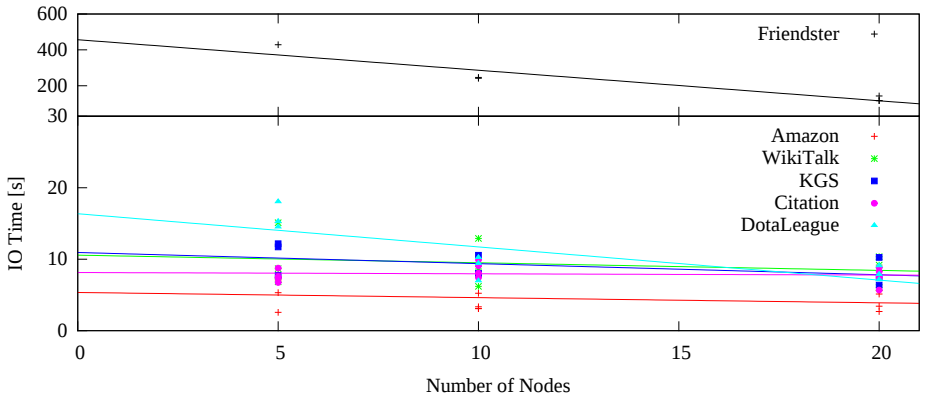


Figure 5.13: T_{io} for Giraph jobs.

In Figure 5.13, it can be observed that for tiny datasets such as Amazon (17 MB), it takes around 5 seconds for data loading and offloading in Giraph. For jobs with small datasets such as WikiTalk (66 MB), KGS (210 MB) and Citation (280 MB), T_{io} is appr. 10 seconds, regardless of the number of nodes used. As the DFS is configured with a default block size of 64 MB, Giraph required at least the time for loading 1 data block from the DFS. For relatively large datasets such as DotaLeague (655 MB) and Friendster (31 GB), the DataLoad time decreases as more compute nodes are being used.

In Figure 5.14, similar patterns can also be observed that for GraphX jobs. However, T_{io} is appr 2 - 3 times more than that of Giraph. There are several explanations. First, GraphX needs to convert the vertex-based dataset to edge list, while Giraph does not need to perform such operation. And then, GraphX does not always uses all the containers assigned for its job, while Giraph always use all the containers (to load and offload data).

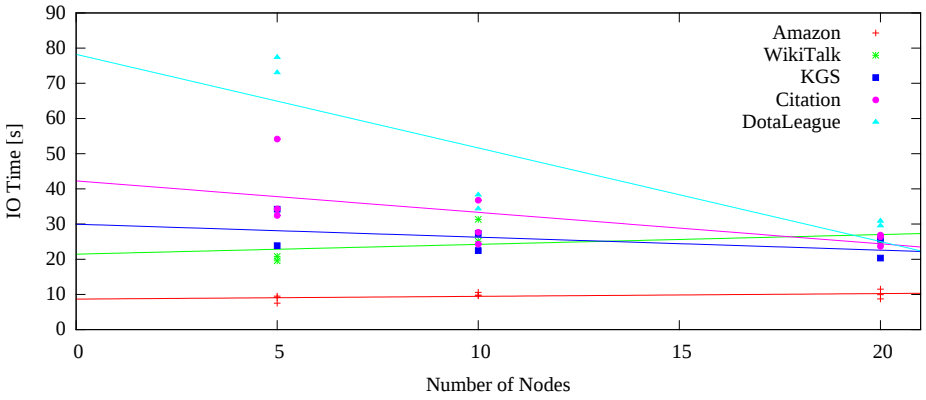


Figure 5.14: T_{io} for GraphX jobs.

