Improving Scheduling in Heterogeneous Grid and Hadoop Systems

Aysan Rasooli, B.E., M.S.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree PhD of Software Engineering

McMaster University

McMaster University © Copyright by Aysan Rasooli

August 7, 2013

Abstract

Advances in network technologies and computing resources have led to the deployment of large scale computational systems, such as those following Grid or Cloud architectures. The scheduling problem is a significant issue in these distributed computing environments, where a scheduling algorithm should consider multiple objectives and performance metrics. Moreover, heterogeneity is increasing at both the application and resource levels. The heterogeneity in these systems can have a huge impact on performance in terms of metrics such as average completion time. However, traditional Grid and Cloud scheduling algorithms neglect heterogeneity in their scheduling decisions. This PhD dissertation studies the scheduling challenges in Computational Grid, Data Grid, and Cloud computing systems, and introduces new scheduling algorithms for each of these systems.

The main issue in Grid scheduling is the wide distribution of resources. As a result, gathering full state information can add huge overhead to the scheduler. This thesis introduces a Computational Grid scheduling algorithm which simultaneously addresses minimizing completion times (by considering system heterogeneity), while requiring zero *dynamic state information. Simulation results show the promising performance of this algorithm, and its robustness with respect to errors in parameter estimates.*

In the case of Data Grid schedulers, an efficient scheduling decision should select a combination of resources for a task that simultaneously mitigates the computation and the communication costs. Therefore, these schedulers need to consider a large search space to find an appropriate combination. This thesis introduces a new Data Grid scheduling algorithm, which dynamically makes replication and scheduling decisions. The proposed algorithm reduces the search space, decreases the required state information, and improves the performance by considering the system heterogeneity. Simulation results illustrate the promising performance of the introduced algorithm.

Cloud computing can be considered as a next generation of Grid computing. One of the main challenges in Cloud systems is the enormous expansion of data in different applications. The MapReduce programming model and Hadoop framework were designed as a solution for executing large scale data-intensive applications. A large number of (heterogeneous) users, using the same Hadoop cluster, can result in tensions between the various performance metrics by which such systems are measured. This research introduces and implements a Hadoop scheduling system, which uses system information such as estimated job arrival rates and mean job execution times to make scheduling decisions. The proposed scheduling system, named COSHH (Classification and Optimization based Scheduler for Heterogeneous Hadoop systems), considers heterogeneity at both the application and cluster levels. The main objective of COSHH is to improve the av-

erage completion time of jobs. However, as it is concerned with other key Hadoop performance metrics, it also achieves competitive performance under minimum share satisfaction, fairness and locality metrics, with respect to other well-known Hadoop schedulers. The proposed scheduler can be efficiently applied in heterogeneous clusters, in contrast to most Hadoop schedulers, which assume homogeneous clusters.

A Hadoop system can be described based on three factors: cluster, workload, and user. Each factor is either heterogeneous or homogeneous, which reflects the heterogeneity level of the Hadoop system. This PhD research studies the effect of heterogeneity in each of these factors on the performance of Hadoop schedulers. Three schedulers which consider different levels of Hadoop heterogeneity are used for the analysis: FIFO, Fair sharing, and COSHH. Performance issues are introduced for Hadoop schedulers, and experiments are provided to evaluate these issues. The reported results suggest guidelines for selecting an appropriate scheduler for different Hadoop systems. The focus of these guidelines is on systems which do not have significant fluctuations in the number of resources or jobs.

There is a considerable challenge in Hadoop to schedule tasks and resources in a scalable manner. Moreover, the potential heterogeneous nature of deployed Hadoop systems tends to increase this challenge. This thesis analyzes the performance of widely used Hadoop schedulers including FIFO and Fair sharing and compares them with the COSHH scheduler. Based on the discussed insights, a hybrid solution is introduced, which selects appropriate scheduling algorithms for scalable and heterogeneous Hadoop systems with respect to the number of incoming jobs and available resources. The proposed hybrid scheduler considers the guidelines provided for heterogeneous Hadoop systems in the case that the number of jobs and resources change considerably.

To improve the performance of high priority users, Hadoop guarantees minimum numbers of resource shares for these users at each point in time. This research compares different scheduling algorithms based on minimum share consideration and under different heterogeneous and homogeneous environments. For this evaluation, a real Hadoop system is developed and evaluated using Facebook workloads. Based on the experiments performed, a reliable scheduling algorithm is suggested which can work efficiently in different environments.

Acknowledgements

First and foremost, I am heartily thankful to my supervisor, Dr. Douglas Down, whose encouragement, supervision and support from the preliminary to the concluding level enabled me to develop an understanding of the subject. I would like to express deep gratitude to my present and past PhD committee members: Dr. Alan Wassyng, Dr. Frantisek Franek, Dr. Skip Poehlman, and Dr. Kamran Sartipi for agreeing to serve in my PhD committee, and for their insightful comments and discussions. Finally, and most importantly, I am indebted to the love and support that I have received from my family members. In special, I am thankful to my fiancee, my parents, and siblings who made several sacrifices to ensure that I could realize my dream of pursuing a PhD.

This work was supported by the Natural Sciences and Engineering Research Council of Canada. The LCG Grid traces are provided by the HEP e-Science group at Imperial College London. A major part of this work was done while the author was visiting UC Berkeley. In particular, the author would like to thank Randy Katz, Ion Stoica, Yanpei Chen and Sameer Agarwal for their comments on this research. Also, the author gratefully acknowledge Facebook and Yahoo! for permission to use their workload traces in this research.

Dedications

I dedicate this dissertation to my beloved family and friends, especially ...

to my mother who taught me loving someone unconditionally and with no expectations;

to my father for instilling the importance of hard work and higher education;

to my fiancee, Mehran, for his love, patience, advice, encouragement, generosity, and goodness of spirit supported me the hurdles of graduate school;

to my older brother and sister, Aidin and Aylar, for their patience and advices;

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly authored works that I have included in my thesis. The PhD Research problem, i.e., demands for efficient schedulers in heterogeneous Computational and Data Grids and Hadoop systems, and the proposed scheduling systems including COSHH (Classification and Optimization based Scheduler for Heterogeneous Hadoop systems) are all my original ideas/work and contributions during my PhD program at McMaster University. Throughout each step of my research, I received valuable consultations/ feedback/ supervision from my supervisor Dr. Down.

Contents

1 Introduction **1**
1.1 Characteristics of the Problem Domain 2
1.2 Research Motivations 4
1.3. Goals of the Research 5
1.4 Contributions of the Research 6
1.5 Structure of the Thesis 8

2 Grid Background **9**
2.1 Grid Systems 9
2.1.1 Grid Services 10
2.1.2 Grid Architecture 10
2.2 Grid Scheduling Process 12
2.3 Evaluation 13
2.3.1 Performance Metrics 13
2.3.2 Evaluation Method 14
2.4 Conclusion 15

3 State Independent Resource Management for Distributed Computational Grids **16**
3.1 Introduction 16
3.2 Computational Grid Scheduling Algorithms 18
3.3 Workload and System Model 20
3.4 Shadow Routing Algorithm 20
3.5 Grid Shadow Routing Algorithm 22
3.6 Experimental Environment 24
3.7 Experimental Results 25
3.7.1 Over and Under Estimation 25
3.7.2 Over Estimation 28

3.8 Conclusion 29

4 Reducing the Search Space and State Information in Data Grids 30
4.1 Workload Model 31
4.2 Data Grid Model 32
4.3 Proposed Scheduler 34
4.4 Computation Intensive Algorithm 35
4.5 Data Intensive Algorithm 37
4.6 Experimental Results 40
4.7 Related Work 44
4.7.1 Data Replication 45
4.7.2 Computation and Data Scheduling Interaction 46
4.7.3 Performance goals of schedulers 47
4.8 Conclusion 49

5 Hadoop Background 50
5.1 MapReduce Programming Model 51
5.2 Hadoop System 52
5.2.1 Hadoop MapReduce 53
5.2.2 Hadoop Distributed File System (HDFS) 53
5.2.3 Common 53
5.2.4 Other Hadoop projects 54
5.3 Hadoop Architecture 55
5.4 Hadoop Execution Process 57
5.5 Conclusion 61

6 COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems 62
6.1 Motivation 64
6.2 Proposed Hadoop Scheduling System 65
6.2.1 Hadoop System Model 66
6.2.2 Task Scheduling Process 68
6.3 Queuing Process 70
6.3.1 COSHH Classification 70
6.3.2 COSHH Optimization 73
6.4 Routing Process 76
6.5 Hadoop Performance Metrics 78

6.6 Experimental Results - Synthetic Workload 79
6.6.1 Experimental Environment 79
6.6.2 Compared Schedulers 81
6.6.3 Results and Analysis 82
6.7 Experimental Results - Real Hadoop Workload 85
6.7.1 Experimental Environment 86
6.7.2 Results and Analysis 88
6.8 Discussion 91
6.9 Sensitivity Analysis 93
6.10 Related Work 95
6.10.1 Simple schedulers 96
6.10.2 User Fairness based Schedulers 97
6.10.3 Formal Model based Schedulers 99
6.10.4 Job Deadline based Schedulers 100
6.11 Conclusion 101

7 Guidelines for Selecting Hadoop Schedulers based on System Heterogeneity 102
7.1 Performance Issues 103
7.1.1 Problem I: Small Jobs Starvation 103
7.1.2 Problem II: Sticky Slots 104
7.1.3 Problem III: Resource and Job Mismatch 106
7.1.4 Problem IV: Scheduling Complexity 106
7.2 Heterogeneity in Hadoop 108
7.3 Evaluation: Settings 109
7.3.1 Experimental Environment 109
7.4 Evaluation: Homogeneous Hadoop System 111
7.4.1 Case Study 1: Homogeneous-Small 111
7.4.2 Case Study 2: Homogeneous-Large 112
7.5 Evaluation: Heterogeneous Hadoop System 113
7.5.1 Case Study 3: Heterogeneous-Small 114
7.5.2 Case Study 4: Heterogeneous-Large 115
7.5.3 Case Study 5: Heterogeneous-Equal 117
7.6 Guidelines for Scheduler Selection 119
7.6.1 Homogeneous Hadoop 120
7.6.2 Heterogeneous Hadoop 121
7.7 Related Work 121

7.8 Conclusion 122

8 A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems 125

8.1 Introduction 125
8.2 Scalability in Hadoop Systems 126
8.3 Motivation 127
8.3.1 Scalable Hadoop Schedulers 127
8.3.2 Heterogeneity Aware Scheduling 127
8.3.3 Considering Critical Hadoop Performance Metrics 128
8.3.4 Avoiding Large Scheduling Overhead 128
8.3.5 Scheduling for the MapReduce model 129
8.4 Performance Issues 129
8.5 Analysis 130
8.5.1 Case Study 1: Job Number Scalability 130
8.5.2 Case Study 2: Resource Number Scalability 132
8.6 Hybrid Solution 133
8.7 Conclusion 136

9 The Effect of Minimum Shares on Performance of Hadoop Schedulers 137

9.1 Minimum share effects on Hadoop schedulers 138
9.2 Analysis 140
9.3 Experimental Results 145
9.3.1 Experimental Environment 145
9.3.2 Results 146
9.4 Discussion 148
9.5 Related Work 150
9.6 Conclusion 152

10 A Prototype System 153

10.1 Design Diagram 153
10.2 Implementation Challenges 159
10.3 Lessons Learned 160
10.4 Installing the COSHH scheduler 161

11 Discussion and Future Work 163

11.1 Applications 164
11.2 Challenges 166

11.3 Future Work 167
11.4 Relevant Publications and Submissions by the Author 169

Bibliography **169**

Chapter 1

Introduction

The topic of this thesis is improving the performance in heterogeneous Grid and Hadoop systems. Our focus of attention is how to efficiently integrate system heterogeneity in the scheduling process to provide better performance levels while considering the critical requirements and priorities of each system.

The increasing number of computational problems in various fields such as science, engineering, and business are not tractable using the current generation of high-performance computers (Luther et al. [2006]). In fact, due to their size and complexity, these problems need to work with distributed application models and components. Advanced networks, data and computing resources, and networks of clusters were consolidated into many national projects to establish wide-area computing infrastructures, known as Grid computing. Computational Grid systems could be used to solve computation-intensive large scale problems. However, the next generation of these problems were more data-intensive, where the input data was generated in geographically distributed resources. Researchers further developed the Grid computing paradigm to address these new problems, producing large-scale Data Grid systems. More recently, there has been a surge of interest to analyze massive data, thus motivating greatly increased demand for computing. Using low-cost virtualization along with the availability of commercial large-scale commodity clusters containing hundreds of thousands of computers has led to the development of Cloud computing systems. Operating at this increased scale, and analyzing increasingly large amounts of data demands fundamentally different approaches. The MapReduce programming model (Dean and Ghemawat [2008]), and its open source implementation Hadoop (Apache Hadoop Foundation [2010b]) have been introduced with a goal of providing highly efficient and scalable solutions for massive data analysis.

Scheduling is a significant issue in any distributed computing environment. A scheduling algorithm should consider multiple objectives, including managing the system based on resource

features and application requirements. An efficient scheduler should adapt to changes in the system. Based on the experiments and observations in this work, considering system heterogeneity is a significant challenge in scheduling of Grid and Hadoop systems. The choice of scheduling algorithm can highly affect the performance in term of metrics such as average completion times. There may also be additional constraints on a scheduler, imposed for various reasons (one example is guaranteed resource shares in Hadoop).

Traditional Grid and Hadoop scheduling algorithms either ignore system heterogeneity or consider it with the cost of adding significant overhead. This research proposes new scheduling algorithms for each of the Computational Grid, Data Grid, and Hadoop models. The proposed algorithms are designed to improve performance in heterogeneous environments. This chapter briefly discusses characteristics of Grid and Hadoop systems (Section 1.1). The intent and motivation behind the proposed schedulers are discussed in Section 1.2. The research goals and objectives are addressed in Section 1.3, ending with the problem statement of the thesis. Section 1.4 briefly discusses the contributions of this dissertation. Finally, the outline for the remainder of the dissertation is presented in Section 1.5.

1.1 Characteristics of the Problem Domain

Grid systems were proposed as the next generation computing platform and global infrastructure for solving large scale problems (Luther et al. [2006]). Grids allow sharing of scientific instruments, which have high cost of ownership, such as a particle accelerator (The Large Hadron Collider, CERN [2004]). Moreover, they make it possible to support on-demand and real-time processing and analysis of data generated by these scientific instruments. This capability significantly enhances the possibilities for scientific and technological research and innovation, industrial and business management, application software service delivery and commercial activities. With respect to this thesis, the important characteristics of a Grid computing environment are listed as follows:

- The scheduler is a critical part of any Grid computing system, having a direct effect on system performance. The scheduler assigns submitted jobs to the Grid resources. To achieve the goals of Grid systems, effective and efficient scheduling algorithms are fundamentally important (Dong [2009]).

- Grid schedulers are evaluated with different performance metrics such as flowtime (average completion time of all tasks), makespan (maximum completion time of all tasks), average resource utilization, and data availability (availability of data for each task).

- The Grid scheduling problem is known to be NP-complete (Abraham et al. [2000]). Several optimization criteria are considered for scheduling in Grids, making it a multi-objective problem.

- A typical Grid system includes heterogeneous resources which may be widely distributed. To achieve promising performance levels, a Grid scheduler needs to reduce the completion times and communication costs in this heterogeneous environment.

- There are different types of Grid computing systems (such as Computational Grids and Data Grids), employing different schedulers. For example, a scheduler which targets computation times is suitable for Computational Grids, while a Data Grid needs a scheduler which considers both computation and data transfer times. Due to the considerable differences between various Grid types, it is extremely difficult to define a single scheduling algorithm that performs uniformly well for all Grid systems. Therefore, a practical approach is to study and define a scheduler for each Grid system based on its specific concerns.

- To schedule a job in a Data Grid system, the scheduler needs to search through possible combinations of computational resources (for executing the job) and storage resources (for locating the job's input data) to find the appropriate combination. Moreover, for a job to be executed in a Data Grid system, its input data is transferred from the storage resource to the computational resource; therefore, the data transfer cost varies based on the selected computational and storage resources (when the data have multiple replicas). The scheduler is responsible for reducing this communication cost.

The Grid computing paradigm has many applications, particularly in scientific fields. Some notable ongoing Grid projects include: LHC Computing Grid (LCG) (Worldwide LCG Computing Grid [2012]) (a global collaboration for analysis of LHC experiments in the CERN project (The Large Hadron Collider, CERN [2004])), NEESgrid (The NEESGrid System Integration Team [2004]) (a Grid system linking earthquake researchers across the United States), and BIRN (Jovicich et al. [2005]) (Bioinformatics Information Research Network).

As a successor to Grid systems, Cloud systems and particularly Hadoop (Apache Hadoop Foundation [2010b]) clusters are gaining a lot of attention for their various applications for individuals and enterprises. The MapReduce (Dean and Ghemawat [2008]) and Hadoop frameworks were designed to support efficient large scale computations. Some of the important Hadoop characteristics for this dissertation are as follows:

- Hadoop was implemented based on the MapReduce programming model, where each job is divided into number of map and reduce tasks. A Hadoop scheduler is responsible for assigning map and reduce tasks to the available Hadoop resources.

- Hadoop schedulers can be evaluated using different metrics such as average completion time of jobs, minimum share satisfaction, fairness (among users), data locality (higher values indicate lower data transmission cost), and scheduling time (lower value leads to less scheduling overhead).
- Hadoop has been used for efficient processing of Big Data (Agrawal et al. [2011]) (massive and rapidly growing collection of data sets). Hadoop schedulers are critical elements for preserving system scalability. The scalability of Cloud infrastructures has significantly increased their applicability. Therefore, Hadoop is required to be highly scalable for providing desired performance levels independent of the system scale.
- Similar to Grid, various Hadoop systems require different schedulers, and there has been no evidence that a unique scheduler works best in all Hadoop systems. For instance, a Hadoop scheduler ignoring system heterogeneity may perform well in a homogeneous system, but result in poor performance for a heterogeneous Hadoop system.

1.2 Research Motivations

This PhD research is motivated by various problems in the scheduling process of Grid and Hadoop systems. The main motivations are listed as follows:

1. Simplicity is considered as the key feature in selecting a scheduler for Hadoop systems. There are several examples of using simple and fast schedulers in very complicated systems such as applying the Fair Sharing (Apache Hadoop Fair Scheduler [2010]) and Capacity (Apache Hadoop Capacity Scheduler [2010]) schedulers in large scale Hadoop clusters used by Facebook and Yahoo!, respectively. The main intuition behind these schedulers is to reduce the scheduling time and overhead. This results in ignoring several key performance metrics in their scheduling decisions. One of the main motivations in this thesis is how to improve the completion time as well as several other performance metrics (e.g. fairness, minimum share satisfaction, and data locality) even if it requires adding some complexity to the decision making process.
2. A transition from a homogeneous Hadoop environment to a heterogeneous environment can be observed in many applications. However, existing Hadoop schedulers are not appropriately customized for heterogeneous systems. This research was originally motivated by addressing the scheduling challenges arising from the increase in heterogeneity of distributed systems. Heterogeneity is increasing in both applications and Hadoop clusters. For instance, Facebook defines various applications including business intelligence, spam detection, and

ad optimization running on its Hadoop cluster (Zaharia et al. [2010]); the New York Times uses Hadoop to convert millions of different size articles and images into PDF files (Gottfrid [2013]), and there are also Hadoop clusters executing many other experimental jobs, ranging from multi-hour machine learning computations to 1-2 minute ad-hoc queries (Zaharia et al. [2010]). Heterogeneous systems introduce new scheduling challenges, directly affecting the system performance. A state of the art scheduler should address challenges introduced by these environments.