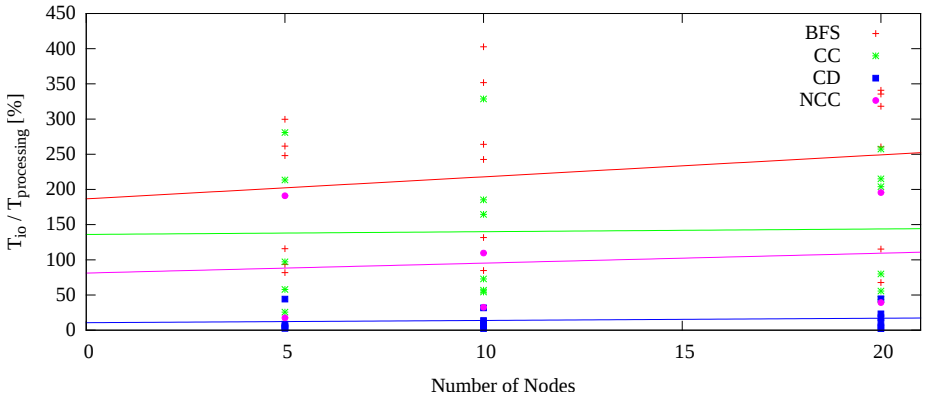


Figure 5.15: Ratio of T_{io} to $T_{processing}$ for Giraph jobs.

Figure 5.15 and Figure 5.16 show the ratio of T_{io} to $T_{processing}$ for Giraph and GraphX jobs. Regardless of the number of nodes used, it can be observed that for many jobs, T_{io} can even be longer than the BSP time. Especially for graph algorithms such as BFS and CC which are not compute-intensive, the average T_{io} is appr. 140% to 200% that of the average $T_{processing}$ in Giraph, and 70% to 110% in GraphX. For compute-intensive graph algorithm such as CD which requires all vertices to be active in all supersteps, T_{io} is however less relevant.

KEY FINDINGS

- For small datasets those are only a few times larger than the DFS block size (usually 64 MB), T_{io} of Giraph and GraphX is at least the time to load one DFS data block, approximately 10 seconds for Giraph and 30 seconds for GraphX.
- For large datasets, T_{io} of Giraph and GraphX is inversely proportional to the number of nodes. Thus the data loading and offloading operations are scalable.
- Regardless of the number of nodes used, if the graph algorithm is not compute-intensive (such as BPS and CC), the average T_{io} can be 140% to 200% that of the average $T_{processing}$.

5.7. QUANTITATIVE ANALYSIS OF DATA PROCESSING OPERATIONS

In this section, we aim at quantifying the performance of data loading and offloading operations for both Giraph and GraphX jobs by using the performance metrics *processing time* $T_{processing}$.

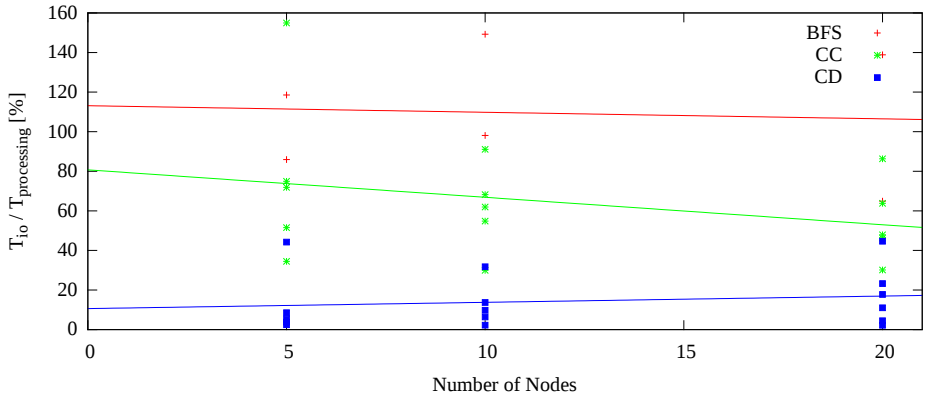


Figure 5.16: Ratio of T_{io} to $T_{processing}$ for GraphX jobs.

EXPERIMENT RESULTS

$T_{processing}$ of both Giraph and GraphX jobs are depicted in Figure 5.17 and Figure 5.18. 20 of 24 Giraph job succeed, and 16 of 25 GraphX job succeed. Jobs (denoted with the F symbol) that are either faulty or still running after 2 hours are terminated. GraphX jobs running NCC algorithm is not using the Pregel engine of GraphX, which is excluded from Figure 5.18.