3. The nature of Cloud and Grid systems is dynamic, meaning that the number, size, and complexity of jobs as well as the number and types of available resources may vary over time. For example, a medium-sized company with data privacy concerns would prefer to have a private Cloud system. This Cloud system may have a large number of jobs to be scheduled on several available resources during business hours, while after business hours, less work stations are available for a smaller number of jobs. There may be no single scheduler which performs well in all conditions of these distributed computing environments. Therefore, instead of using a single scheduler, an appropriate approach would be a combination of different powerful schedulers with a smart switch to change the schedulers based on the current environmental context.

4. Grid scheduling based on gathering system state information and searching over all possible options leads to large overheads (Dong and Akl [2006]). As in the case of the CERN project (The Large Hadron Collider, CERN [2004]), resources are generally provided by different organizations which are geographically distributed (Baker et al. [2002]). To tackle the issue of system heterogeneity, Grid schedulers typically gather system state information, and search through a large space of all possible options to find an appropriate matching. Involving the state information can add huge overhead and increases the scheduling time (and consequently the completion time). However, ignoring the state information could significantly affect the performance. One of the motivations for this PhD thesis is to reduce this overhead while considering the system heterogeneity.

1.3. Goals of the Research

This PhD research considers the scheduling problem in heterogeneous Grid and Hadoop systems. As discussed in the Motivation section, several performance issues arise from ignoring heterogeneity in making scheduling decisions. As a result, performance can be improved by considering the following two issues:

- *System heterogeneity*: a scheduler which makes decisions with respect to the heterogeneity in both workload and resources can improve performance metrics such as average completion time.

- *System state information*: schedulers that reduce the required state information to be gathered from all distributed resources can improve the performance in terms of overhead and communication cost.

Most traditional schedulers in distributed computing systems either neglect these issues or consider only one of them, with a resulting large degradation in the other.

> *The goal of this research is to design schedulers for heterogeneous Grid and Cloud systems, which improve performance in terms of quantities related to the completion time of jobs for both data and computation-intensive applications. Moreover, the overhead should be negligible compared to the improvement in the performance.*

1.4 Contributions of the Research

In what follows, the key contributions of this dissertation are listed.

1. Three schedulers are introduced for Computational Grid, Data Grid, and Hadoop systems, respectively. All of the proposed schedulers are designed to reduce the average completion time by considering resource and application heterogeneity in their scheduling decisions.

    - A scheduling algorithm is introduced for *Computational Grid* systems, which simultaneously addresses several objectives namely, minimizing completion times, while requiring *zero* dynamic state information. Simulation results show the promising performance of this algorithm, and its robustness with respect to errors in parameter estimates.

    - The scheduler proposed for *Data Grid* systems makes scheduling decisions based on system heterogeneity. The introduced approach simultaneously considers reducing the search space and decreasing the required state information. The proposed scheduler adjusts its scheduling and data replication decisions dynamically based on changes in system parameters, such as arrival rates.

    - A new *Hadoop* scheduling system is introduced and implemented, named COSHH (Classification and Optimization based Scheduler for Heterogeneous Hadoop systems), which considers heterogeneity at both the application and cluster levels. The main objective of COSHH is to minimize the average completion time. However, as it is concerned with

other key Hadoop performance metrics, the proposed scheduler also achieves competitive performance under minimum share satisfaction, fairness and locality metrics (with respect to other well-known Hadoop schedulers). This approach can be efficiently applied in heterogeneous clusters, in contrast to most Hadoop scheduling systems, which assume a homogeneous Hadoop cluster.

2. A comprehensive analysis of the proposed Grid schedulers is performed in simulated Grid environments. The schedulers are evaluated using different real Grid workloads from the CERN Grid project (The Large Hadron Collider, CERN [2004]), and the Blast application suite (Altschul et al. [1990]) used by biologists to perform searches on nucleotide and protein databases. The proposed schedulers are compared with other well-known schedulers in each of the Computational Grid and Data Grid systems.

3. Several guidelines are proposed for selecting schedulers in both heterogeneous and homogeneous Hadoop systems. For this purpose, the Hadoop system is described based on three main factors: cluster, workload, and user. Each factor is either heterogeneous or homogeneous, which reflects the overall heterogeneity level of the Hadoop system. This research studies the effects of heterogeneity in each Hadoop factor on the Hadoop schedulers' performance. Three schedulers which consider different levels of Hadoop heterogeneity are used for the analysis: FIFO, Fair sharing, and COSHH. Performance problems of these Hadoop schedulers are discussed, and experiments are provided to evaluate the impact of these problems.

4. A hybrid Hadoop scheduler is introduced for dynamic systems. This thesis analyzes the performance of widely used Hadoop schedulers including FIFO and Fair sharing, and compares them with the COSHH scheduler. Based on the developed insights, a hybrid solution is introduced, which selects appropriate scheduling algorithms for scalable and heterogeneous Hadoop systems with respect to the number of incoming jobs and available resources.

5. A complete analysis of COSHH is presented on a simulated system using real Hadoop workloads from Facebook and Yahoo!. Based on these experiments, a complete analysis (in terms of system heterogeneity and scalability) is provided for COSHH.

6. For further analysis and verification of the simulation results, COSHH is implemented in a real Hadoop environment. The developed scheduler is installed on a cluster of multiple nodes. Experimental results from the real environment confirm the simulation results performed earlier. The COSHH software prototype can be used as a starting point for future implementation work.

1.5 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 provides an introduction to Grid computing systems and the corresponding scheduling processes. A scheduler for Computational Grids is proposed and evaluated in Chapter 3. A scheduler for Data Grid systems is then introduced in Chapter 4. Chapter 5 provides necessary background information on Hadoop systems. A new Hadoop scheduling system is proposed in Chapter 6. Chapter 7 presents some guidelines for selecting Hadoop schedulers based on different levels of system heterogeneity. The scalability analysis of Hadoop schedulers is provided in Chapter 8, which is followed by the introduction of a hybrid scheduler for Hadoop. The evaluation of COSHH on a real Hadoop cluster is presented in Chapter 9, including an analysis of Hadoop schedulers in different settings of minimum shares. Chapter 10 describes some implementation details and lessons learned from developing COSHH on a real Hadoop system installed on a cluster. Finally, Chapter 11 discusses the conclusions reached and outlines some potential future research directions.

Chapter 2

Grid Background

The availability of powerful computers and high speed network technologies have led to a solution for increasingly large scale problems. This solution is popularly known as Grid computing (Livny and Raman [1999]). Grid systems enable the sharing, selection, and aggregation of a wide variety of distributed resources including supercomputers, storage resources, data sources, and specialized devices. The Grid resources can be geographically distributed, owned by different organizations, and are used for solving large scale problems in science, engineering, commerce, and various other fields. The concept of Grid computing was originated as a project to link geographically dispersed supercomputers, but now it has grown far beyond its original intent. The Grid infrastructure can benefit many applications, including collaborative engineering, data exploration, high throughput computing, and distributed supercomputing. An introduction to Grid computing systems, its services, and architecture is provided in Section 2.1. The Grid scheduling process and corresponding message flows are discussed in Section 2.2. Section 2.3 presents the Grid performance evaluation process, and the last section concludes this chapter.

2.1 Grid Systems

A high level view of activities involved within a seamless, integrated computational and collaborative Grid environment is shown in Figure 2.1. In a Grid system, a task is an atomic unit to be scheduled and assigned to a resource, while an application is a set of atomic tasks that are carried out on a set of resources. The end users interact with the Grid resource broker, which performs resource discovery, scheduling, and task monitoring processes on the distributed Grid resources. In order to provide users with a seamless computing environment, Grid middleware needs to address several inherent challenges (Livny and Raman [1999]). One of the main challenges is heterogeneity in Grid environments, which results from the multiplicity of heterogeneous resources and the vast range of encompassed technologies. Another challenge involves autonomy issues due to geo-

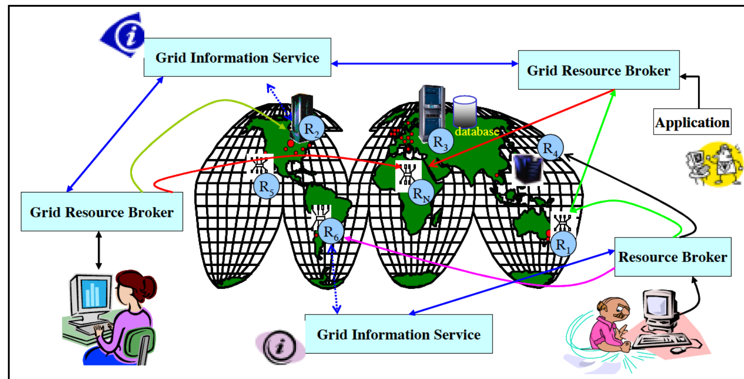


Figure 2.1. A world wide Grid computing environment (Baker et al. [2002]).

graphically distributed Grid resources across multiple administrative domains owned by different organizations. Other challenges include scalability (potential performance degradation as the size of a Grid increases) and dynamicity/ adaptability (volatility of resources). The Grid middleware must dynamically adapt and use available resources and services efficiently and effectively.

2.1.1 Grid Services

From the end-user point of view, Grid systems can be used to provide different types of services. The most general services are listed as follows:

- Computational services: provide (often secure) services for executing large scale computation intensive tasks on distributed computational resources (individually or collectively). A Grid providing computational services is often called a Computational Grid. Some examples of computational Grids are: NASA IPG (Johnston et al. [1999]), and the World Wide Grid (Buyya [2001]).

- Data services: are concerned with providing and managing secure access to distributed datasets. To provide scalable storage and access, data sets may be replicated, catalogued, and even stored in different locations to create an illusion of mass storage. The processing of datasets is carried out using Computational Grid services, resulting in a combination commonly called a Data Grid. Sample applications of these services are in high-energy physics (Hoschek et al. [2000]) and the use of distributed chemical databases for drug design (Buyya et al. [2003]).

2.1.2 Grid Architecture

Figure 2.2 shows the hardware and software stack within a typical Grid architecture. It consists of four layers: fabric, core middleware, user level middleware, and applications and portals layers.

The Grid fabric level consists of distributed resources such as computers, networks, storage devices and scientific instruments. The computational resources represent multiple architectures such as clusters, supercomputers, servers and ordinary PCs, running a variety of operating systems (such as UNIX variants or Windows). Scientific instruments such as telescopes and sensor networks provide real-time data that can be transmitted directly to computational resources or stored in a database. Core Grid middleware offers services such as remote process management, co-allocation of resources, storage access, information registration and discovery, security, and aspects of Quality of Service (QoS) such as resource reservation and trading. The Core Grid services abstract the complexity and heterogeneity of the fabric level by providing a consistent method for accessing distributed resources.

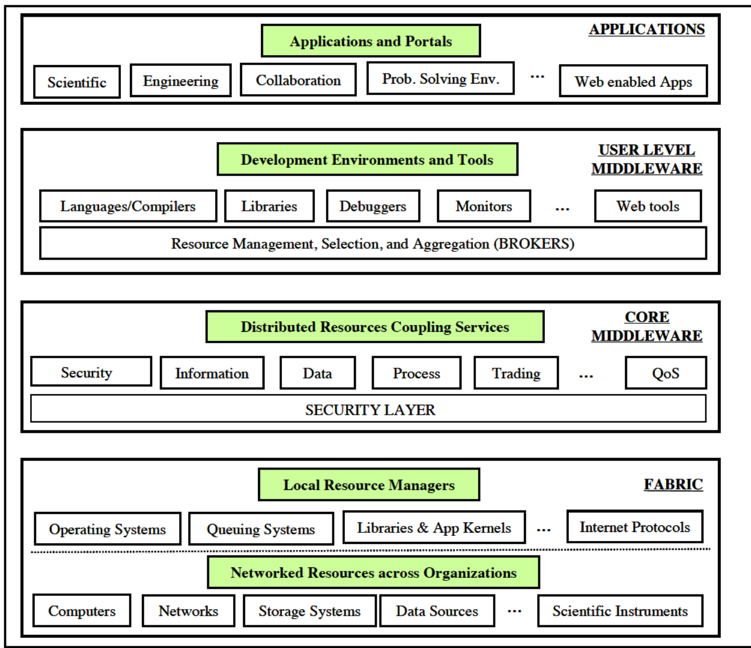


Figure 2.2. A Layered Grid Architecture and components (Baker et al. [2002]).

User level Grid middleware utilizes the interfaces provided by the low level middleware to define higher level abstractions and services. The services of user level middleware include application development environments, programming tools and resource brokers for managing resources and scheduling tasks on Grid resources. Grid applications and portals are typically developed using Grid enabled languages and utilities such as HPC++ (Gannon et al. [2009]) or MPI (Denis et al. [2001]). An example application, such as parameter simulation or a grand-challenge problem, would require computational power, access to remote data sets, and may need to interact with scientific instruments. Grid portals offer web-enabled application services, where users can submit and collect results for their tasks on remote resources through the Web.

One motivation of Grid computing is to aggregate the power of widely distributed resources, and provide non-trivial services to users. To achieve this goal, an efficient Grid scheduling system is an essential component of the user level Grid middleware.

2.2 Grid Scheduling Process

Generally, the scheduling process can be divided into three stages: (i) resource discovery and filtering, (ii) resource selection and scheduling according to certain objectives, and (iii) task submission (Shan et al. [2003]). Figure 2.3 presents a model for Grid scheduling in which functional components are connected by two types of data flow: resource or application information flow and task or task scheduling command flow. A Grid scheduler receives tasks from the users, selects feasible and available resources for these tasks according to acquired information from the Grid Information Service (GIS) module, and finally generates task-to-resource mappings based on certain objective functions and predicted resource performance. Figure 2.3 shows one Grid scheduler, while in practice multiple such schedulers might be deployed, and organized to form different structures (centralized, hierarchical and decentralized (Li [2005])) according to different concerns, such as performance and/or scalability.

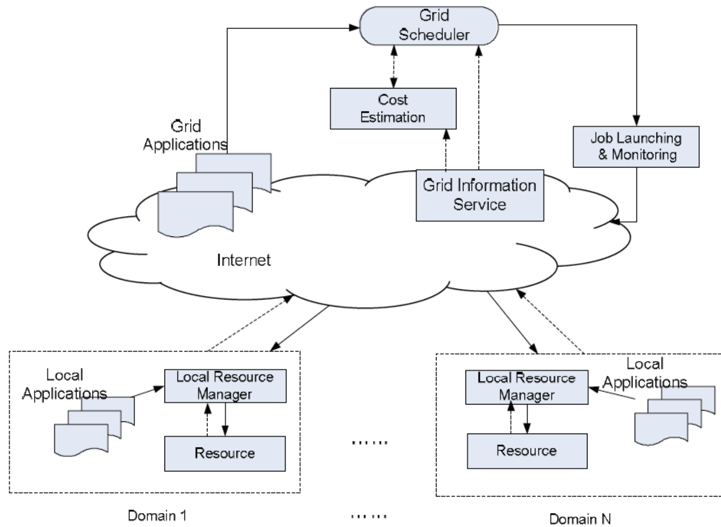


Figure 2.3. A logical Grid scheduling architecture: broken lines show resource or application information flows and solid lines show task or task scheduling command flows (Dong and Akl [2006]).

The status information of available resources is an important factor to make appropriate scheduling decisions; in particular when the heterogeneous and dynamic nature of a Grid system is taken into account. The GIS is responsible to provide such information for the Grid schedulers. It

collects and predicts resource state information, such as CPU capacities, memory size, network bandwidth, software availabilities and resource load. To make a scheduling decision, in addition to the raw resource information from the GIS, the task properties (e.g., approximate instruction quantity, memory and storage requirements, task dependencies, and communication volumes), as well as resource performance for different tasks are also required. The cost estimation module computes the cost of all candidate resource and task matchings. The computed costs are used by the scheduler to select the matchings which optimize the desired performance metrics. The Launching and Monitoring module implements a scheduling decision by submitting tasks to selected resources, staging input data and executables if necessary, and monitoring the execution of the tasks. A Local Resource Manager (LRM) is mainly responsible for two tasks: local scheduling inside a resource domain for the tasks submitted from both the exterior and domain's local users, and reporting resource information to the GIS. Within a domain, one or multiple local schedulers run with locally specified resource management policies. A LRM also collects local resource information using tools such as Network Weather Service (Swany and Wolski [2002]), and Hawkeye (Zhang et al. [2003]), and reports the results to the GIS.

2.3 Evaluation

This section introduces typical Grid performance metrics, and the methods and toolkits used to evaluate Grid scheduling algorithms.

2.3.1 Performance Metrics

There are a number of key performance metrics, the importance of which depends on the Grid system and the applications being executed. The metrics are defined as follows, where the first two metrics are used for Computational Grid algorithms, and Data Grid scheduling algorithms are evaluated using all three metrics.

- *Makespan* : the maximum completion time of all tasks.
- *Flowtime* : the average completion time of all tasks.
- *Data availability* : measures the task data access time for each resource. More precisely, it measures how much time a task requires to obtain a unit of data. This measure will tell us how close the computing resource is to the resource which stores the required data. Let $a_{l,j}$ denote the availability of data for task l when its computation resource is j. It is computed as:

$$a_{l,j} = \frac{t_{l,j}}{d_l}$$

where d_l is the amount of data required by task l (e.g. in MBytes), and $t_{l,j}$ is the time that task l needs to gather all its required data from the selected storage resources to the computational resource j (e.g. in seconds). The quality of a scheduling algorithm in choosing a resource for executing data intensive jobs can be defined based on the average data availability over all tasks and resources. This can be written as

$$\bar{a} = \frac{\sum_{j=1}^{M} \sum_{l=1}^{n^t} a_{l,j}}{n^t},$$

where n^t is the total number of tasks executed, and M is the total number of resources in the system.

2.3.2 Evaluation Method

A full scale evaluation on a Grid testbed can require significant interference with production workloads. In addition, it is difficult to evaluate new replication strategies and scheduling algorithms in a repeatable and controlled manner. Therefore, another method is required for testing Grid scheduling algorithms over a range of complex Grid systems and workloads.

Simulation appears to be a feasible way to analyze algorithms on large scale distributed systems of heterogenous resources. Unlike using the real system in real time, simulation avoids the overhead of coordination of real resources; therefore, it does not add unnecessary complexity to the analysis mechanism. Simulation is also effective in working with very large problems that would otherwise require the involvement of a large number of active users and resources.

This need has led to the development of various Grid simulators. Among the available simulators, GridSim (Buyya and Murshed [2002]) was selected for this research, because of its complete set of features for simulating realistic Grid testbeds. Some of the main features of GridSim include: modelling heterogeneous computational resources, scheduling tasks based on time or space shared policies, differentiated network service, and simulation of workload traces from real systems. Moreover, GridSim allows incorporating new components into its existing infrastructure.

GridSim has been extended in (Sulistio et al. [2008]) to handle features which specifically target Data Grid environments: (1) replication of data to several resources; (2) queries for location of replicated data; (3) access to replicated data, and (4) complex queries about data attributes. This simulator facilitates integrated studies of novel and on-demand data replication strategies and task scheduling approaches.

2.4 Conclusion

This chapter provided an introduction to Grid systems, architecture, performance metrics, and the associated scheduling process. In the next two chapters new scheduling algorithms are proposed for different types of Grid systems. The proposed schedulers will be an extension to the user level middleware layer of the Grid architecture.

Chapter 3

State Independent Resource Management for Distributed Computational Grids [1]

In Computational Grid systems, there are typically two kinds of objectives. The first is the system performance in terms of quantities related to the completion time of tasks. The second is the amount of required state information, which is often measured in terms of quantities such as communication costs. These two objectives are often in tension with one another. For example, gathering large amounts of state information can lead to low completion times. In this chapter, a scheduling algorithm is introduced, which simultaneously addresses the objectives listed above namely, minimizing completion times, while requiring *zero* dynamic state information. The evaluation demonstrates promising performance of the proposed algorithm, and its robustness with respect to errors in parameter estimates.

3.1 Introduction

Innovations in computational and network technologies have led to the emergence of computational Grid systems (Livny and Raman [1999]). Task scheduling is an integral part of a distributed computing system. The scheduling algorithms involve matching of task needs with resource availability. Several optimization criteria are considered for scheduling in Grids, making the problem a multi-objective one in its general formulation. Grid scheduling is an NP-complete optimization problem targeting several criteria (Abraham et al. [2000]).

[1]This chapter is mostly based on the paper: A. Rasooli and D. G. Down, *State Independent Resource Management for Distributed Grids*, **Proceeding of the 6th International Conference on Software and Data Technologies (ICSOFT 2011)**, July 18-21, 2011, Seville, Spain.

In recent years, several analogies from natural and social systems have been leveraged to form powerful heuristics for Grid scheduling. These heuristics have proven to be successful in attacking several NP-hard global optimization problems (Abraham et al. [2000]). These scheduling policies may use different types and amounts of system information to make reasonable scheduling decisions; some parameters are typically periodically estimated (e.g., resource execution rates), and some are highly dynamic in nature, needing to be gathered in real-time (e.g., current resource loads).

To the best of our knowledge there is no single Grid scheduling algorithm which is effective over all Grid systems with their different applications and features. This chapter addresses the scheduling problem of Grid systems whose resources are widely distributed, resulting in a considerable communication cost between the resources. The most well known application of these Grid systems is in the Enabling Grids for E-sciencE (EGEE) (Erwin and Jones [2009]) project, which aims to provide a Grid platform as a service to the broader e-science community. The main contributions in this chapter are:

- A theoretical idea is taken from the literature (the so-called Shadow Routing algorithm (Stolyar and Tezcan [2009])) to develop a practical scheduling algorithm for Grid systems.
- This basic theoretical approach is modified to be efficient for Grid systems, and the advantages of the proposed algorithm for widely distributed Grid systems are discussed.

In general, Grid scheduling algorithms aim to improve the system performance, which can be evaluated by various criteria such as flowtime or makespan. Furthermore, if an algorithm reduces the amount of state information required at the time of scheduling, it leads to reductions in the communication cost and synchronization overhead.

The Shadow Routing method is a robust, generic scheme, introduced in (Stolyar and Tezcan [2009]) for routing arriving tasks in systems of parallel queues with flexible, many-server pools. This algorithm has proven to achieve good performance levels in queuing systems. However, it needs to be customized and adjusted to be applicable in Grid systems. This research modifies the structure of the Shadow Routing approach, and introduces a scheduling algorithm for Grid systems, called the Grid Shadow Routing algorithm. First, the structure of the basic Shadow Routing algorithm is changed to be applicable for a typical Grid workload model (Iosup et al. [2006]). Second, the algorithm is extended in a way that leads to significant improvement of its performance in Grid systems.

The Grid Shadow Routing algorithm defines virtual queues to keep track of the loads on resources. These virtual queues are estimates of the actual queue lengths, and remove the need for gathering real-time load information. The only information required by the proposed algorithm

is estimates of task lengths and resource execution rates. These two parameters are used in most Grid scheduling algorithms, and various prediction methods have been introduced (Zhang et al. [2006], Lu et al. [2004]). An important advantage of the Grid Shadow Routing algorithm is that it does not require highly accurate estimates to provide efficient scheduling results.

The scheduling algorithms can be classified as either static or dynamic, based on their required information and the timing of scheduling decisions. Static scheduling algorithms do not use any dynamic state information, but there can be a huge performance degradation in comparison to dynamic algorithms. On the other hand, dynamic algorithms can make better scheduling decisions, while increasing the communication cost. Therefore, there is a trade-off in algorithm selection. If a system has low communication overhead, a dynamic algorithm with full state information can significantly improve performance. On the other hand, widely distributed systems may require a significant amount of time to gather full state information (this chapter targets these systems). In such systems, a dynamic scheduling algorithm can potentially require significant overhead and have resulting performance degradation. The Grid Shadow Routing algorithm requires *zero* state information from resources, yet it can achieve better performance than commonly used dynamic algorithms that use full state information. This is particularly advantageous for large, highly loaded systems with widely distributed resources, where communication costs are significant.

The Grid Shadow Routing algorithm is evaluated using simulation, and compared with two well known Grid scheduling algorithms: Minimum Completion Time (MCT) and Join the Shortest Queue (Braun et al. [2001]). The former is a dynamic algorithm, which greedily aims to reduce the average completion time without considering the communication overhead. The latter algorithm uses partial state information, and does not require estimation of task lengths for its scheduling decisions. To analyze the sensitivity of the proposed algorithm to accuracy of the estimated parameters, it is evaluated in a system with various levels of error in its estimates.

The remainder of this chapter is organized as follows. Section 3.2 provides an overview of several Grid scheduling algorithms. The task scheduling problem and workload model are introduced in Section 3.3. Sections 3.4 and 3.5 propose the Shadow Routing algorithm and details of the proposed scheduling algorithm, respectively. In Section 3.6 the evaluation environment is described. The experimental results are provided for various Grid systems with various error models for parameter estimates in Section 3.7. Finally, some concluding remarks are given in the last section.

3.2 Computational Grid Scheduling Algorithms

A large number of algorithms have been proposed to schedule independent tasks on Computational Grid resources. This section presents several of these.

Opportunistic Load Balancing (OLB) (Armstrong et al. [1998]) assigns each task in a random order to the next available resource without considering the task's expected execution time on that resource. The major advantage of this approach is its simplicity, but its performance is often far from optimal (Fidanova and Durchova [2005]).

Minimum Execution Time (MET) (Braun et al. [2001]) assigns each task to the resource with the smallest expected execution time for that task. While assigning the fastest resource to each task, this algorithm does not consider the current load of each resource. This can cause a severe load imbalance among resources. The advantage of this algorithm is that it does not require any state information.

K-Percent Best (KPB) (Maheswaran et al. [1999]) uses the same approach as the MCT algorithm, but instead of searching for the minimum completion time among all resources, it only examines a subset of the resources; thus, it reduces the communication costs. This subset consists of a percentage ($KM/100$) of all the resources with the smallest execution times for the incoming task, where $100/M \leq K \leq 100$. Although the KPB policy can reduce the required state information, it may cause severe performance degradation if K is not chosen correctly. This algorithm considers the same number of resources for all types of tasks, which may not be desirable.

Linear Programming Based Affinity Scheduling (LPAS) is an algorithm first introduced in (Al-Azzoni and Down [2008b]), as a mapping heuristic for Heterogeneous Computing Systems. LPAS_DG (Al-Azzoni and Down [2008a]) adapts this algorithm for Desktop Grid systems. The algorithm uses an optimization approach to find the best set of candidate resources. It aims to combine the advantages of the MCT and MET algorithms, in the spirit of the KPB algorithm. The LPAS algorithm simultaneously reduces the state information and average completion time of tasks, but it requires the arrival rates and mean execution times for each class of tasks on each resource.

The following are two well-known algorithms used as benchmarks for evaluating the proposed algorithm.

- *Minimum Completion Time (MCT)*: assigns each task to the resource with the minimum expected completion time for that task (Dong and Akl [2006]). The expected completion time for an arriving task is computed at each resource, and is sent to the scheduler. The scheduler selects the resource with the minimum expected completion time for the incoming task. This algorithm has the cost of requiring full state information, and consequently may have a large communication cost. The MCT algorithm requires the following parameters: 1) estimated length of the incoming task, 2) current estimated resource execution rate (considering the fluctuations in the execution rates of the resources), 3) estimated currently available bandwidth between resources as well as between resources and the scheduler, and 4)

real-time load on each resource. The last three parameters are collected from each resource at the time of each scheduling decision.

- *Join the Shortest Queue* (JSQ*)*: assigns each task to the resource with the minimum number of waiting tasks in its queue. The advantage of this approach is that it does not require the task length for its scheduling decision. It only requires one parameter: the real-time number of tasks in each resource queue. This parameter is collected from all of the resources at the time of each scheduling decision.

3.3 Workload and System Model

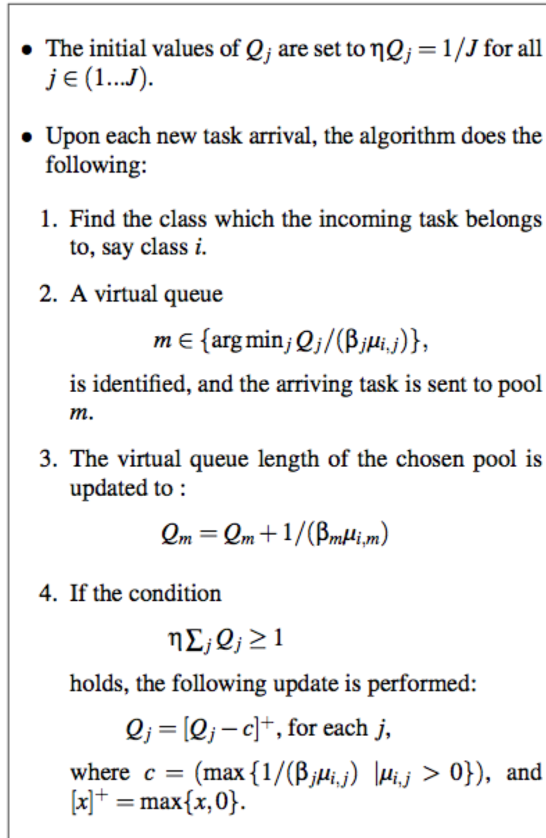
This section defines a workload model based on a typical Grid workload (Iosup et al. [2006]). Let the number of resources in the system be M. The actual resource execution rate for resource r is given by μ_r, and the length of task k is defined by L_k. Typically, existing Grid scheduling algorithms assume that estimates of task lengths and resource execution rates are available at the time of scheduling. The defined workload model uses one of the available estimation methods to provide estimates of resource execution rates for all resources and the length of each incoming task. The estimated length of task k is defined as $\hat{L}_k$, and the estimated execution rate of resource r is denoted by $\hat{\mu}_r$.

To model a widely distributed Grid system, the network is defined to have associated delays. Delay in the network is calculated based on the bandwidth and the load on the Grid network. When a task arrives, the Grid scheduling algorithm is used to route the arriving task to an available resource. It is assumed that all local schedulers are using the classical FIFO algorithm; however, in general each resource can use its own local scheduling algorithm.

3.4 Shadow Routing Algorithm

The Shadow Routing algorithm was first introduced in (Stolyar and Tezcan [2009]) as a routing algorithm for parallel queues in a virtual queueing system. It is a generic routing algorithm that appropriately balances the loads, and automatically identifies the best set of matchings without requiring the flow input rates, or explicitly solving any optimization problem.

A brief description of the model used in the basic Shadow Routing algorithm is provided here; more details of this model are presented in (Stolyar and Tezcan [2009]). The queuing model is defined as follows: it is assumed to have I classes (types) of tasks, and J pools of resources in the system, where the resources in each pool are homogeneous. The mean execution time of a task in class i running on a resource of pool j is given by $\mu_{i,j} \geq 0$ (if $\mu_{i,j} = 0$, tasks of class i can not be executed on resources of pool j).



- The initial values of Q_j are set to $\eta Q_j = 1/J$ for all $j \in (1...J)$.
- Upon each new task arrival, the algorithm does the following:

1. Find the class which the incoming task belongs to, say class i.
2. A virtual queue
$$m \in \{\arg\min_j Q_j/(\beta_j \mu_{i,j})\},$$
is identified, and the arriving task is sent to pool m.
3. The virtual queue length of the chosen pool is updated to :
$$Q_m = Q_m + 1/(\beta_m \mu_{i,m})$$
4. If the condition
$$\eta \sum_j Q_j \geq 1$$
holds, the following update is performed:
$$Q_j = [Q_j - c]^+, \text{ for each } j,$$
where $c = (\max\{1/(\beta_j \mu_{i,j}) \ |\mu_{i,j} > 0\})$, and $[x]^+ = \max\{x, 0\}$.

Figure 3.1. The Basic Shadow Routing Algorithm (Stolyar and Tezcan [2009]).

According to (Stolyar and Tezcan [2009]), in a heavily loaded queuing system, if the routing algorithm chooses only certain matchings of tasks to resource pools, it can keep task queues stable and provide asymptotically optimal performance. To be more precise, if the number of resources in all pools and the input rates of tasks scale up simultaneously by a factor r (which grows to infinity), the Shadow Routing algorithm keeps the load of the queueing system within $O(\sqrt{r})$ of its optimal capacity.

Figure 3.1 presents the basic Shadow Routing algorithm. This algorithm maintains a virtual (shadow) queue Q_j for each resource pool j - the virtual queues are used to keep loads balanced. The algorithm makes each routing decision based on the values and simple updates of virtual queues; virtual here means that the queues are simply variables maintained by the algorithm. The parameter $\eta > 0$ is a small number (defined later), which controls the tradeoff between the algorithm's responsiveness and its accuracy.

Upon the arrival of a task, its class is determined, and the ratio of the length of the virtual queue to the execution rate of that task on each resource pool is computed. Then, the resource pool with the smallest ratio is selected. The algorithm keeps track of the load in each pool by

using virtual queues. In fact, the length of the virtual queue provides an estimate of the (relative) time that the pool will be busy with executing previously assigned tasks. The algorithm trades off selecting a pool with a small (virtual) queue length versus a fast execution rate. After selecting a pool for the arriving task, the mean execution time of the task on the selected pool is added to the virtual queue of that pool. The increased load on faster resource pools may reach a point where the appropriate load balancing is impossible. At this point, the total virtual queue length of all pools reaches a predefined limit. Consequently, the algorithm reduces the virtual queue lengths of all pools. In this way, the virtual queue lengths of slower resource pools become shorter, and the chance of choosing slow pools for executing future tasks increases.

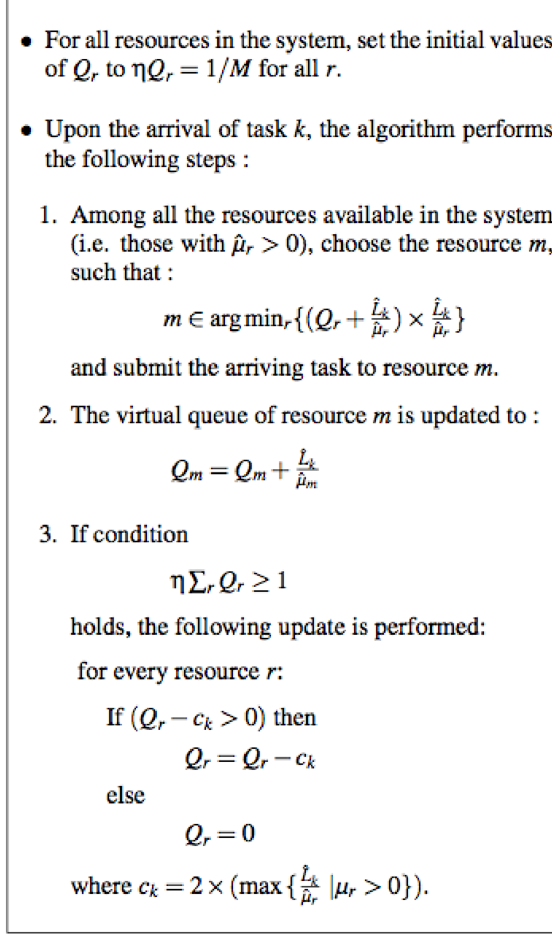
The advantage of the Shadow Routing algorithm is its appropriate load balancing without requiring any state information. However, as discussed before, the original shadow routing algorithm has limitations in the Grid systems. The following section extends this algorithm, and introduces a new scheduling algorithm for Grid systems.

3.5 Grid Shadow Routing Algorithm

The proposed algorithm, called the *Grid Shadow Routing algorithm (Grid Shadow)*, is introduced in Figure 3.2. Instead of using the class based model, the algorithm is based on the typical Grid workload model defined in Section 3.3. There are various estimation methods introduced in the literature for the estimates of task lengths and resource execution rates in Grids (see (Zhang et al. [2006], Lu et al. [2004]) for example). Rather than going into detail of any particular estimation method, it is assumed that such estimates have been provided, with associated possible errors. Therefore, the required parameters of the Grid Shadow Routing algorithm are: 1) estimated incoming task length, and 2) estimated resource execution rate. Consequently, the expected execution time of task k on resource r is calculated by $\frac{\hat{L}_k}{\hat{\mu}_r}$.

In Grid scheduling, it is important to consider the load that any incoming task may add to each resource; particularly, when the resources are heterogeneous, and the system load is moderate or light. Therefore, here the first step of the basic Shadow Routing algorithm is modified to include the expected load of the incoming task on each resource. Instead of comparing the current loads, the proposed algorithm considers the size of the virtual queue plus the expected load of the incoming task on the corresponding resource. Another way to look at this is from the analytic perspective, if the load on the system approaches 1 as in (Stolyar and Tezcan [2009]), then the effect of the incoming task is negligible. This may not be true in practice, and should be accounted for.

For each incoming task, the Grid Shadow Routing algorithm aims to increase the virtual queues by the minimum possible amount. As a result, the normalization step will be triggered less



- For all resources in the system, set the initial values of Q_r to $\eta Q_r = 1/M$ for all r.
- Upon the arrival of task k, the algorithm performs the following steps :

1. Among all the resources available in the system (i.e. those with $\hat{\mu}_r > 0$), choose the resource m, such that :

$$m \in \arg\min_r \{(Q_r + \frac{\hat{L}_k}{\hat{\mu}_r}) \times \frac{\hat{L}_k}{\hat{\mu}_r}\}$$

and submit the arriving task to resource m.