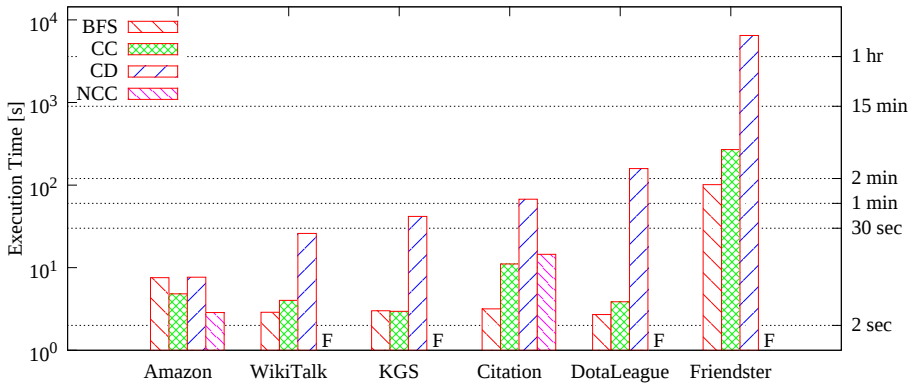


Figure 5.17: $T_{processing}$ for Giraph jobs with $20 \times 2000MB$ containers.

In contrasts to the $T_{overall}$ of Giraph and GraphX jobs, the variations in algorithms is observable with $T_{processing}$. By looking at the $T_{processing}$ of the Giraph jobs, it seems that most Giraph jobs running BFS and CC algorithms have a $T_{processing}$ between 2 to 10 seconds. Giraph jobs running CD algorithms is much longer than jobs running BFS or CC algorithm.

For the GraphX jobs, similar observation can be made. By looking at the $T_{processing}$ of the GraphX jobs, it seems that most GraphX jobs running BFS and CC algorithms have a $T_{processing}$ between 15 to 90 seconds. Most GraphX jobs running CD algorithms is much longer than jobs running BFS or CC algorithm. However, GraphX has a longer $T_{processing}$ than Giraph for all jobs.

The difference in dataset is also observable with $T_{processing}$. Both Giraph and GraphX jobs using the Amazon dataset (17MB) has a longer $T_{processing}$ than jobs using larger datasets such as WikiTalk (66MB) or KGS (210MB) when BFS or CC algorithms are running. As Amazon dataset has a larger diameter, more supersteps are required to traverse the graph in BFS and CC algorithms, hence the longer $T_{processing}$.

Figure 5.19 shows the ratio of $T_{processing}$ of Giraph to $T_{processing}$ of GraphX. It can be observed for most jobs, $T_{processing}$ of Giraph is only 5 to 20 % of $T_{processing}$ of GraphX. For larger data, the data is not available since it takes GraphX more than 2 hours to complete. For GraphX jobs running NCC algorithms are not implemented with the Pregel engine, the execution time is difficult to define.

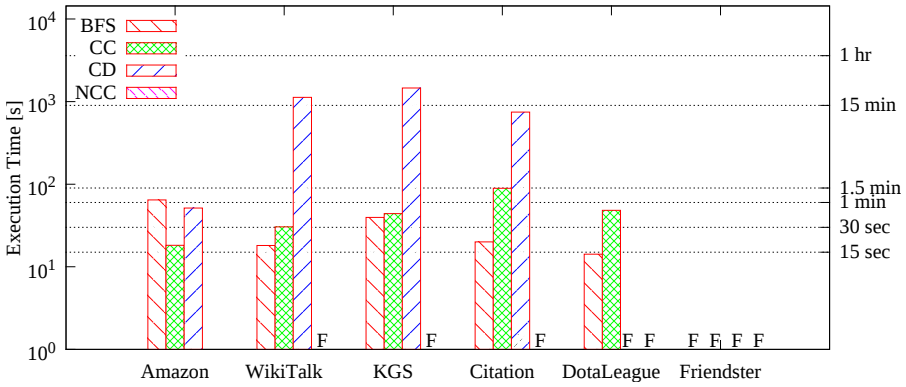


Figure 5.18: $T_{processing}$ for GraphX jobs with $20 \times 2000MB$ containers.