2. The virtual queue of resource m is updated to :

$$Q_m = Q_m + \frac{\hat{L}_k}{\hat{\mu}_m}$$

3. If condition

$$\eta \sum_r Q_r \geq 1$$

holds, the following update is performed:

for every resource r:

If $(Q_r - c_k > 0)$ then

$$Q_r = Q_r - c_k$$

else

$$Q_r = 0$$

where $c_k = 2 \times (\max\{\frac{\hat{L}_k}{\hat{\mu}_r} | \mu_r > 0\})$.

Figure 3.2. The Grid Shadow Routing Algorithm

frequently. This results in less overhead due to scheduling. Typically, Grid systems have a large number of resources, where searching over all resources and updating all virtual queues (as in step 3 of the basic Shadow Routing algorithm) may cause a large overhead. The proposed modification reduces the number of times that step 3 is triggered, and leads to improving the performance.

The introduced algorithm considers three factors for selecting a resource for each incoming task: the current load of all resources, the estimated execution time of the arriving task on each resource, and the load that the incoming task adds to each resource. In the first step, the proposed algorithm compares the quantity $((Q_r + \frac{\hat{L}_k}{\hat{\mu}_r}) \times \frac{\hat{L}_k}{\hat{\mu}_r})$ for all resources, and selects the one with the smallest value. There is a trade-off between selecting a resource with the earliest completion time for the currently assigned tasks versus a resource with the fastest execution for the incoming task. Moreover, the algorithm aims to minimize the load which will be added to each resource. When a resource is selected, the estimated execution time of the incoming task on the selected resource is added to the virtual queue of the corresponding resource, which is given by $\frac{\hat{L}_k}{\hat{\mu}_m}$ for task k on

resource m. There is a normalization step in the proposed algorithm, in which the virtual queues are reduced by the maximum load that the incoming task can add to the resources. In this way, the virtual queue lengths of the slower resources will be smaller, and consequently, their chance to be selected for executing future incoming tasks is increased.

The parameter η in the Grid Shadow Routing algorithm is system dependent. If η is a large number, the algorithm aims to equally divide the loads, which reduces the impact of resource heterogeneity on the scheduling decisions. Therefore, η should be a small number, and the smaller its value, the more accurate matching of resources to tasks would be. This leads to appropriate load balancing based on both the load and speed of each resource. However, the rate at which the algorithm adapts to changes in the system parameters is proportional to η (the smaller η, the slower the rate) and thus η should not be chosen too small. This parameter can be chosen by running a few experiments, and tracking how fast the normalization step (step 3 in the algorithm) is triggered. If step 3 is triggered each time that a virtual queue is updated, we need to reduce η. On the other hand, if the load for one resource is quickly increasing, and multiple subsequent jobs are assigned to that resource, while other resources are free, we need to increase η to trigger step 3 more often. For the workloads used in this work, a good value of η was determined to be 1/300. As the value of η just affects the number of times that the normalization step is triggered, the algorithm's decisions are not too sensitive to small changes in η.

As the Grid Shadow Routing algorithm is based on the lengths of the virtual queues, if the task input rates and/or resource execution rates change, no explicit detection of such an event (or any other input rate measurement/estimation) is necessary. The virtual queues automatically re-adjust and the algorithm adapts its decisions accordingly.

3.6 Experimental Environment

The experiments were run on a simulated Grid system consisting of 50 dedicated resources with different CPU speeds. In a widely distributed Grid system, there is typically low bandwidth between system elements located far from each other. Therefore, in our simulated environment, the bandwidth inside the elements is set to be 1 Gbps, and the bandwidth between the scheduler and the 50 resources is defined to be 10 Mbps. The GridSim simulator calculates the network delay between any two elements of the system using the bandwidth and load on each network at any given time.

As the proposed algorithm is mostly advantageous for EGEE Grids, it is evaluated in a real workload from the CERN Grid project (The Large Hadron Collider, CERN [2004]). The evaluation workload is collected from the LCG project, where the LCG testbed represents the Large Hadron Collider (LHC) Computing Grid. The LCG trace version 0.1 is used, which is provided by the

Grid Workloads Archive (Iosup et al. [2006]) in a typical Grid Workloads Format (GWF). Each simulation is run until the first $20,000$ tasks of the trace arrive; the arrival stream is then turned off, and the experiment continues until the system empties.

The Grid Shadow Routing algorithm uses estimates of the task lengths and resource execution rates. However, various estimation methods may have different levels of accuracy. Therefore, the algorithm is evaluated in a system with various levels of error in the estimated task lengths and resource execution rates. To study the robustness of the algorithm, it is evaluated in cases with 0% to 40% error in the estimates; however, typically these errors are on the order of 10% (Akioka and Muraoka [2004]). An error model is defined based on the model discussed in (Iosup et al. [2008]) for estimating task lengths and resource execution rates. Generally the two models of error in these estimates are:

- *Over and Under Estimation Error.* Consider actual task lengths and resource execution rates to be L_k for task k and μ_r for resource r, respectively. Let $\hat{L}_k$ denote the (corresponding) estimated task length, and $\hat{\mu}_r$ denote the estimated resource execution rate. In the simulations, $\hat{L}_k$ and $\hat{\mu}_r$ are obtained using the following relations: $\hat{L}_k = L_k \times (1 + E_k)$ and $\hat{\mu}_r = \mu_r \times (1 + E_r)$. Here, E_k and E_r are the errors for task lengths and resource execution rates, respectively, which are sampled from the uniform distribution $[-I, +I]$, where I is the maximum error.

- *Over Estimation Error.* This error model is obtained using the relations $\hat{L}_k = L_k \times (1 + E'_k)$ and $\hat{\mu}_r = \mu_r \times (1 + E'_r)$. The variables E'_k and E'_r are the errors for task lengths and resource execution rates, respectively. These errors are sampled from the uniform distribution $[0, +I]$, where I is the maximum error. This model is used for systems which always over estimate the parameters (resource powers are estimated to be the maximum amount without considering fluctuations in their power caused by increase of the load, and task lengths are always conservative).

3.7 Experimental Results

The Grid Shadow Routing algorithm is evaluated for both parameter estimation error models.

3.7.1 Over and Under Estimation

The experiments in this section apply the over and under estimation error model for the parameter estimates. First, the algorithms are evaluated in an environment with accurate resource execution rates, and errors in estimating task lengths. The results are provided in Figures 3.3 and 3.4, from the flowtime and makespan perspectives, respectively. Then, the experiments are

run in an environment with errors in estimating both task lengths and resource execution rates. The results of these experiments are provided in Figures 3.5 and 3.6. A range of error levels are presented in each of the figures.

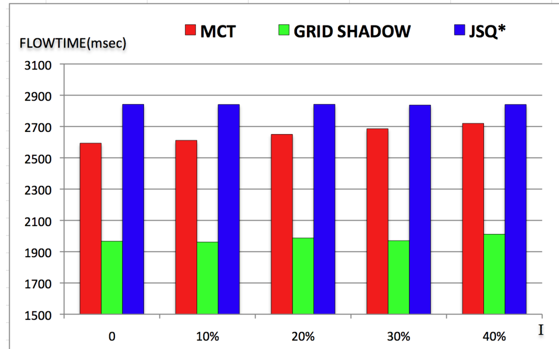


Figure 3.3. Flowtime-over & under estimating task length

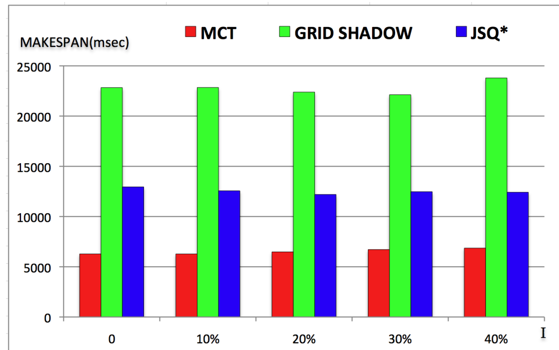


Figure 3.4. Makespan-over & under estimating task length

In these results, the MCT algorithm is the only algorithm that uses full state information in making scheduling decisions. Therefore, it is expected that in the absence of overhead, this algorithm achieves the smallest makespan and flowtime, and leads to a good balance between the loads on the resources. As the simulations consider a highly loaded system, in which gathering full state information causes large overhead, the MCT algorithm results in poor flowtime performance compared to the Grid Shadow Routing algorithm. However, the MCT algorithm achieves the best makespan performance by reducing the completion time for each incoming task.

In general, minimizing flowtime can happen at the expense of increasing the largest task's completion time. On the other hand, minimizing the makespan aims to reduce the completion time of each individual task; however, in practice this generally leads to increasing the completion times of most tasks. In summary, reducing the makespan may result in increasing the flowtime. By considering this issue, and the greedy approach of the MCT algorithm in minimizing the

completion time for each individual task, this algorithm can lead to poor performance in terms of the average completion time of all tasks. The poor performance of the MCT algorithm in terms of flowtime results from its high overhead and greedy approach.

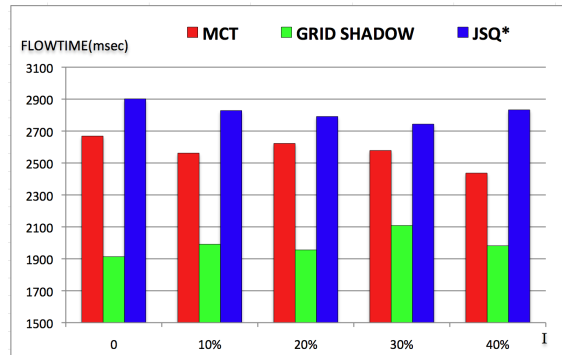


Figure 3.5. Flowtime-over & under estimating task length and resource rate

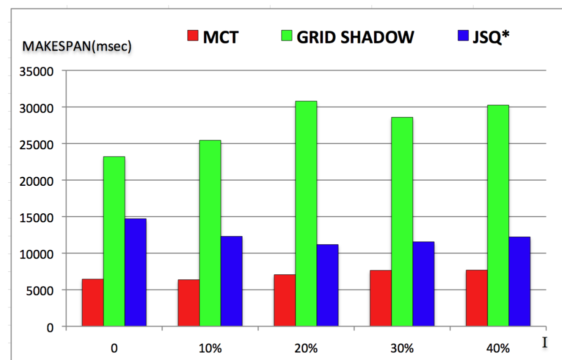


Figure 3.6. Makespan-over & under estimating task length & resource rate

These results suggest that the Grid Shadow Routing algorithm can yield significant improvement of the flowtime compared to the MCT algorithm. The use of zero state information in the Grid Shadow Routing algorithm might lead one to believe that it leads to poor performance compared to the MCT algorithm. However, the fact that the algorithm is not greedy, along with its long term approach in minimizing completion times and balancing loads, leads to improving aggregated metrics such as flowtime. Moreover, unlike the MCT algorithm, the Grid Shadow Routing algorithm has no overhead due to gathering state information. As the Grid Shadow Routing algorithm considers overall balancing of loads, and does not concentrate on minimizing the completion time of individual tasks, it can increase the completion time for a small number of tasks, which results in larger makespans.

The proposed algorithm is most useful for EGEE-like Grid systems in which gathering full state information for scheduling each incoming task causes significant overhead for the system.

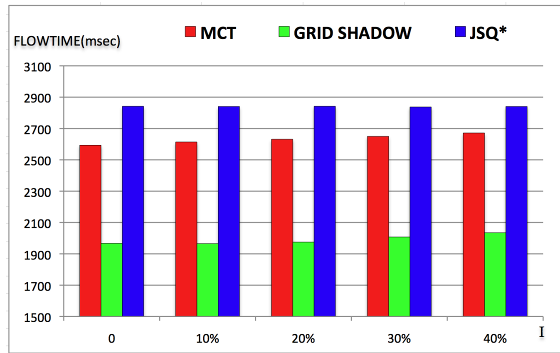


Figure 3.7. Flowtime-over estimating task length

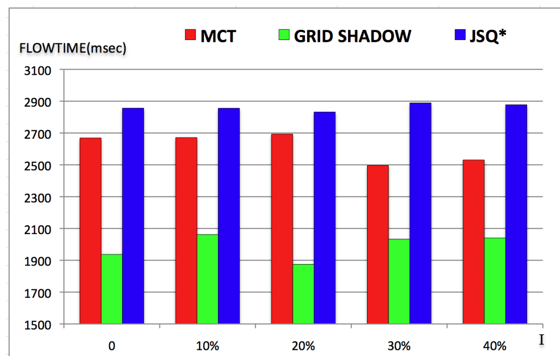


Figure 3.8. Flowtime-over estimating task length and resource rate

In these Grid systems the average completion time (flowtime), interpreted as QoS (Maheswaran et al. [1999]), is generally more important than the maximum completion time of individual tasks (makespan), interpreted as throughput of the system. According to the results, even in systems which have large estimation errors, the Grid Shadow Routing algorithm still has much better flowtime than the MCT algorithm.

3.7.2 Over Estimation

In this section, the algorithms are first evaluated in an environment with accurate resource execution rates, but various error levels in estimating task length. Figures 3.7 and 3.9 present the results from the flowtime and makespan perspectives, respectively. Then, the environment is set to include errors in both task length and resource execution rate estimates. Figures 3.8 and 3.10 compare the flowtime and makespan of the scheduling algorithms in an environment with errors in all estimates. Different error levels are considered in these figures. The results show that the Grid Shadow Routing algorithm can tolerate up to 40 percent over estimation, without significant degradation in its performance.

To summarize the observations in this section, in a real Grid workload, the Grid Shadow

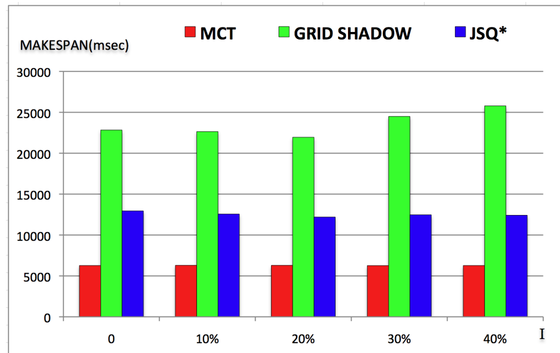


Figure 3.9. Makespan-over estimating task length

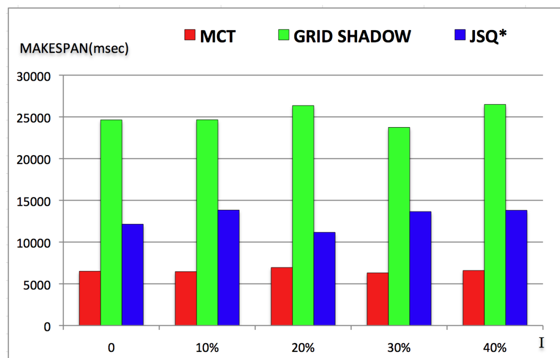


Figure 3.10. Makespan-over estimating task length and resource rate

Routing algorithm has significantly better flowtime than the MCT algorithm, and it achieves this performance level without collecting any state information. The MCT algorithm leads to the best makespan at the cost of collecting full state information. The proposed algorithm is a promising candidate for widely distributed, and highly loaded Grid systems.

3.8 Conclusion

A new Grid scheduling algorithm is introduced, called the Grid Shadow Routing algorithm. The performance of this algorithm is evaluated by simulation, and the results show its promising performance for aggregate measures such as flowtime. The introduced algorithm and the MCT algorithm use the same system parameters, but they differ in the amount of dynamic state information used for scheduling, in which the Grid Shadow Routing algorithm requires *zero* state information. The results and analysis conclude that in Grid environments where the system elements are not tightly coupled, and the communication cost is considerable, applying the proposed algorithm can be a good alternative to the MCT algorithm.

Chapter 4

Reducing the Search Space and State Information in Data Grids [1]

Data Grid systems are designed to enable the access, modification and transfer of large amounts of geographically distributed data (Allcock et al. [2005]). These applications generally include loosely coupled tasks and large data sets, and are found in fields such as high-energy physics, astronomy and bioinformatics. As scheduling decisions have a direct effect on the system performance, it is crucial to employ an appropriate scheduling algorithm for each Grid system. For instance, an algorithm seeking to assign the best computational resource to each task may perform well for computation intensive tasks but lead to significant performance degradation in Data Grids. The scheduling problem in Data Grids involves several challenges such as finding appropriate resources for both the data location and the computation on the data.

The basic approach in most Data Grid scheduling algorithms is to search through all possible combinations of computational and data storage resources, compute each combination's cost, and select the combination with the minimum cost (Dong and Akl [2007], Alhusaini et al. [1999]). The calculated cost may include different factors such as the communication and computation times. As the search space in this approach is typically very large, different Data Grid scheduling algorithms attempt to reduce the search space. Moreover, the scheduler should take into account heterogeneity in resources and tasks. In this chapter, a new scheduler is introduced, which reduces the search space, and simultaneously requires less state information than typical Data Grid schedulers. Considering the system heterogeneity in the scheduling decisions leads to improvements in the average completion time of the proposed scheduler.

This chapter introduces the Data Grid workload model and the system models in Sections 4.1

[1] This chapter is mostly based on: A. Rasooli and D. G. Down, *Improving Overhead of Scheduling in Computational and Data Distributed Computing Systems*, **Poster presented in IBM CASCON 2010 Technology Showcase**, November 1 4, 2010.

and 4.2, respectively. A new Data Grid scheduler is proposed in Section 4.3, where the scheduler consists of computation intensive and data intensive algorithms, introduced in Sections 4.4 and 4.5, respectively. The proposed scheduler is evaluated using a Data Grid simulator in Section 4.6. An overview of Data Grid scheduling algorithms is provided in Section 4.7. Finally, the last section presents concluding remarks.

4.1 Workload Model

Generally in Data Grid systems, a large volume of data is generated by a small number of resources, and this data is analyzed by various users from different scientific points of view. Moreover, the tasks submitted by a user are mostly in the same scientific field with similar computational and storage requirements. Therefore, classifying tasks based on their users is reasonable in most Data Grid systems.

This chapter introduces the following workload model which was inspired by the studies of the real Data Grid workloads presented in (Goddeau [2005]): it is assumed that the tasks are classified into a number of classes. Each class has two sets of requirements: computation and data, where for each class they are defined based on the average requirements of the included tasks. The task's computation time (length) and its required data size define its computation and data requirements, respectively. Similarly, each resource has two parts: the computation and the storage parts, which are called the computational resource and storage resource, respectively.

- M: is the total number of resources.
- N: is the total number of classes.
- λ_i: is the arrival rate of tasks in class i.
- $\mu^c_{i,j}$: is the computation rate of class i on resource j.
- $\mu^d_{i,j}$: is the rate of accessing class i's required data from resource j.