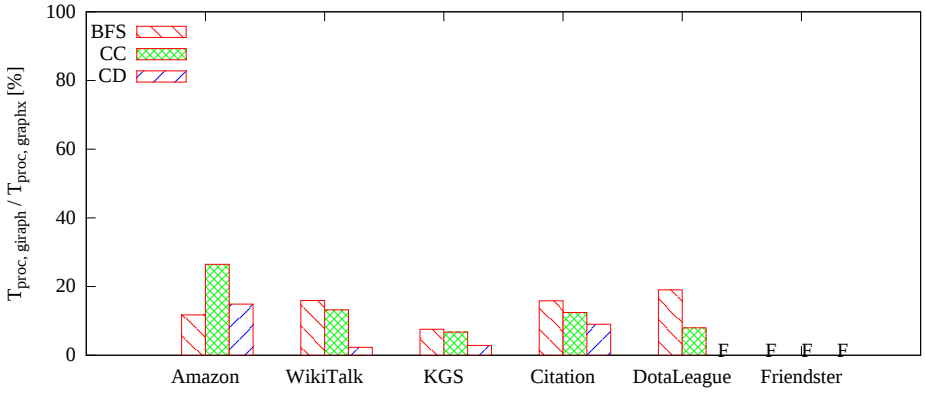


Figure 5.19: The ratio of $T_{processing,giraph}$ to $T_{processing,graphx}$ with $20 \times 2000MB$ containers.

KEY FINDINGS

- In contrasts to the $T_{overall}$ of Giraph and GraphX jobs, the variations in algorithms and datasets is observable with $T_{processing}$.
- It can be observed for most jobs, $T_{processing}$ of Giraph is only 5 to 20 % of $T_{processing}$ of GraphX.

5.8. DISCUSSION

By using the performance model, we decompose $T_{overall}$ into $T_{overhead}$, T_{io} and $T_{processing}$. From this empirical data, we derive the empirical model of the system overhead ($T_{overhead}$), which is a static value for both Giraph and GraphX. By this, we identify the margin of further optimization on the system overhead. For the data loading and offloading operations (T_{io}), the number of nodes N is an important factor. It can be expected that $T_{io} \times N$ is a constant value, as more nodes being used, the less time it requires for data loading and offloading. However, more performance metrics are required for the empirical model of T_{io}. And the main data processing operation $T_{processing}$ is an even more complicated process, and a in-depth exploration of the empirical model of $T_{processing}$ will be listed as the future work.

5.9. SUMMARY

we conduct fine-grained performance evaluation on two graph processing systems, Giraph and GraphX. First, we quantify the performance characteristics of the graph workloads. Then we decompose graph processing jobs into system overhead, IO operations and data processing operations and analyze their impacts on the job performance quantitatively.

- we analyze the workloads used to evaluate graph processing systems and quantify the single job performance using the metrics proposed in the performance models. This information can be used to study why certain jobs encounter performance problem.
- we analyze the overall performance of Giraph and GraphX jobs. For the graph workload we use, many Giraph and Graph jobs have a $T_{overall}$ between 30 seconds to 2 minutes, despite the variations in algorithms and datasets. This indicates hidden system overhead in $T_{overall}$. The actual performance of the data processing operations is not revealed by $T_{overall}$ due to system overhead.
- we analyze and quantify the hidden performance overhead of Giraph and GraphX job. The *overhead time* $T_{overhead}$ of Giraph is averaged at 31.6 seconds, which is 2.3 times more than the $T_{overhead}$ of 13.7 seconds of GraphX. For both Giraph and GraphX, $T_{overhead}$ seems to be a constant value unaffected by the number of nodes used. The *resource allocation time* T_{alloc} of both Giraph and GraphX is similar, while the *coordination time* T_{coord} of Giraph is 11.4 times more than that of GraphX. Communication with Zookeeper, and cleaning up the server and clients mainly causes the T_{coord} overhead.
- we analyze the performance of the main data loading and offloading operations. For small datasets those are only a few times larger than the DFS block size (usually 64 MB), T_{io} of Giraph and GraphX is at least the time to load one DFS data block, approximately 10 seconds for Giraph and 30 seconds for GraphX. For large datasets, T_{io} of Giraph and GraphX is inversely proportional to the number of nodes. Thus the data loading and offloading operations are scalable. Regardless of the number of nodes used, if the graph algorithm is not compute-intensive (such as BPS and CC), the average T_{io} can be 140% to 200% that of the average $T_{processing}$. Therefore improvement on T_{io} can significantly reduces the overall execution time.
- we analyze the performance of the data processing operations of Giraph and GraphX. In contrast to the $T_{overall}$ of Giraph and GraphX jobs, the variations in algorithms and datasets is observable with $T_{processing}$. For most jobs, $T_{processing}$ of Giraph is only 5 to 20 % of $T_{processing}$ of GraphX. Despite the optimization efforts, running graph processing jobs on a general-purpose data processing framework such as Apache Spark may introduce significant overhead.