The mean data access time of class i from resource j is $1/\mu^d_{i,j}$, and the mean computation time of class i on resource j is $1/\mu^c_{i,j}$. The value of $\mu^d_{i,j}$ depends on the average data size required by the tasks in class i, and the storage capacity and bandwidth of the storage resource j. In particular, it does not consider the transmission rate between the resources. All the tasks in the model are assumed to be independent from both data and computation perspectives.

4.2 Data Grid Model

A Data Grid model is defined based on a model of its organization and the network topology. The organization model concerns how the storage resources are organized and located. It also defines the number of replica managers, and how specific data can be located and accessed. Selecting an organization model for a Data Grid system depends on various factors such as the data size, the data sharing mode, and the source of generated data - single or distributed. There are four commonly used models for organizing storage resources, defined as follows (Venugopal et al. [2006]).

1. **Monadic.** This model gathers all of the data at a central repository, which replies to user queries and provides the data. It has been applied in the NEESgrid Project (The NEESGrid System Integration Team [2004]) in the United States. The key feature of the monadic model is its single point for accessing data. In contrast, for the other models, the data can be entirely or partially accessed at different points, being available through replication. This model is employed in systems where the impact of the replication overhead is greater than the potential improvement in efficient data access.

2. **Hierarchical.** This model has been used in Data Grids which have a single source of data, and the produced data needs to be distributed to worldwide collaborators. An example is the MONARC (Models of Networked Analysis at Regional Centres) group within CERN, where a tiered infrastructure model is proposed for distribution of CMS data (The Large Hadron Collider, CERN [2004]). This model is used to transfer data from CERN to various groups of physicists around the world. In the hierarchical model of this project, the first level consists of the computational and storage resources at CERN, which store the generated data of the detectors. The produced data is distributed to worldwide resources called Regional Centers. From the Regional Centers, the data is passed downstream to national and institutional centers and finally on to the physicists.

3. **Federation.** This model is used in Data Grids created by institutions who wish to share data in existing databases (Rajasekar et al. [2004]). One example of a federated Data Grid is the BioInformatics Research Network (BIRN) (Jovicich et al. [2005]) in the United States. Researchers at a participating institution can request data from any of the databases within the federation as long as they have the proper authentication. Each institution preserves control over its local database. This model is used in (Venugopal et al. [2004]), where a Belle analysis Data Grid testbed is used for evaluating their Grid scheduling model.

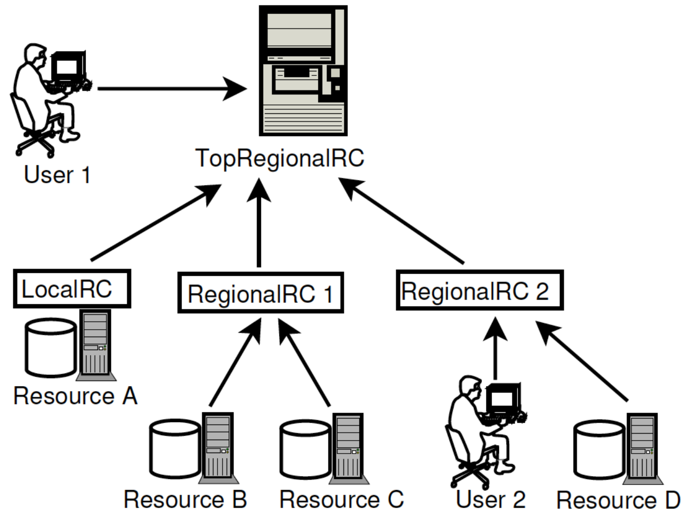


Figure 4.1. A Hierarchical Replica Catalog model (The European DataGrid Project [2007]).

4. **Hybrid.** This model is a combination of the above models, and has begun to emerge as Data Grids mature and enter into production usage.

This research targets Data Grid systems where the massive amount of data generated at a central resource needs to be distributed using a robust mechanism. Each user's tasks may require only a subset of the entire data set generated at the central resource. Therefore, this research uses a hierarchical model, which also simplifies the process of maintaining consistency in the system.

The experiments in this chapter follow the same approach used by the EU DataGrid project (The European DataGrid Project [2007]), in which the hierarchical model is constructed as a tree (Figure 4.1). In this model, some information is stored in the root of the tree, and the rest is located in the leaf nodes. There is a root Replica Catalog (RC), and a number of leaf RCs. This approach enables the leaf RCs to process some of the queries from users and resources; thus, reducing the load on the root RC. The network topology of the Data Grid system used in this chapter is presented in Figure 4.2, which is the same as the network topology of the EU DataGrid TestBed 1 (Hoschek et al. [2000]). This research follows the data model (read-only, atomic) that has been applied for most of the experiments on the EU Data Grid, where a data set is a non-splittable unit of information.

In a DataGrid system each resource consists of two parts: a computational part (which includes the processing elements), and a storage part with a specific size and bandwidth for accessing the data. Each resource has a replica manager which protects the consistency of replicas with the original data. Moreover, the system has a central scheduler that decides where and when to create the data replications, and where to submit the incoming tasks.

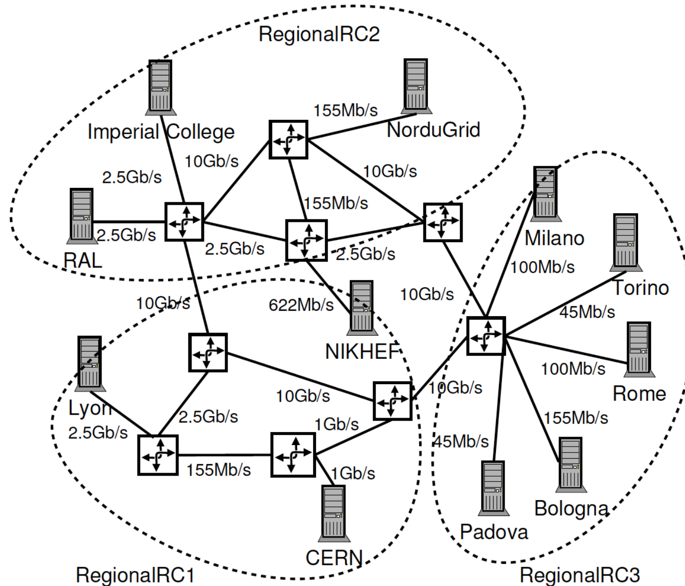


Figure 4.2. The Network Topology of EU DataGrid TestBed 1 (Venugopal et al. [2006]).

4.3 Proposed Scheduler

In this section, a novel Data Grid scheduler is introduced, called DATALPAS (DATA intensive Linear Programming based Affinity Scheduler), which consists of two parts: a computation intensive algorithm and a data intensive algorithm. The former only considers the computational requirements of each class, and provides a set of suggested computational resources for each class i, $Compute_i$. On the other hand, the data intensive algorithm considers the data requirements of each class i, and computes a set of suggested storage resources for that class, $Storage_i$. Both the data intensive and computation intensive algorithms use a class-based workload model and optimization methods to reduce the search space and required state information. Moreover, they consider system heterogeneity in their optimization methods to improve the system performance. The proposed scheduler, presented in Figure 4.3, uses both sets of suggested computation and storage resources.

The values of Completion($task$, j_1) and Transmission($task$, j_1, j_2) are received as state information from resources j_1 and j_2, respectively. The expected transmission time depends on the current available bandwidth inside resource j_2, and the current available bandwidth between resources j_1 and j_2. If any of the system parameters such as arrival rates, resource computation rates, or resource data access rates changes, the scheduler recomputes its computation and data intensive optimizations based on the new parameters, and updates the sets $Compute_i$ and $Storage_i$.

1. Upon arrival of a *task* from a class (say i):

   Calculate Response time of the *task* as follows:

   $$Response(task, j_1, j_2) = \text{Completion}(task,\ j_1) + \text{Transmission}(task,\ j_1,\ j_2),$$

   where:

   - The computational resource $j_1 \in Compute_i$
   - The storage resource $j_2 \in Storage_i$
   - Completion($task$, j_1)= expected completion time of *task* on resource j_1
   - Transmission($task$, j_1, j_2)= expected data access time for input data of *task* on resource j_2 plus the transmission time of the input data from resource j_2 to resource j_1.

2. Select the resources j_1 and j_2 with the minimum $Response(task, j_1, j_2)$.

3. Assign *task* to resource j_1, while receiving its input data from storage resource j_2.

Figure 4.3. The Proposed Scheduler

In typical Data Grid schedulers, $Compute_i$ and $Storage_i$ sets include all the computational and storage resources, respectively. Although this approach may provide the minimum completion time for the incoming task, it leads to significant overheads due to the large search space and corresponding state information. The DATALPAS scheduler aims to reduce the number of resources in the sets $Compute_i$ and $Storage_i$ through its computation intensive and data intensive algorithms. The following two sections introduce these two algorithms.

4.4 Computation Intensive Algorithm

This section proposes an algorithm for providing a set of suggested computational resources for each class of tasks. The algorithm solves the following Linear Programming (LP) problem, in which the decision variables are λ^c and $\delta^c_{i,j}$, and the corresponding optimal solution is defined by λ^{c*}, $\delta^{c*}_{i,j}$. The main idea of this LP is to determine which computational resources are the best choices for each class when the system is highly loaded.

$$\max \lambda^c$$

$$s.t. \sum_{j=1}^{M} \frac{\mu_{i,j}^d}{\mu_{i,j}^c + \mu_{i,j}^d} \times \mu_{i,j}^c \times \delta_{i,j}^c \geq \lambda^c \alpha_i, \ for\ all\ i = 1, \ldots, N, \tag{4.1}$$

$$\sum_{i=1}^{N} \delta_{i,j}^c \leq 1, \ for\ all\ j = 1, \ldots, M, \tag{4.2}$$

$$\delta_{i,j}^c \geq 0, \ for\ all\ i = 1, \ldots, N, \ and\ j = 1, \ldots, M. \tag{4.3}$$

The variables and constraints of the above LP are defined as follows:

- $\delta_{i,j}^c$ is the proportion of computational resource j allocated to class i.
- The computation proportion of each class i on each resource j is extracted as follows:

$$\frac{\text{mean computation time of class } i \text{ on resource } j}{\text{mean total execution time of class } i \text{ on resource } j} = \frac{1/\mu_{i,j}^c}{1/\mu_{i,j}^c + 1/\mu_{i,j}^d} = \frac{\mu_{i,j}^d}{\mu_{i,j}^c + \mu_{i,j}^d}.$$

- Constraint (4.1) enforces that the total computation capacity allocated for a class should be at least as large as the scaled arrival rate for that class. This constraint is needed to keep the computational resources stable, and it allows the computational load to increase only by an amount that can be satisfied by the available computational resources. Constraint (4.2) prevents over allocating a computational resource, and (4.3) states that negative allocations are not allowed.
- λ^{c*} can be interpreted as the maximum capacity of the computational resources.
- ${\delta_{i,j}^c}^*$ is defined as the long run fraction of time that computational resource j should spend for class i to keep the load of all computational resources stable under maximum capacity conditions.

The above LP solution presents a set of suggested computational resources for any class i, as follows:

$$Compute_i = \{j : {\delta_{i,j}^c}^* \neq 0\}.$$

The LP solution aims to provide the best set of resources for each class by considering the heterogeneity of tasks and resources.

4.5 Data Intensive Algorithm

An optimization method is used to provide the best set of storage resources for each class. The proposed method considers heterogeneity in the storage resources and the input data sizes. Using the suggested set of storage resources, the algorithm generates replicas of each class's input data on the suggested storage resources. Generating a large number of replicas can improve the data access rate. However, doing so can lead to significant overhead due to the cost of maintaining consistency between the replicas. Moreover, the storage resources may become overloaded, which degrades the performance. An LP to find the best matching of storage resources for each class requires the following variables:

- The decision variables $\delta^d_{i,j}$ represent the proportion of storage resource j which is allocated to the input data of tasks in class i.
- As the goal is to find appropriate storage resources, the proportion of time that each class spends on accessing data from each resource is required:

$$\frac{\text{mean data access time for class } i \text{ on resource } j}{\text{mean total execution time of class } i \text{ on resource } j} = \frac{1/\mu^d_{i,j}}{1/\mu^c_{i,j} + 1/\mu^d_{i,j}} = \frac{\mu^c_{i,j}}{\mu^c_{i,j} + \mu^d_{i,j}}.$$

The proposed LP, presented as follows, considers the data access costs for all storage resources to find the best set of storage resources for each class. For this purpose, it maximizes the capacity of the system by finding the best choices for each class when the data load on storage resources is very high.

$$\max \lambda^d$$

$$s.t. \sum_{j=1}^{M} \frac{\mu^c_{i,j}}{\mu^d_{i,j} + \mu^c_{i,j}} \times \mu^d_{i,j} \times \delta^d_{i,j} \geq \lambda^d \alpha_i, \ \text{for all } i = 1, \ldots, N, \tag{4.4}$$

$$\sum_{i=1}^{N} \delta^d_{i,j} \leq 1, \ \text{for all } j = 1, \ldots, M, \tag{4.5}$$

$$\delta^d_{i,j} \geq 0, \ \text{for all } i = 1, \ldots, N, \ \text{and } j = 1, \ldots, M. \tag{4.6}$$

- The left-hand side of (4.4) represents the total data storage capacity assigned to class i by all of the storage resources. The right-hand side represents the arrival rate of tasks

that belong to class i scaled by a factor of λ^d. Thus, (4.4) enforces that the total storage capacity allocated for a class should be at least as large as the scaled arrival rate for that class. This constraint is needed to keep the storage resources stable. The satisfaction of this constraint guarantees that the system can provide the required bandwidth and required storage capacity with its available storage resources.

- The constraint in (4.5) prevents over allocation of the storage resources, and (4.6) states that negative allocations are not allowed.
- The optimal solution ${\lambda^d}^*$ is interpreted as the maximum capacity of the storage resources, and ${\delta^d_{i,j}}^*$ is the long run fraction of time that storage resource j should spend retrieving class i's data to stabilize the load on storage resources under maximum capacity conditions.

The physical meaning of ${\lambda^d}^*$ requires that its value is greater than or equal to one. Therefore, if ${\lambda^d}^*$ is less than one, the data access rate for storage resources is insufficient. This situation implies that the system is overloaded in terms of the data load. In this case, it is recommended to use the main storage resource rather than generating a replica on low rate resources.

The LP solution suggests the best set of replication resources for each class. A set $Replica_i$ is defined as a set of suggested storage resources for class i, where:

$$Replica_i = \{j : {\delta^d_{i,j}}^* \neq 0\},$$

and $|Replica_i|$ denotes the size of the set $Replica_i$. Using the suggested storage resources, the scheduler generates replicas as follows. $Storage_i$ is defined as the set of all storage resources for the required data for class i. Whenever a new replica of class i's data is generated, the set $Storage_i$ is updated with the new replica information. Based on the available network bandwidth, the following replication methods can be used:

- **Method 1.** For systems with high available bandwidth, it is suggested to generate all the replicas at the beginning of scheduling as follows:

  1. For each class i generate a replica of its input data on each storage resource in the set $Replica_i$.
  2. If any of the system parameters change (data access rates of the resources or the arrival rates of classes), the LP is solved again with the updated parameters. Consequently, if the relation ${\delta^d_{i,j}}^* \neq 0$ changes for any of the (i, j) pairs, the set $Replica_i$ is updated for the corresponding class.

3. If the set $Replica_i$ changes for any class i, new replicas are generated for that class. The storage resources which are not in $Replica_i$ remove the input data of class i from their storage.

Simultaneously generating all of the replicas may add a considerable load to the network when the data is being transferred to the corresponding resources. This research considers Data Grid systems where the system is underloaded at the beginning of scheduling; therefore, the extra load on the network is acceptable at this time. However, in some Data Grid systems the benefits of generating all replicas at the same time may not compensate for its overhead cost. These Data Grid systems, called low bandwidth systems, require replicas to be generated whenever performance is sufficiently degraded. To determine if a system is low bandwidth, different factors must be considered, such as available network bandwidth, replicated data size, and data access rate on the storage resources. The following introduces a possible replication method for low bandwidth Data Grid systems. However, as these Data Grid systems are not the main focus of this research, the further analysis of low bandwidth systems is left as future work.

If the transmission time of data required by class i is greater than the mean data access time on their storage resources, it is suggested to use Method 2 for replication. In this situation the transmission cost would be greater than the access time using a single point for accessing data. Therefore, using Method 2 is a means to attempt to mitigate the high overhead.

- **Method 2.** The following method is introduced to reduce the replication overhead of Method 1.

  1. For any class i, and any storage resource j in the set $Replica_i$, compute the expected time to transmit the required data from the main database to resource j. This requires the value of the currently available bandwidth in the network from the main database to resource j.
  2. The resource with the minimum expected transmission time is selected, and a replica of class i's input data is generated on the selected resource. By generating a replica on any resource j for any class i, the corresponding resource is removed from the set $Replica_i$, and is added to the set $Storage_i$.
  3. To generate extra replicas, the scheduler monitors the system performance in terms of the data access times. For this purpose, it defines the data availability metric, introduced in Section 2.3.1, for each class. The average data availability over all tasks of class i ($\bar{a}_i$) can be written as

$$\bar{a}_i = \frac{\sum_{j=1}^{M} \sum_{l=1}^{n_i^t} a_{l,j}}{n_i^t}$$

where n_i^t is the total number of tasks in class i. The data availability time ($\bar{a}_i$) for each class i is updated when a new task of class i arrives to the system.

If the value of $\bar{a}_i$ for any class i reaches a predefined $threshold_i$, the scheduler generates a new replica. The $threshold_i$ value can depend on different factors such as the available bandwidth, the load on the system, and the data access rates for storage resources. One possible value for $threshold_i$ is:

$$\frac{\sum_{j=1}^{M} \frac{1}{\mu_{i,j}^d}}{M}.$$

Using this $threshold_i$ value for each class i, it is guaranteed that the time for accessing one unit of data for a class i is less than its average mean data access time over all resources. As considering the low bandwidth Data Grid system is beyond the main focus of this research, deeper analysis on defining the thresholds is left as future work.

The scheduler monitors the values of $\bar{a}_i$ for all classes. If in any class i, $\bar{a}_i$ reaches a defined threshold and $|Replica_i| \neq 0$, or if there is any change in the system parameters, steps 1 and 2 are repeated for class i.

This method generates replicas at the beginning of scheduling, and it generates additional replicas only when a small number of replicas leads to performance degradation. As the network overhead of this method is lower than the first one, it is expected to perform better for low bandwidth systems.

4.6 Experimental Results

A heterogeneous Data Grid system is defined to evaluate the proposed scheduler. An extended version of GridSim (called Data GridSim (Sulistio et al. [2008])) designed for simulation of Data Grids is used for the experiments. An evaluation component is added to this simulator to calculate the desired performance metrics and provide the final results in the log files. The simulated Data Grid system consists of six heterogeneous resources, each capable of processing five classes of tasks. Each resource has a computational part and a storage part, where the characteristics of each resource are provided in Table 4.1. The arrival rates are defined as follows, in which the arrival rate of the tasks in class i (λ_i) is the *ith* element of the array

$$\lambda = \begin{bmatrix} 21.3 & 7.2 & 8.1 & 0.5 & 1.1 \end{bmatrix}.$$

The computation rates are presented in the following matrix. The (i, j) element of the matrix is the computation rate of a task from class i on the computational part of resource j.

$$\mu^c = \begin{bmatrix} 6.85 & 6.45 & 6.78 & 13.63 & 9.48 & 39.61 \\ 20.06 & 30.43 & 41.18 & 79.42 & 76.10 & 139.10 \\ 16.63 & 23.23 & 34.29 & 44.88 & 48.58 & 71.05 \\ 2.82 & 2.61 & 6.08 & 6.69 & 6.72 & 8.35 \\ 0.99 & 1.01 & 2.7 & 2.84 & 2.73 & 2.88 \end{bmatrix}$$

The following matrix presents the data access rates. The (i, j) element of the matrix is the data access rate for a task of class i from the storage part of resource j.

$$\mu^d = \begin{bmatrix} 9.85 & 18.35 & 17.42 & 15.37 & 16.13 & 8.69 \\ 10.34 & 17.87 & 36.52 & 4.18 & 59.80 & 5.80 \\ 2.27 & 0.97 & 14.01 & 0.92 & 23.93 & 1.45 \\ 0.18 & 0.39 & 1.52 & 0.91 & 1.58 & 0.35 \\ 0.01 & 0.09 & 0.3 & 0.06 & 0.27 & 0.12 \end{bmatrix}$$

The data access rates depend on the input data size in each class, and the speed of data retrieval in the corresponding storage resources. The parameters for our experiments are taken from a real system, which is implemented for executing BLAST applications (Goddeau [2005]). The arrivals of class i tasks follow a Poisson process with rate λ_i, while computational and data access times of tasks in class i on resource j follow exponential distributions with rates $\mu^c_{i,j}$, and $\mu^d_{i,j}$, respectively. The simulation runs until 20,000 tasks arrive to the system, at which point no further arrivals occur, and the simulation then continues until the system becomes empty.

Table 4.1. Resources used in the Data Grid experiments

Resource	CPU	Memory	Disk
Type 1	733 MHz	256 MB	40 MB/s IDE
Type 2	525 MHz	8 GB	60 MB/s SCSI
Type 3	2.8 GHz	256 MB	40 MB/s IDE
Type 4	2.8 GHz	256 MB	60 MB/s SCSI
Type 5	2.8 GHz	1.5 GB	40 MB/s IDE
Type 6	2.8 GHz	1.5 GB	60 MB/s SCSI

In the simulated system, there are two central data bases which generate the data for the system. The BLAST application suite has two databases, named NR (which stores protein sequences), and NT (which stores nucleotide sequences). The input data of classes 2 and 3 are generated in database NR, and database NT stores the generated data used by the rest of the classes. Therefore, at the beginning of the experiment, all data sets are placed on the NR and NT databases. Copies of the data sets will then be replicated based on the scheduler's decisions.

To evaluate the DATALPAS algorithm, its performance is compared with four benchmark Data Grid scheduling algorithms. In order to have a fair comparison, the same number of replicas is generated for all of the schedulers. In the experiments, when the first task of any class i arrives, the replication method generates replicas of its required file on all storage resources of the set $Replica_i$. However, upon arrival of subsequent tasks of the same class i, no extra replication is generated unless there is a change in any of the data access rates, in which case the set $Replica_i$ is modified. The four simulated scheduling algorithms used for comparison are as follows:

- *Minimum ComPutation Time (MCPT)*: searches among all the possible combinations of computational and storage resources, and chooses a combination with the minimum Computation time. This algorithm is used as a representative of algorithms which have full state information about the computational resources.

- *Minimum Data Access Time (MDAT)*: searches all possible combinations of computational and storage resources, and selects a combination with the minimum data access time. The data access time from storage resource j_1 to computational resource j_2 is calculated by:

  $$\textit{Data access time from } j_1 \textit{ to } j_2 = \textit{Data retrieval time in } j_1 + \textit{Transmission time from } j_1 \textit{ to } j_2$$

  This algorithm is a representative of algorithms with full state information about the storage resources.

- *LPAS*: is mainly used in Computational Grid systems, and it does not consider the complexities of accessing data in its scheduling (Al-Azzoni and Down [2008b]). It uses an LP and partial state information to find the best resource and task matching. This algorithm is used as a representative of algorithms which consider resource and task heterogeneity in their scheduling decisions.

- *Minimum Response Time (MRT)*: searches all possible combinations of computational and storage resources, and selects a combination with the minimum response time. As the MRT requires full state information, and uses an exhaustive search, it is not applicable in real systems. However, it can be used as a benchmark.

The algorithms' performance is compared in terms of the four main performance metrics for Data Grids, as introduced in Section 2.3.1. Figure 4.4 compares the algorithms based on the time that each algorithm spends on making their scheduling decisions. Each experiment was repeated 30 times to construct 95%-confidence intervals. The lower and upper bounds of the confidence intervals are presented with lines on each bar. The LPAS algorithm, which only needs to gather

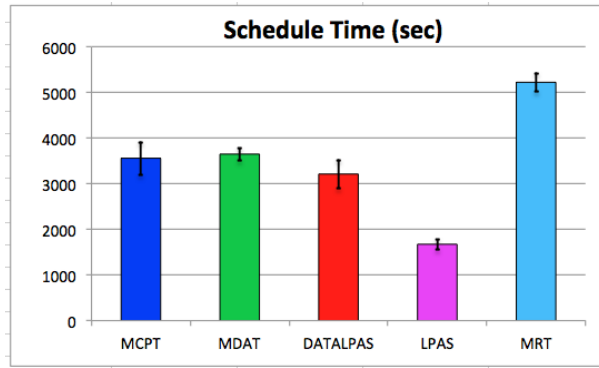


Figure 4.4. The Schedule Generation Time of Algorithms

the completion time of tasks from some of the resources, has the minimum scheduling time among the considered algorithms. On the other hand, the MRT algorithm has the maximum scheduling time due to gathering full state information from all of the computation and storage resources. The large search space of the MRT algorithm is the cause of the large overhead. The MCPT and MDAT algorithms gather state information only from the computational and storage resources, respectively. As a result, their scheduling times are similar. As the DATALPAS algorithm gathers state information from some of the computation and storage resources, it has less scheduling overhead compared to the MRT, MCPT and MDAT algorithms.

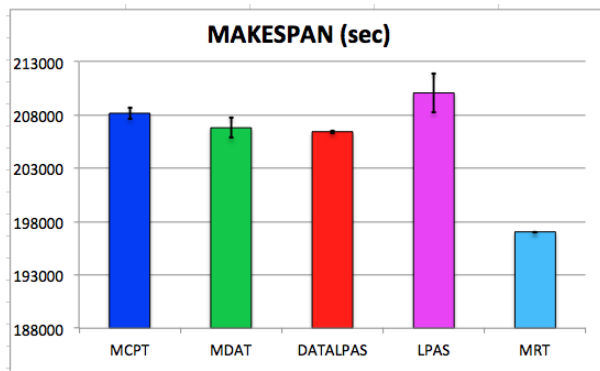


Figure 4.5. Makespan of the Algorithms

Figures 4.5 and 4.6 present the makespan and flowtime results, respectively. The LPAS algorithm, which makes scheduling decisions based on available computation rates, and does not consider data issues, leads to poor makespan and flowtime. As expected, the MRT algorithm has the best flowtime and makespan with the cost of using full state information, and maximizing the communication cost. As the MDAT algorithm uses more state information compared to the MCPT algorithm, in this data intensive workload, it leads to better performance than the MCPT algorithm. The results suggest that the DATALPAS algorithm can achieve competitive makespan

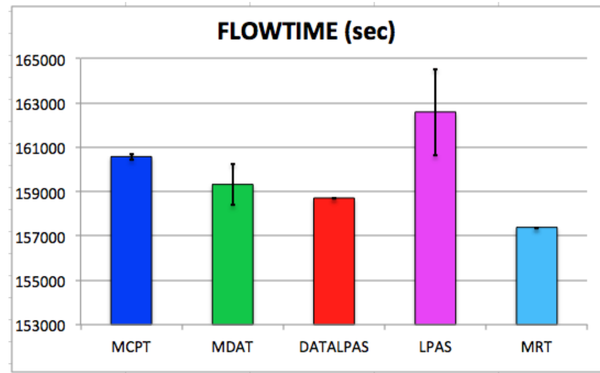


Figure 4.6. The Flowtime of Algorithms

and flowtime with the MRT algorithm. Compared to the MRT algorithm, the DATALPAS algorithm has two main advantages: first, it requires much less state information; therefore, it reduces the network communication load generated for scheduling decisions. Second, the MRT algorithm uses an exhaustive search approach, but the DATALPAS algorithm searches over a much smaller space, which leads to using less computational resources for scheduling.

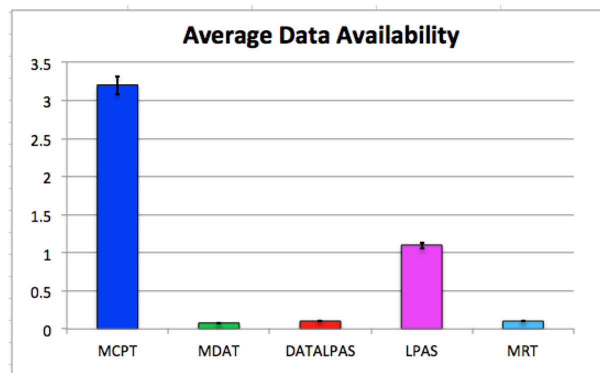


Figure 4.7. The Data Availability Time of Algorithms

Finally, in Figure 4.7, the average data availability is presented. As the MCPT algorithm aims to select the best computational resources, it leads to poor data availablity time. The MDAT algorithm, which makes scheduling decisions based on minimizing the data access time, results in the minimum average data availability. Here, the DATALPAS algorithm performs well.

4.7 Related Work

In general, when the scheduling problem with data movement is considered, there are two main approaches: whether data replication is allowed or not. This section provides an overview of relevant scheduling techniques categorized from different perspectives.

4.7.1 Data Replication

An approach which does not allow replication can serve better in Grid systems where the overhead of replication is not compensated by an increase in efficiency of accessing data (e.g. systems where all accesses are local). This approach is used in the NEESgrid (The NEESGrid System Integration Team [2004]) Data Grid project, where there is only a single point for accessing data. In Pegasus (Luther et al. [2005]), it is assumed that accessing an existing data set is always preferable to generating a new one. The vast amount of data produced in one place and the wide distribution of computational resources lead to significant performance degradation in Data Grids using a single storage resource (or no replication). The proposed scheduler in this chapter defines replication methods for improving the performance.

Replication is performed using different models. A brief overview of various replication strategies is presented as follows:

- Ranganathan and Foster [2003] consider the data sets in the Grid as a tiered system, and use dynamic replication strategies to improve data access time. These data replication strategies work at the system level instead of the application level, aiming to reduce the bandwidth consumption and the workload at data hotspots, to decrease the network message traffic. However, the data mapping of this approach is not optimal.

- Economy based replication (Park et al. [2004]) is a long term optimization technique aiming to minimize the overall file access cost given a finite amount of storage resources. In (Lamehamedi et al. [2002]), a replication algorithm is proposed which uses a cost model to predict whether replicas are worth creating or not. This approach is found to be more effective in reducing average task completion time than the basic case where there is no replication.

- Bandwidth Hierarchy based Replication (BHR) (Ranganathan and Foster [2001], Park et al. [2004]) extends resource level replica optimization to the network level based on the hierarchy of bandwidth appearing in the Internet. As the bandwidth within a region is typically larger, BHR aims to increase the size of required data in the same region to fetch replicas faster. The proposed replication algorithm in this chapter uses the same tiered structure as (Ranganathan and Foster [2003]). However, the proposed algorithm is in the application level meaning that each task's requirement is considered to generate replicas close to the tasks which use them most. The tiered structure reduces the search and access time for specific data, and provides more convenient maintenance for consistency of the replicated data in all resources.

4.7.2 Computation and Data Scheduling Interaction

In Data Grids the scheduling process includes two separate parts to schedule the computation part and the data part of each task. There are two different approaches with respect to the interaction of computation scheduling and data scheduling: decoupling them or conducting joint scheduling.

Decoupled scheduling is used in (Ranganathan and Foster [2002]) to consider dynamic task scheduling with data staging requirements. They perform the computation scheduling and data replication strategies independently. They suggest four simple and dynamic computation scheduling algorithms as follows: TaskRandom (selects a random resource for computation); TaskLeastLoaded (selects the resource with minimum load); TaskDataPresent (selects a resource where the required data is already located), and TaskLocal (runs a task at the local resource). Moreover, three data scheduling algorithms are used: DataDoNothing (performs no active replication); DataRandom (replicates a popular data set to a random resource), and DataLeastLoaded (selects the least loaded resource as the destination for a new data set replication). Using various combinations of these computation and data scheduling algorithms, 12 schedulers are defined. In the no replication case (DataDoNothing replication algorithm), a task execution is preceded by a fetch of the required data, leading to a strong coupling between task scheduling and data movement. By contrast, the other two replication strategies are loosely coupled to the task execution.

Joint scheduling is defined based on the fact that in general, selecting a computational resource and a storage resource for a task are two interrelated issues. The choice of a computational resource may depend on the input data location, and the storage resource selection may depend on where the computation will take place. If the data sets involved in a computation are large, it is preferable to move the computation close to the data. On the other hand, if the computation size is very large compared to the data transfer cost, selecting the best possible computational resource will have higher priority than selecting the best data location. Based on this idea, Alhusaini et al. (Alhusaini et al. [1999]) combine scheduling of computation with data replica selection to reduce the makespan for a collection of tasks unified as a Directed Acyclic Graph (DAG). The data access and data transfer costs are combined to compute the total time cost of a task in the DAG. A static level-by-level algorithm is used to search for an optimal schedule. In this algorithm, the original DAG is partitioned along with the directed edges into levels, where all tasks in the same level are assumed to be independent. Well known min-min and max-min algorithms are applied to schedule tasks in the same level. However, this paper does not discuss the problem of finding levels in an unbalanced task graph; moreover, a level-by-level partition can not harness locality to facilitate tasks with dependencies.

Dong et al. (Dong and Akl [2007]) introduce an algorithm that jointly considers data and computation scheduling; the proposed workflow scheduling algorithm is called JDCS (Joint Data and Computation Scheduling). It is assumed a workflow can be represented by a DAG, and the objective is to minimize the total schedule length of the entire workflow. The algorithm uses the possibility of overlapping the input data preloading time and the computation time to reduce the task waiting time. This is achieved by using a back-trace technique. To overcome the difficulties of performance fluctuation, JDCS takes advantage of mechanisms such as Grid performance prediction and resource reservation (Yang et al. [2003]). These mechanisms can capture resource performance information and provide performance guarantees. The problem with this approach is that most Grid workflows can not be represented as a DAG.

The research in (Desprez and Vernois [2006]) aims to maximize the throughput for independent tasks in a Grid, where each task requires a certain data set, and there is a constraint for storage capacity in each computational resource. The algorithm is based on the following assumptions: (i) data replication is allowed; (ii) a task can only use the local data on the same resource (whose storage capacity is limited); (iii) the execution time of a task is linear with its input data set size, and (iv) the number of each task type in a task set follows a fixed proportion. The objective is to maximize the total size of completed tasks in a specified time interval. As the problem itself is NP-complete, an integer solution to the defined linear programming problem is used as an approximation. Using simulation, the authors verified their method by comparing with the MCT and max-min algorithms. As the algorithm is static, it is clearly not applicable for long tasks on highly loaded resources (or when the load cannot be predicted). As a result, this heuristic does not always give the best mapping of data. Moreover, this algorithm does not consider communication costs.

4.7.3 Performance goals of schedulers

The Data Grid scheduling algorithms can also be classified based on their performance goals, which generates the following classes: system-centric, economy-based and application-centric schedulers. A system-centric scheduler focuses on the overall performance of the entire set of tasks and the entire Grid system. An example of this is Condor (Litzkow et al. [1988]), which aims to increase workstations' utilization by using idle workstations for sharing task execution. An economy-based scheduling system is based on the idea of market economy. Under this scheme, scheduling decisions are made based on an economic model. For example Nimrod-G (Abramson et al. [2000]) implements a producer and consumer mechanism for Data Grids. The economic methods compare different costs of replica replacement and remote access.

Buyya et al. [2005] propose a scheduling algorithm considering two kinds of costs: the economic budget and time. The algorithm aims to optimize one cost while constraining the other, e.g., minimize the budget, while not missing a specified deadline. The incoming tasks consist of a set of independent tasks each of which requires a computational resource and access to a number of data sets located on different storage resources. The algorithm assumes that each data set has only one copy in the Grid; therefore, the resource selection is only for computational resources. As a result, the algorithm has to take into account the communication costs from data storage resources to different computational resources as well as the computation costs. Instead of searching thorough the number of data sets requested by a task, whose search space is exponential, the resource selection procedure simply performs a local search, which only guarantees that the current mapping is better than the previous one. Therefore, the search procedure cost is linear. The main drawback of this strategy is that it is not guaranteed to find a feasible schedule even if there is one.

An application-centric scheduling system tries to maximize the performance of individual tasks. An adaptive scheduling model, such as AppLes (Berman et al. [2003]), is one of the most important application-centric scheduling systems. AppLes focuses on the run-time resource availability and the development of scheduling agents. The scheduling agents consider different requirements of tasks based on load prediction and dynamic resource availabilities. To gain precise information about dynamic resource availabilities and other influencing factors, AppLes uses NWS (Network Weather Service) (Swany and Wolski [2002]).

The choice of scheduling model based on access cost, such as Chameleon (Karypis et al. [1999]), is a type of application-centric scheduling strategy. Chameleon calculates the cost by considering the location of resources holding data sets, and the location of resources performing computation. A task is scheduled by comparing the access costs in different Grid resources. The access costs are defined according to five possible situations of data and computational resources. The scheduler monitors dynamic system attributes including network bandwidth and the number of available resources, and makes scheduling decisions based on this information. Then, a data mover element decides the location and movement of data.

The Access Cost scheduling algorithm (AC) (Cameron et al. [2003]) schedules data intensive tasks on Grid resources with minimal access cost. Access cost is an estimated value based on the network status for obtaining all files required by each task. This scheduling algorithm considers the importance of file distribution, whereas it neglects the task length as an influencing factor in the waiting queues. In contrast to the algorithm proposed in this chapter, the AC algorithm assumes that the data distribution will be static in both the scheduling and execution times. However, the distribution differs when replica replacements occur frequently in each Grid resource.

4.8 Conclusion

This chapter studies Data Grid models and associated workload models. A new scheduling algorithm is introduced for Data Grids, whose main objective is reducing the required state information and search space of the scheduler. The proposed algorithm is evaluated in a simulated high bandwidth Data Grid system.