6

CONCLUSION AND FUTURE WORK

A decade after Apache Hadoop was released, general interest in Big Data still seems to be growing rapidly. Increasingly more BDP systems and applications are incorporated into the Big Data ecosystem, from high-level applications that aim at enhancing usability, to execution engines that support alternate programming paradigms. BDP systems are by definition performance-critical, as both efficiency and cost are deeply affected by performance. To better understand the performance of BDP systems, many studies aim to assess the system performance, to explain the difference in performance between systems, to identify system overheads and performance bottlenecks, and to recommend further improvements on the BDP systems. However, the current state of performance evaluation on BDP systems faces several challenges: coarse-grained evaluation does not provide sufficient insights of the BDP system; fine-grained performance evaluation is too time-consuming; and the results of performance studies are limited in applicability and rarely shared. Together, these elements indicate that the efficiency and effectiveness of BDP performance studies is at best very limited. In this work, we investigated how to improve the efficiency and effectiveness of the process of evaluating the performance of BDP systems.

6.1. CONCLUSION

We described in Chapter 1 three main research questions that we have investigated through this thesis work. We presented our work by answering these questions in Chapter 3, 4, and 5. Here, we summarize our answers:

RQ1: How to facilitate comprehensive performance evaluation on BDP systems? (Chapter 3)
Comprehensive understanding of the performance can only be achieved by in-depth analysis of how well the internal operations of a BDP system perform. However, fine-grained evaluation is extremely time-consuming due to the inefficiency in the evaluation process, which is ultimately caused by the lack of reusability. To facilitate comprehensive performance evaluation, we proposed an incremental evaluation method which focuses on improving efficiency by substantially reuse the previous work on the same BDP system. To support analysts in applying this method, we develop *Granula*, a performance evaluation framework which facilitates performance modeling, archiving and visualizing. By using *Granula*, the performance evaluation of BDP systems can function as a continuously evolving process that integrates into the Big Data ecosystem.

RQ2: How to understand the job performance of large-scale graph processing systems? (Chapter 4)
By using the Granula framework, we build job performance model for two large-scale graph processing systems i.e., Giraph and GraphX. The performance models recursively define a graph processing job by its internal operations. The model explains how the programming model is implemented in each graph processing system, and derives fine-grained job performance metrics from the internal operations. We created an exemplary performance archive for each of Giraph and GraphX, such that users can study the performance model described in these performance archives by using Granula visualizer. The evaluation steps used to build the performance models are integrated into the source code of the graph processing systems, which is reproducible by other analysts running their own graph processing jobs.

RQ3: How well do real-world large-scale graph processing systems perform, and Why? (Chapter 5)
Based on the performance models we built to answer RQ2, we conduct fine-grained performance evaluation on two graph processing systems, Giraph and GraphX. We evaluate these systems by running real-world workloads on these systems, collecting fine-grained job performance data and performing statistical analysis on the collected data. Using metrics defined in each performance model, we assess the performance characteristics of the graph workload. Furthermore, by decomposing the overall execution time of a graph-processing job into overhead time, io time, and processing time, we analyze quantitatively the performance of the internal operations of graph-processing jobs.

6.2. FUTURE WORK

To continue this work, we identify four directions for future research directions:

1. Granula is already efficient in exploring complex internal operations of BDP jobs, and reducing internal operations into simpler sub-operations. However, the low-level data processing operations of BDP systems are highly repetitive. Although their performance is most interesting, lossless preservation of all performance data at that level is no longer possible, due to the magnitude of typical datasets used by big data jobs. Instead, counters are currently used to aggregate performance information. However, simple aggregation does not provide sufficient insight. For example, by means of data aggregation and filtering, is it possible to use a stack trace visualizer such as FlameGraph to visualize the stack traces specifically used by low-level data processing operations? Similar advanced profiling techniques could still be explored for better illustration of those performance data.

2. For the performance model that we built, we only used the performance information reported by the internal operations of Big Data Processing systems. While we were able to collect many fine-grained performance metrics, the information regarding resource utilization reported by the operating system was not used at all. Several considerations led to not including such performance information in our study yet. First, users may experience difficulties in setting up resource monitoring tools in addition to the BDP systems. Second, in most studies, such performance-metrics describe the resource utilization of a computation node, not that of a process running on the node. Third, the monitoring interval for resource utilization is typically much larger than the duration of small operations, which means that the performance and the utilization metrics are difficult to analyze together. A research question for the future is how to combine performance and utilization information in performance studies of BDPs, without exceedingly overloading the system itself with either transferring or processing too many of the measurement results?

3. A Granula archive typically contains many fine-grained performance metrics for each BDP job. The analyst can find out precisely how long each internal operation took. However, the performance model is only descriptive, and in particular cannot predict how long a future BDP job will take. By experience gained through our real-world experiments, we believe that many operations have a constant execution time, and other operations like data loading depend predictably on the data size, the repetition factor, etc. With comprehensive performance information, can we build a predictive performance model to estimate the overall execution time of a future job by estimate the execution time of the internal operations? Which performance information are we still missing?

4. In the current approach, the analyst decides which information are indicative to performance that indicate to viewers where the performance bottleneck is. The next step is to automate the process of tuning and optimizing BDP systems. Can performance bottleneck be identified automatically, by an algorithm that simply analyzes the performance metrics? Which performance metrics are indicative of relevant bottlenecks? And if such performance bottleneck is detected, can an optimization technique be automatically assigned, and configured?

The *Granula* performance evaluation framework allows its user to efficiently study and compare BDP systems, and to effectively express their ideas and concepts on how to evaluate the performance of those systems. Only with a good understanding, we can improve. By providing a comprehensive understanding of the performance of BDP systems, we believe that *Granula* will help analysts in improving the performance of these systems in an incremental manner.

A

LIST OF PERFORMANCE ARCHIVES

This is a collection of the Granula performance archives created and used in this thesis work (in case the links listed below are no longer active, please check `https://github.com/tudelft-atlarge/granula` for newest updates). Arc_1 - Arc_2 aim to explain the performance models of Giraph and of GraphX. Arc_3 - Arc_5 contain the performance results for the Giraph experiments, and Arc_6 - Arc_8 contain the performance results for the GraphX experiments. By providing these performance archives, our goal is to improve the provenance of evaluation results.

Download	`http://52.17.112.251/archive/Giraph-1.1.0-DemoArchive.xml`
Visualization	`http://52.17.112.251/?arc=archive/Giraph-1.1.0-DemoArchive.xml`

Arc_1 (Giraph-1.1.0-DemoArchive) contains the performance archive of an exemplary Giraph job which runs CommunityDetection algorithm on Citation dataset. Viewers can study and explore the Giraph performance model embedded in this performance archive.

Download	`http://52.17.112.251/archive/GraphX-1.1.1-DemoArchive.xml`
Visualization	`http://52.17.112.251/?arc=archive/GraphX-1.1.1-DemoArchive.xml`

Arc_2 (GraphX-1.1.1-DemoArchive) contains the performance archive of an exemplary GraphX job which runs CommunityDetection algorithm on Citation dataset. Viewers can study and explore the GraphX performance model embedded in this performance archive.

Download	`http://52.17.112.251/archive/5Giraph.xml`
Visualization	`http://52.17.112.251/?arc=archive/5Giraph.xml`

Arc_3 (5Giraph) contains the performance archive of 24 standard Giraph jobs defined in section 5.2.1 running on 5 DAS4 nodes. Failed jobs are not included.