We address the average completion time in various distributed computing systems, by considering the system heterogeneity. The previous chapters addressed this goal in Grid computing systems. First, a scheduler was introduced for Computational Grids, which concentrate on computing-intensive tasks. Then, a more complicated version of Grid computing systems, Data Grids, were addressed, and a scheduler was introduced for the tasks which are both data-intensive and computation-intensive. As the Cloud computing paradigm and Hadoop systems deployed in the Cloud were proposed as a successor to the Grid framework, the rest of this PhD thesis will concentrate on introducing schedulers for Cloud and Hadoop environments. The proposed schedulers address the idea of improving performance by considering system heterogeneity.

Chapter 5

Hadoop Background

As the demands for processing large scale data increase, more computation and storage resources are required. The Big Data paradigm (Agrawal et al. [2011]) has emerged from the data generated every day from various sources such as sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals. The problem of Big Data is more than a matter of storage size; it is the issue of analyzing huge data sets to gain insight in new and emerging types of data and content. Distributed computing methods such as Grid computing (specifically Data Grids) address the storage problem of Big Data. However, to harness the available resources for improving the performance of large applications running on Big Data, more efficient methods are required for processing data on distributed and scalable resources. MapReduce (Dean and Ghemawat [2008]) is a programming model, introduced by Google (Lämmel [2008]), that provides an efficient framework for automatic parallelization and distribution of large scale computations and data analysis.

Hadoop is a well known open source implementation of the MapReduce programming model (Apache Hadoop Foundation [2010b]). It is designed to scale up from a single server to thousands of resources, with a high degree of fault tolerance. The distributed file system underlying the Hadoop system provides efficient and reliable distributed data storage for applications involving large data sets. Node failures are automatically handled by the Hadoop framework (Apache Hadoop Foundation [2010b]), which enables applications to work with thousands of computers and petabytes of data.

The Hadoop system is widely being used for Big Data analysis in various scientific (e.g., natural language processing (Manning and Schütze [1999]) and seismic simulation (Pratson and Gouveia [2002])) and web applications. Yahoo! and Facebook are employing Hadoop clusters with thousands of cores for their massive data analysis requirements. Hadoop is deployed in several other large companies such as Cloudera, Amazon, and Salesforce Inc. for managing their massive daily

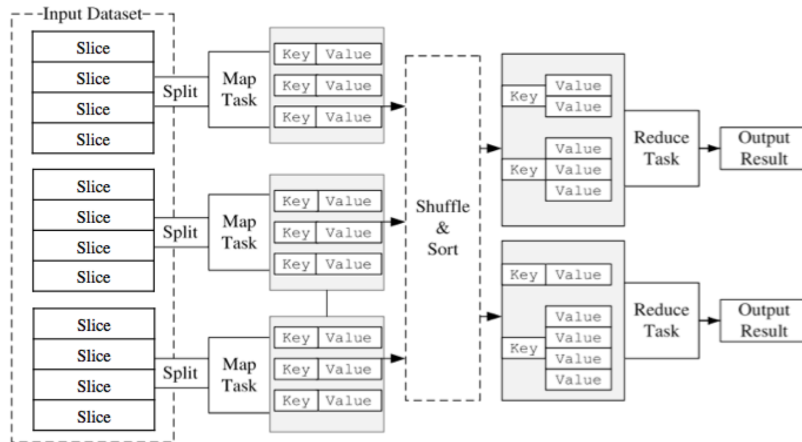


Figure 5.1. The MapReduce model.

data analysis requirements (Pike et al. [2005]). However, there is a growing number of small businesses intending to use Hadoop (Apache Hadoop Foundation [2012]). For instance, Able Grape company (Apache Hadoop Foundation [2012]) uses a small Hadoop System with a two node cluster for analyzing and indexing its textual information.

An introduction to the MapReduce programming model and the Hadoop system is provided in Sections 5.1 and 5.2, respectively. The Hadoop architecture is presented in Section 5.3, and Section 5.4 describes the execution process for the Hadoop system.

5.1 MapReduce Programming Model

Google's MapReduce programming model (Lämmel [2008]) provides processing of large data sets in a massively parallel manner. The model is inspired from the map and reduce functions in Lisp and other functional programming languages (Dean and Ghemawat [2008]). It was designed for processing unstructured data on large clusters of commodity hardware. The model divides the execution of each job into a number of map tasks and reduce tasks.

Figure 5.1 presents the MapReduce model, which consists of the following steps: (i) iteration over the input data sets and slices; (ii) computation of <key, value> pairs from each slice of input data (executed by the map tasks); (iii) grouping of all intermediate values by key; (iv) iteration over the resulting groups, and (v) reduction of each group (executed by the reduce tasks).

The following example, reported in (Apache Hadoop Foundation [2010b]), explains the process of a MapReduce job. The Word Count problem asks for the number of occurrences of each word in a large collection of documents. In this example, the map function could be programmed as the following pseudo-code. The *EmitIntermediate* method generates the <key, value> pairs as it processes the input data.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

The reduce function could be coded as follows, where the *Emit* method generates a single result given all the previously emitted values for a given key.

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

The MapReduce system runs a number of tasks based on the defined map and reduce functions. Figure 5.2 presents the MapReduce steps for running the WordCount job on an example input data set.

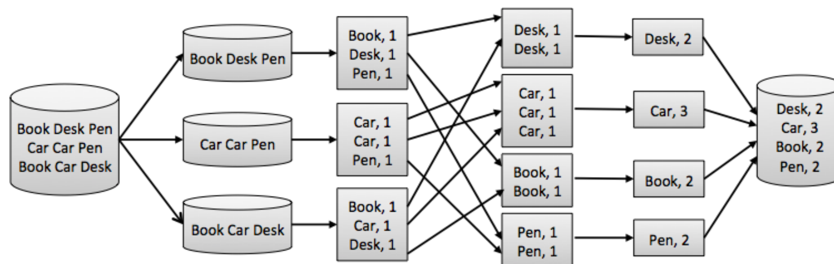


Figure 5.2. The MapReduce process of a Word Count example.

5.2 Hadoop System

Hadoop is a popular open source implementation of the MapReduce programming model. It is implemented by the Apache foundation in the Java programming language (Apache Hadoop Foundation [2010b]). There are three main projects in the Apache Hadoop system (Apache Hadoop Foundation [2010b]): MapReduce, HDFS, and Common. HDFS is a distributed file system that

underlies the Hadoop system; MapReduce provides a framework for automatic parallelization and execution of jobs, and the Common project manages the collaboration of the MapReduce and the HDFS projects. There are various other projects developed for the Hadoop system to provide different services for Hadoop users and developers. Some well known projects include: Pig (Apache Hadoop Foundation [2010f]), Hive (Apache Hadoop Foundation [2010e]), HBase (Apache Hadoop Foundation [2010c]), and Zookeeper (Apache Hadoop Foundation [2010g]). This section introduces different projects for the Hadoop system.

5.2.1 Hadoop MapReduce

Hadoop MapReduce is a software framework for easily writing MapReduce jobs. It splits the jobs' input data sets into independent chunks, which are processed by the map tasks in parallel. The framework sorts the outputs of the map tasks, which are input to the reduce tasks. Typically, both the input and output of a job are stored in the Hadoop file system. The framework performs task scheduling, task execution monitoring, and re-executing failed tasks. Moreover, it offers various features such as application deployment, task duplication, and aggregation of results. The functional style of Hadoop MapReduce automatically parallelizes and executes large jobs over a computing cluster. For the programmer, it abstracts various aspects of distributed and parallel programming.

5.2.2 Hadoop Distributed File System (HDFS)

HDFS is a distributed file system designed to handle very large files on clusters of commodity hardware. It is the primary distributed storage used by Hadoop applications. While HDFS has many similarities with existing distributed file systems, it provides several important features. HDFS is highly fault tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications that have large data sets. Moreover, HDFS relaxes several POSIX (Portable Operating System Interface) (Zlotnick [1991]) requirements to enable streaming access to file system data. However, HDFS is not fully POSIX compliant because the requirements for a POSIX filesystem differ from the target goals of a Hadoop application. The tradeoff of not having a fully POSIX compliant filesystem leads to improving performance for data throughput.

5.2.3 Common

The Common project contains the required files and scripts for starting the Hadoop system. It provides utilities and access to other Hadoop projects, such as:

- File system abstraction. The file system shell includes various shell-like commands that directly interact with the HDFS as well as other file systems that Hadoop supports.

- Service level authorization. The initial authorization mechanism is provided to ensure that clients connecting to a particular Hadoop service have the necessary pre-configured permissions and are authorized to access the given service. For example, a Hadoop cluster can use this mechanism to allow a configured list of users to submit jobs.

5.2.4 Other Hadoop projects

- Hive (Apache Hadoop Foundation [2010e]): is a framework designed on top of the Hadoop system for performing ad-hoc queries on data. It supports HiveQL language, which is similar to SQL, but does not include complete SQL constructs. Hive converts a HiveQL query into a Java MapReduce program and then submits it to the Hadoop cluster. The same outcome can be achieved using a Java MapReduce program directly, but compared to HiveQL, it requires a lot of code to be written and debugged.

- Pig (Apache Hadoop Foundation [2010f]): provides a higher level abstraction over Hadoop MapReduce. It supports the PigLatin language, which is converted to Java MapReduce, and then submitted to the Hadoop cluster. While HiveQL is a declarative language like SQL, PigLatin is a data flow language. This means that the output of one PigLatin construct can be sent as input to another PigLatin construct. In effect, PigLatin programming is similar to specifying a query execution plan, making it easier for programmers to explicitly control the flow of their data processing task.

- HBase (Apache Hadoop Foundation [2010e]): provides real time read and write access on top of the HDFS. Apache HBase is an open source, non-relational, distributed database modeled after Google's BigTable (Chang et al. [2008]). Its goal is to provide hosting of very large tables in the Hadoop cluster. HBase features compression, in-memory operation, and Bloom filters on a per-column basis as outlined in the original BigTable. Tables in HBase can serve as the input and output for MapReduce jobs running in Hadoop.

- Zookeeper (Apache Hadoop Foundation [2010g]): is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. These services are widely used in distributed applications. However, different implementations of these services lead to the need to manage complexity when the applications are deployed. Zookeeper addresses all of these issues for Hadoop developers. Its architecture supports high availability through redundant services. Zookeeper nodes store

their data in a hierarchical name space, where the users can read and write from/to the nodes, and use a shared configuration service.

5.3 Hadoop Architecture

There are two layers in the Hadoop architecture: MapReduce and HDFS, where both layers follow a master-slave model. The high level architecture of the Hadoop system is presented in Figure 5.3. In the MapReduce layer, there is a master node with a Job Tracker, and there are multiple slaves with Task Trackers. In the HDFS layer, the master and slave nodes include a Name Node and Data Nodes, respectively.

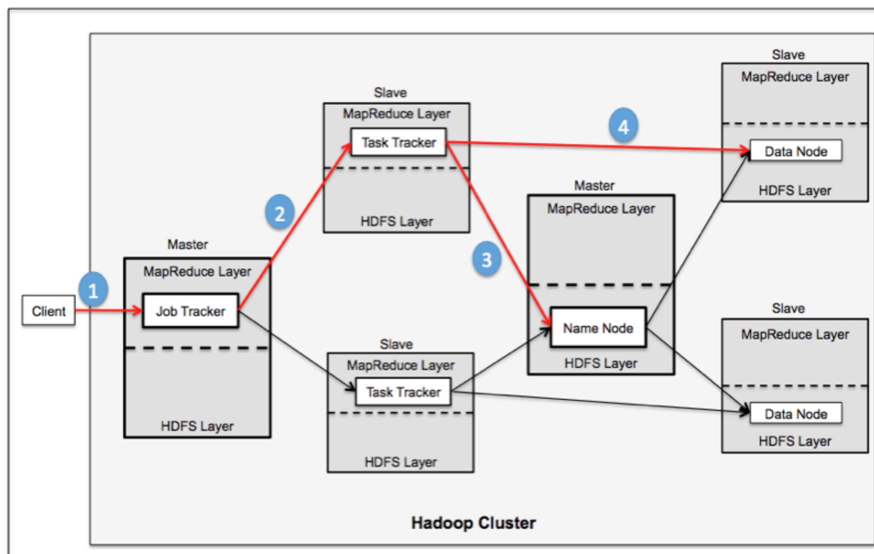


Figure 5.3. The Hadoop Architecture.

In general, small Hadoop clusters include a single master and multiple slave nodes, in which the master node acts as both the Job Tracker and Name Node. In a larger cluster, the HDFS is managed through a dedicated Name Node resource, and similarly, there is a dedicated resource for the Job Tracker.

- MapReduce Layer: Figure 5.3 presents MapReduce as the top layer of each resource in the Hadoop cluster. The Job Tracker daemon, located in the master node, receives the submitted jobs. It is responsible for distributing the job's tasks to the slaves, and scheduling them on the Task Trackers. Moreover, it monitors task execution, and provides status and diagnostic information to the users. The Task Trackers execute the tasks as directed by the Job Tracker.

The Hadoop system allows parallel processing of the map and reduce tasks in each job. Each map task is independent of the others; this means that all map tasks can be performed in parallel on multiple resources. In practice, the number of concurrent map tasks is limited by the data source and/or the number of CPUs near that data. Similarly, a set of reduce tasks can be performed in parallel. All outputs of map tasks that share the same key are presented to the same reduce task. Parallelism in MapReduce offers some possibility of recovering from partial failure of resources during operation. In other words, if one map or reduce task fails, the task can be rescheduled, assuming the input data is still available. Input data sets are in most cases available even in the presence of resource failures, because each data set normally has three replicas stored on three different resources.

The scheduler, which is a fundamental component of the Hadoop system, is part of the Job Tracker in the MapReduce layer. Scheduling in the Hadoop system is pull based, which means that when a resource is free, it sends a heartbeat to the scheduler. Consequently, the scheduler searches through all of the queued jobs, selects a job based on some performance metric(s), and sends a task of the selected job to a free CPU of the pulling resource. The heartbeat message contains some information such as the number of currently free CPUs on the resource. Various Hadoop scheduling algorithms consider different performance metrics in making scheduling decisions.

- HDFS Layer: the master node in the HDFS layer has a single Name Node, which manages the file system namespace and regulates access to files by the file system's clients. The slaves in the HDFS architecture have Data Nodes (usually one per resource in the cluster), which manage the storage process of their resource. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of Data Nodes. Typical block sizes used by HDFS are 64 or 128 MB.

  A user or an application can create directories and store files inside these directories. The Name Node maintains the file system namespace, and records any change to the file system namespace or its properties. It also executes the operations of the file system namespace such as opening, closing, and renaming files and directories. Moreover, it determines the mapping of blocks to Data Nodes. The Data Nodes are responsible for serving read and write requests from the file system's clients. They also perform block creation, deletion, and replication upon instruction from the Name Node.

  The HDFS is designed to reliably store very large files across resources in a large cluster. The blocks of a file are replicated for fault tolerance. The replication factor is specified at

file creation time and can be changed later. The Name Node makes all decisions regarding replication of blocks. It periodically receives a heartbeat and a blockreport from each Data Node in the cluster. Receiving a heartbeat from a Data Node implies that it is functioning properly. A blockreport contains a list of all blocks on a Data Node.

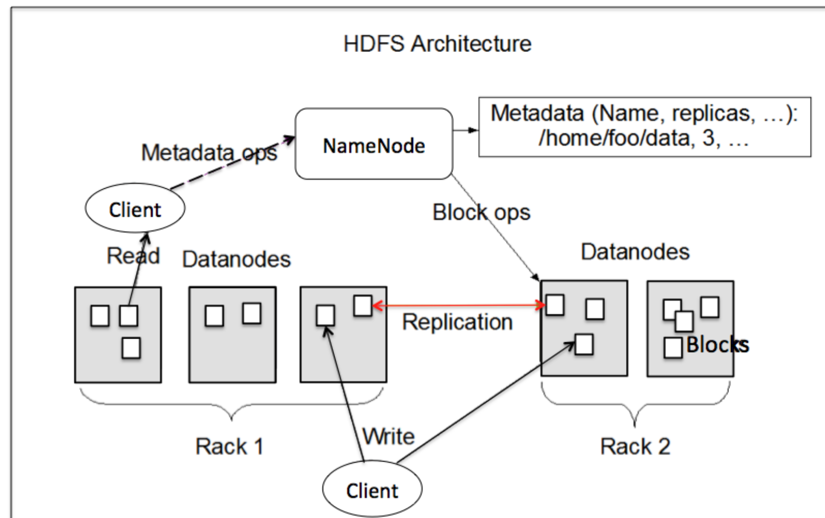


Figure 5.4. The HDFS Architecture (Apache Hadoop Foundation [2010d]).

Figure 5.4 presents more details of the HDFS architecture. The architecture does not preclude running multiple Data Nodes on the same resource, but in reality this is rarely done. The existence of a single Name Node in a cluster greatly simplifies the system architecture. The Name Node is the arbitrator and repository for all HDFS metadata. The system is designed in a way that user data never flows through the Name Node. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but read it one or more times and require these reads to be performed at streaming speeds.

5.4 Hadoop Execution Process

Following the two layers of the Hadoop architecture, the execution process is defined in two layers. In the MapReduce layer, the execution process can be divided into two parts, namely, a Map section and a Reduce section.

1. **Map execution process:** assigns each map task to a portion of the input file called a slice. By default, each slice contains a single HDFS block, and the total number of file blocks normally determines the number of map tasks. Execution of a map tasks consists of the following steps:

(a) The task's slice is read and organized into records (<key, value> pairs), and the map function is applied to each record.

(b) After the map function's completion, the commit phase registers the final output with the Task Tracker, which informs the Job Tracker about the task's completion.

(c) The Output Collector stores the map output in a format that is easy for the reduce tasks to consume. Intermediate keys are assigned to reduce tasks by applying a partitioning function. Thus, the Output Collector applies this function to each key produced by the map function, and stores each record and partition number in an in-memory buffer.

(d) The Output Collector spills this information to the disks when a buffer reaches its capacity. A spill of the in-memory buffer involves sorting the records first by partition number, and second by key values. The buffer content is written to a local file system as a data file and an index file. The index file points to the offset of each partition in the data file. The data file contains the records, which are sorted by the key within each partition segment.

(e) During the commit phase, the final output of a map task is generated by merging all the spill files produced by this map task into a single pair of data and index files. These files are registered with the Task Tracker before the task is completed. The Task Tracker reads these files to service requests from reduce tasks.

2. **Reduce execution process:** contains three steps: shuffle, sort, and reduce.

(a) In the shuffle step, the intermediate data generated by the map phase is fetched. Each reduce task is assigned a partition of the intermediate data with a fixed key range; so the reduce task must fetch the content of this partition from every map task's output in the cluster.

(b) In the sort step, records with the same key are grouped together to be processed by the reduce step.

(c) In the reduce step, the user defined reduce function is applied to each key and corresponding list of values.