Download	`http://52.17.112.251/archive/10Giraph.xml`
Visualization	`http://52.17.112.251/?arc=archive/10Giraph.xml`

Arc_4 (10Giraph) contains the performance archive of 24 standard Giraph jobs defined in section 5.2.1 running on 10 DAS4 nodes. Failed jobs are not included.

Download	`http://52.17.112.251/archive/20Giraph.xml`
Visualization	`http://52.17.112.251/?arc=archive/20Giraph.xml`

Arc_5 (20Giraph) contains the performance archive of 24 standard Giraph jobs defined in section 5.2.1 running on 20 DAS4 nodes. Failed jobs are not included.

Download	`http://52.17.112.251/archive/5GraphX.xml`
Visualization	`http://52.17.112.251/?arc=archive/5GraphX.xml`

Arc_6 (5GraphX) contains the performance archive of 24 standard GraphX jobs defined in section 5.2.1 running on 5 DAS4 nodes. Failed jobs are not included.

Download	`http://52.17.112.251/archive/10GraphX.xml`
Visualization	`http://52.17.112.251/?arc=archive/10GraphX.xml`

Arc_7 (10GraphX) contains the performance archive of 24 standard GraphX jobs defined in section 5.2.1 running on 10 DAS4 nodes. Failed jobs are not included.

Download	`http://52.17.112.251/archive/20GraphX.xml`
Visualization	`http://52.17.112.251/?arc=archive/20GraphX.xml`

Arc_8 (20GraphX) contains the performance archive of 24 standard GraphX jobs defined in section 5.2.1 running on 20 DAS4 nodes. Failed jobs are not included.

B

List of Queries

This is a collection of the XQueries that are used to extract performance information from Granula archives listed in Appendix A. By providing these queries, our goal is allow analysts to explicitly define how they have extracted information from the performance archives, such that the information extraction process is reproducible, given that these archives are available.

```
for $job in $archive/Workload/Job
  let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
  let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
  let $vertices := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="AvgVertices"]/string(@value)
  let $totalVertices := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="Vertices"]/string(@value)
  let $totalActVertices := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="ActiveVertices"]/string(@value)
return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$vertices,' ',$totalVertices,' ',$totalActVertices)
```

Qry_{A1} extracts V, V_T, $V_{T,act}$ from Giraph jobs.

```
for $job in $archive/Workload/Job
  let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
  let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
  let $edges := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="AvgEdges"]/string(@value)
  let $totalEdges := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="Edges"]/string(@value)
  let $totalRecMsgVol := $job//Operation[Mission/@type="BspIteration"]/Infos/Info[@name="ReceivedMsgVolume"]/string(@value)
return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$edges,' ',$totalEdges,' ',$totalRecMsgVol)
```

Qry_{A2} extracts E, E_T, $M_{T,rec}$ from Giraph jobs.

```
for $job in $archive/Workload/Job
  let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
  let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
  let $overallTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Duration"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$overallTime)
```

Qry_{B1} extracts $T_{overall}$ from Giraph jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
    let $overallTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Duration"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$overallTime)
```

Qry_{B2} extracts $T_{overall}$ from GraphX jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
    let $allocTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ResourceAllocTime"]/string(@value)
    let $coordTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="CoordinationTime"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)
return concat($computationClass,' ',$dataset,' ', number($allocTime) + number($coordTime) - number($ioTime))
```

Qry_{C1} extracts $T_{overhead}$ from Giraph jobs, which is T_{alloc} + T_{coord} - T_{io}

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
    let $allocTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ResourceAllocTime"]/string(@value)
    let $coordTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="CoordinationTime"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)
return concat($computationClass,' ',$dataset,' ',number($allocTime) + number($coordTime) - number($ioTime))
```

Qry_{C2} extracts $T_{overhead}$ from GraphX jobs, which is T_{alloc} + T_{coord} - T_{io}

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
    let $allocTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ResourceAllocTime"]/string(@value)
    let $coordTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="CoordinationTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$allocTime,' ',$coordTime)
```

Qry_{C3} extracts T_{alloc} and T_{coord} from Giraph jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
    let $allocTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ResourceAllocTime"]/string(@value)
    let $coordTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="CoordinationTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$allocTime,' ',$coordTime)
```

Qry_{C4} extracts T_{alloc} and T_{coord} from GraphX jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
    let $numContainers := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ContainersLoaded"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$ioTime)
```

Qry_{D1} extracts T_{io} from Giraph jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
    let $numContainers := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ExecutorSize"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$ioTime)
```

Qry_{D2} extracts T_{io} from GraphX jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
    let $numContainers := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ContainersLoaded"]/string(@value)
    let $bspTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="BspTime"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$bspTime,' ',$ioTime)
```

Qry_{D3} extracts T_{io} and $T_{processing}$ from Giraph jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
    let $numContainers := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ExecutorSize"]/string(@value)
    let $bspTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="BspTime"]/string(@value)
    let $ioTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="IOTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$numContainers,' ',$bspTime,' ',$ioTime)
```

Qry_{D4} extracts T_{io} and $T_{processing}$ from GraphX jobs.

```
for $job in $archive/Workload/Job
    let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="Dataset"]/string(@value)
    let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ComputationClass"]/string(@value)
    let $bspTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="BspTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$bspTime)
```

Qry_{E1} extracts $T_{processing}$ from Giraph jobs.

```
for $job in $archive/Workload/Job
   let $dataset := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="DataInputPath"]/string(@value)
   let $computationClass := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="ApplicationName"]/string(@value)
   let $bspTime := $job//Operation[Mission/@type="TopMission"]/Infos/Info[@name="BspTime"]/string(@value)

return concat($computationClass,' ',$dataset,' ',$bspTime,' ',$bspTime)
```

Qry_{E2} extracts $T_{processing}$ from GraphX jobs.

BIBLIOGRAPHY

[1] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters,* Communications of the ACM **51**, 107 (2008).

[2] T. White, Hadoop: The definitive guide (" O'Reilly Media, Inc.", 2012).

[3] *Neo4j,* `http://neo4j.com/` (2014).

[4] A. Kyrola, G. E. Blelloch, and C. Guestrin, *Graphchi: Large-scale graph computation on just a pc.* in OSDI, Vol. 12 (2012) pp. 31–46.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: cluster computing with working sets,* in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Vol. 10 (2010) p. 10.

[6] L. G. Valiant, *A bridging model for parallel computation,* Communications of the ACM **33**, 103 (1990).

[7] N. Doekemeijer and A. L. Varbanescu, *A survey of parallel graph processing frameworks,* (2014).

[8] *Apache giraph,* `http://giraph.apache.org/` (2014).

[9] 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 Proceedings of OSDI (2014) pp. 599–613.

[10] 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,* Proceedings of the VLDB Endowment **5**, 716 (2012).

[11] B. Elser and A. Montresor, *An evaluation study of bigdata frameworks for graph processing,* in Big Data, 2013 IEEE International Conference on (IEEE, 2013) pp. 60–67.

[12] 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,* (IPDPS, 2013).

[13] 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**, 1047 (2014).

[14] Y. Lu, J. Cheng, D. Yan, and H. Wu, *Large-scale distributed graph computing systems: An experimental evaluation,* Proceedings of the VLDB Endowment **8** (2014).

[15] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, *Navigating the maze of graph analytics frameworks using massive graph datasets,* in Proceedings of the 2014 ACM SIGMOD international conference on Management of data (ACM, 2014) pp. 979–990.

[16] 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 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (ACM, 2010) pp. 135–146.

[17] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, *Graphalytics: A big data benchmark for graph-processing platforms,* (2015).

[18] *Snap,* `http://snap.stanford.edu/index.html/` (2014).

[19] Y. Guo and A. Iosup, *The game trace archive,* (IEEE Press, 2012) p. 4.

[20] Y. Guo, S. Shen, O. Visser, and A. Iosup, *An analysis of online match-based games,* in Haptic Audio Visual Environments and Games (HAVE), 2012 IEEE International Workshop on (IEEE, 2012) pp. 134–139.

[21] *The distributed asci supercomputer 4,* `http://www.cs.vu.nl/das4/` (2014).