The Job Tracker has the information about the location of Task Trackers which store the map outputs, and provides this information to the Task Trackers which execute the reduce tasks. A reduce task can not fetch the output of a map task until the map task has finished its execution and committed its final output to the disk. After receiving partitions from all of the map tasks' outputs, the reduce task enters the sort step. The output generated from

map tasks for each partition is already sorted by the reduce key. The reduce task merges these runs together to produce a single run that is sorted by key. The task then enters the last reduce step, in which the user-defined reduce function is invoked for each distinct key in a sorted order, passing it the associated list of values. The output of the reduce function is written to a temporary location on the HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is automatically changed from its temporary location to its final location.

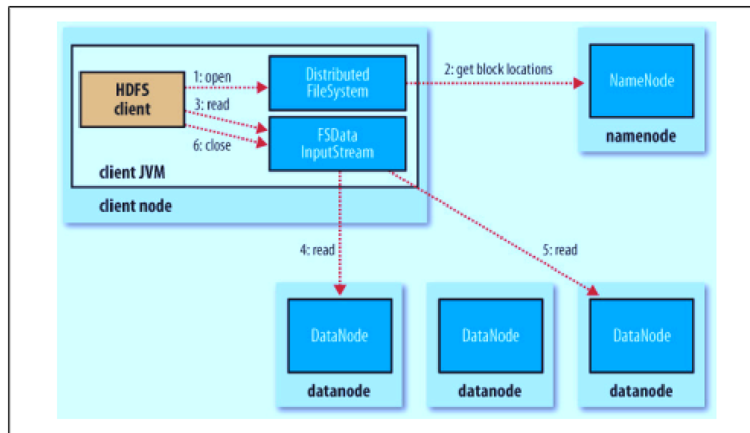


Figure 5.5. A client reading data from the HDFS (Venner [2009])

The execution process in the HDFS layer consists of two main functions: file read and write as follows:

1. **File Reading Process**: includes the following steps (Figure 5.5):

   (a) The client opens the file by calling open() on the DistributedFileSystem.

   (b) The DistributedFileSystem calls the Name Node to determine the locations of the first few blocks of the file. For each block the Name Node returns the addresses of the Data nodes which have a copy of that block.

   (c) The DistributedFileSystem returns an FSDataInputStream to the client from which to read data. An FSDataInputStream wraps a DFSInputStream, which manages the Data Node and Name Node I/O.

   (d) The client calls read() on the stream DFSInputStream, which has stored the Data Node addresses for the first few blocks in the file. It connects to the first (closest) Data Node for the first block in the file. Consequently, data is streamed from the Data Node to the client, which calls read() repeatedly on the stream.

(e) When the end of the block is reached, DFSInputStream will terminate the connection with the Data Node, and find the best Data Node for the next block. This happens transparently to the client.

(f) Finally, when the client has finished reading, DFSInputStream calls close() on the FSDataInputStream.

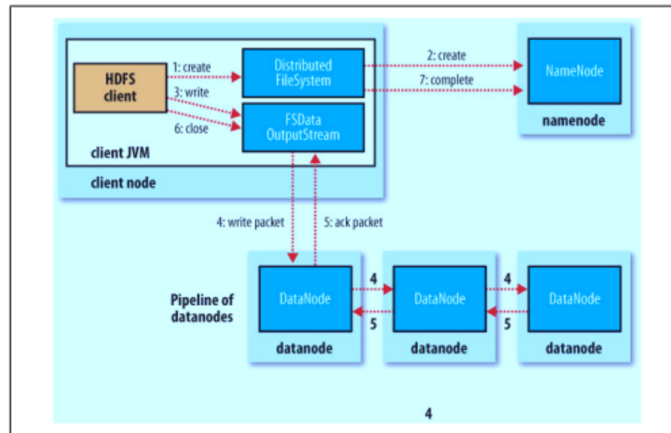


Figure 5.6. A client writing data to the HDFS (Venner [2009])

2. **File Writing Process**: has the following steps (Figure 5.6):

(a) The client creates a file by calling create() on DistributedFileSystem.

(b) The DistributedFileSystem makes an RPC (Remote Procedure Call) to the Name Node to create a new file with no blocks associated with it in the filesystem's namespace. Name Node performs various checks; if all checks pass, DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to it.

(c) FSDataOutputStream wraps a DFSOutputStream, which handles communication with the Data Nodes and the Name Node.

(d) The client writes data. DFSOutputStream splits data into packets, which it then writes to an internal queue, called the data queue. The data queue is consumed by the DataStreamer.

(e) Consequently, the DataStreamer streams the packets to the first Data Node in the pipeline, which stores the packets and forwards them to the second Data Node in the pipeline. The second Data Node stores the packets and forwards them to the third (and last) Data Node.

(f) The DFSOutputStream maintains an internal queue (called the ACK queue) of packets that are waiting to be acknowledged by Data Nodes. A packet is removed from the ACK queue only when it has been acknowledged by all the Data Nodes in the pipeline.

(g) When the client has finished writing, it calls the close() function on the stream. This action flushes all of the remaining packets to the Data Node pipeline and waits for acknowledgments.

(h) Finally, the client contacts the Name Node to signal the file's completion.

5.5 Conclusion

This chapter provided an introduction to the MapReduce programming model and Hadoop system. The following chapters of this thesis address the scheduling problem in Hadoop systems, and propose new Hadoop schedulers. The proposed schedulers will be an extension to Hadoop's MapReduce layer, and its Job Tracker daemon.

Chapter 6

COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems [1]

Hadoop systems were initially designed to optimize the performance of large batch jobs such as web index construction (Zaharia et al. [2010]). However, due to the increasing number of applications running on Hadoop, there is a growing demand for sharing Hadoop clusters amongst multiple users (Zaharia et al. [2010]). Various types of applications submitted by different users require the consideration of new aspects in designing a scheduling system for Hadoop. One of the most important aspects which should be considered is heterogeneity in the system. Heterogeneity can be at both the application and the cluster levels. Application level heterogeneity is taken into account in some recent work on Hadoop schedulers (Ghodsi et al. [2011]). However, to the best of our knowledge, cluster level heterogeneity is a neglected aspect in designing Hadoop schedulers. This PhD thesis introduces a new scheduling system designed and implemented for Hadoop, which considers heterogeneity at both application and cluster levels.

From the user perspective, there are two critical issues. First, Hadoop guarantees a minimum share of resources for every user at any time. Therefore, satisfying the minimum shares of all users is the first key point in designing a Hadoop scheduler. Second, the resources should be divided among the users in a fair way (to prevent starvation of any user). Unlike most existing Hadoop schedulers (Zaharia et al. [2010]), COSHH makes scheduling decisions based on the system

[1] This chapter is mostly based on the following papers:

1. A. Rasooli and D. G. Down, *COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop systems*. **Future Generation Computer Systems (FGCS)**, 2012, (Submitted).
2. A. Rasooli and D. G. Down, *An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems*, **In Proceeding of the 21st Annual International Conference hosted by the Centre for Advanced Studies Research, IBM Canada (CASCON 2011)**, November 7-10, 2011, Toronto, Canada.

heterogeneity, while considering fairness and minimum share objectives. COSHH has two stages to consider these issues. In the first stage, the algorithm aims to satisfy the minimum share requirements, and in the second stage, it considers fairness over all users.

The main approach in the proposed scheduler is to use system information to make better scheduling decisions, which leads to improved performance. To gather this information, COSHH employs a component which was first introduced in (Morton et al. [2010]) and is further developed in (Agarwal and Ananthanarayanan [2010]). This component estimates the job mean execution time based on the job's structure, and the number of map and reduce tasks in each job.

Using the system information, COSHH classifies the jobs, and finds the appropriate job class and resource matchings based on the job classes requirements and resource features. Then, COSHH uses the job class-resource matching as well as user priority, required minimum share, and fair share to make scheduling decisions. The proposed scheduler is dynamic, and updates its decisions based on changes in the system parameters.

The COSHH scheduler is intended mainly for Hadoop systems with large workloads, where the number of jobs waiting in the queue at scheduling instants is large. In such a system, an exhaustive search to find an appropriate matching of jobs and resources can result in significant overhead for the scheduler. However, the proposed scheduler reduces the search space by using a classification approach. Moreover, the classification and optimization approaches in the COSHH scheduler provide suggestions for the location of data replicas. The suggested replication resources could lead to considerable improvement in the locality metric, in particular for large Hadoop clusters.

In this chapter, a Hadoop simulator, MRSIM (Hammoud et al. [2010]), is extended to evaluate the proposed scheduler, and four of the most common Hadoop performance metrics: locality, fairness, minimum share satisfaction, and average completion time, are implemented. The performance of COSHH is compared with two commonly used Hadoop schedulers, the FIFO and the Fair Sharing schedulers (Zaharia et al. [2009]). Moreover, the sensitivity of the proposed scheduler to errors in the estimated job execution times is examined.

The remainder of this chapter is organized as follows. Section 6.1 provides motivation for the proposed scheduler. The high level architecture of COSHH is introduced in Section 6.2. Details of the two main components in the proposed scheduler are presented in Sections 6.3 and 6.4. Section 6.5 introduces the Hadoop performance metrics. In Sections 6.6 and 6.7, the performance of COSHH is studied using synthetic and two well-known real Hadoop workloads, respectively. Section 6.8 provides further discussion about the performance of the COSHH scheduler. Sensitivity analyses are presented in Section 6.9. Current Hadoop scheduling algorithms are given in Section 6.10. Finally, the last section concludes the chapter.

6.1 Motivation

The following discusses a number of observations and motivations to develop a Hadoop scheduler for a heterogeneous environment.

- **Multi-factor Hadoop scheduler.** In a Hadoop system, satisfying the minimum shares of users is the first critical issue. The next important issue is fairness. Traditional Hadoop schedulers address the fairness and the minimum share objectives without considering heterogeneity of jobs and resources. One of the advantages of the COSHH scheduler is that while it addresses the fairness and the minimum share requirements, it makes efficient assignments by considering the heterogeneity. The system heterogeneity is defined based on job requirements (such as estimated mean execution time) and resource features (such as execution rate). Consequently, the proposed scheduler typically reduces average completion times.

- **Reducing the communication cost.** The Hadoop system distributes tasks on multiple resources in parallel to reduce completion times. However, communication costs are not considered in the task distribution process of Hadoop. In a large cluster with heterogenous resources, maximizing a task's distribution may result in overwhelmingly large communication overhead. As a result, job completion times will be increased. COSHH considers the heterogeneity and distribution of resources in the task assignment.

- **Reducing the search overhead for matching jobs and resources.** To find the best matching of jobs and resources in a heterogeneous Hadoop system, an exhaustive search is required. COSHH uses classification and optimization techniques to restrict the search space. Jobs are categorized based on their requirements. When a resource is available, the COSHH scheduler searches through the classes instead of the individual jobs to find the best matching (using optimization techniques). The solution of the optimization problem provides a set of suggested classes for each resource, which is used for making routing decisions. Moreover, to prevent potentially large overheads, COSHH limits the number of times that classification and optimization are performed in the scheduler.

- **Increasing locality.** If the scheduler increases the probability of assigning tasks to the resources which store the corresponding input data, locality can be increased. COSHH makes scheduling decisions based on the suggested set of job classes for each resource. Therefore, the required data of the suggested classes for a resource can be replicated on that resource. This can lead to increasing locality, in particular in large Hadoop clusters, where locality is more critical.

6.2 Proposed Hadoop Scheduling System

The high level architecture of COSHH is presented in Figure 6.1. This section presents a brief overview of the components in the proposed scheduler. Further details of the main components are provided in the following two sections.

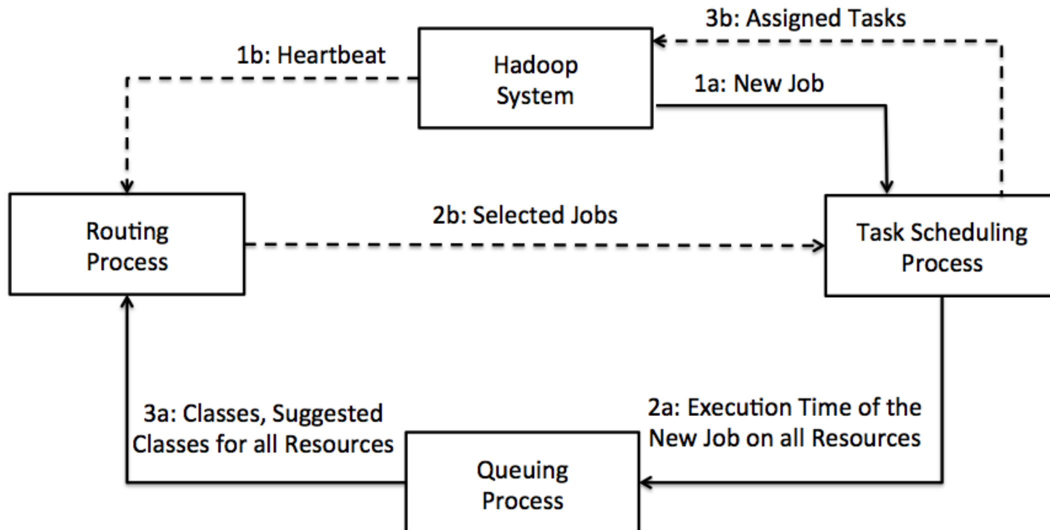


Figure 6.1. The high level architecture of COSHH

A typical Hadoop scheduler receives two main messages from the Hadoop system: a message signalling a new job arrival from a user, and a heartbeat message from a free resource. Therefore, COSHH consists of two main processes, where each process is triggered by receiving one of these messages. Upon receiving a new job, the scheduler performs the queuing process to store the incoming job in an appropriate queue. When a heartbeat message is received, the scheduler triggers the routing process to assign a job to the current free resource. In Figure 6.1, the flows of the job arrival and heartbeat messages are presented by solid and dashed lines, respectively.

The high level architecture of COSHH consists of four components: the Hadoop system, the task scheduling process, the queuing process, and the routing process. The task scheduling process estimates the mean execution time of an incoming job on all resources. These estimates are passed to the queuing process to choose an appropriate queue for the incoming jobs. The routing process selects a job for the available free resource, and sends it to the task scheduling process. Using the selected job's characteristics, the task scheduling process assigns tasks of the selected job to available slots of the free resource. The Hadoop system and task scheduling process are introduced in this section, and the detailed description of the queuing and routing processes, which are the main contributions of this chapter, are discussed in Sections 6.3 and 6.4, respectively.

6.2.1 Hadoop System Model

The Hadoop system consists of a cluster, which is a group of linked resources. The data in the Hadoop system is organized into files. The users submit jobs to the system, where each job consists of a number of tasks. Each task is either a *map task* or a *reduce task*. The Hadoop components are described as follows:

1. The cluster consists of a set of resources, where each resource has a computation unit, and a data storage unit. The computation unit consists of a set of slots (in most Hadoop systems, each CPU core is considered as one slot), and the data storage unit has a specific capacity. A cluster of M resources is defined as as follows:

$$Cluster = \{R_1, \dots, R_M\}$$

$$R_j =< Slots_j, Mem_j >$$

   - $Slots_j$ is the set of slots in resource R_j, where each slot ($slot_j^k$) has a specific execution rate ($exec_rate_j^k$). Generally, slots belonging to one resource have the same execution rate. A resource R_j has the following set of s_j slots:

$$Slots_j = \{slot_j^1, \dots, slot_j^{s_j}\}$$

   - Mem_j is the storage unit of resource R_j, which has a specific capacity ($capacity_j$) and data retrieval rate ($retrieval_rate_j$). The data retrieval rate of resource R_j depends on the bandwidth within its storage unit.

2. Data in the Hadoop system is organized into files, which are usually large. Each file is split into small pieces, called slices (usually, all slices in a system have the same size). Assuming that there are f files in the system, and each file is divided into l_i slices, the files are defined as follows:

$$Files = \{F_1, \dots, F_f\}$$

$$F_i = \{slice_i^1, \dots, slice_i^{l_i}\}$$

3. It is assumed that there are Z users in the Hadoop system, where each user (U_i) submits a set of jobs to the system ($Jobs_i$) as follows:

$$Users = \{U_1, \ldots, U_Z\}$$

$$U_i =< Jobs_i >$$

$$Jobs_i = \{J_i^1, \ldots, J_i^{n_i}\},$$

where J_i^d denotes job d submitted by user U_i, and n_i is the total number of jobs submitted by this user. The Hadoop system assigns a priority and a minimum share to each user based on a particular policy (e.g. the pricing policy of (Sandholm and Lai [2010])).

The priority is an integer which shows the relative importance of a user. Based on the priority ($priority_i$) of a user U_i, its corresponding weight ($weight_i$) is defined, where the weight can be any integer or fractional number. The number of slots assigned to user U_i depends on their weight ($weight_i$). The minimum share of a user U_i (min_share_i) is the minimum number of slots that the system has guaranteed to provide for user U_i at each point in time.

In a Hadoop system, the set of submitted jobs of a user is dynamic, meaning that the set of submitted jobs for user U_i at time t_1 may be completely different than at time t_2. A job J_i is represented by

$$J_i = Maps_i \cup Reds_i,$$

where $Maps_i$ and $Reds_i$ are the sets of *map task*s and *reduce task*s of this job, respectively. The set $Maps_i$ of job J_i is denoted by

$$Maps_i = \{MT_i^1, \ldots, MT_i^{m'_i}\}.$$

Here, m'_i is the total number of map tasks job in J_i, and MT_i^k is *map task k* of job J_i. Each map task MT_i^k performs some processing on the slice ($slice_j^l \in F_j$), where the required data for this task is located. The set $Reds_i$ of job J_i is denoted by

$$Reds_i = \{RT_i^1, \ldots, RT_i^{r'_i}\}.$$

Here, r'_i is the total number of reduce tasks of job J_i, and RT_i^k is *reduce task k* of job J_i. Each reduce task RT_i^k receives and processes the results of some of the *map task*s of job J_i. The value $mean_execTime(J_i, R_j)$ defines the mean execution time of job J_i on resource R_j, and the corresponding execution rate is defined as follows:

$$mean_execRate(J_i, R_j) = 1/mean_execTime(J_i, R_j).$$

6.2.2 Task Scheduling Process

Upon a new job arrival, an estimate of its mean execution times on the resources is required. This component is a result of research in the AMP lab at UC Berkeley (Agarwal and Ananthanarayanan [2010]). The task scheduling process component uses the *Chronos* task duration predictor to estimate the mean execution times of the incoming job on all resources ($mean_execTime(J_i, R_j)$). The *Chronos* prediction algorithm is described briefly here, more details can be found in (Agarwal and Ananthanarayanan [2010]). To define the prediction algorithm, first various analyses are performed in (Agarwal and Ananthanarayanan [2010]), to identify important log signals. Then, the prediction algorithm is introduced using these log signals, and finally the accuracy of the prediction algorithm is evaluated on real Hadoop workloads.

To analyze the correlation between task execution times and various log signals, analyses were performed on 46 GB of logs from Yahoo! and Facebook clusters comprising of around 2000 machines and 3000 resources, respectively. Using Pearson's correlation coefficients (Benesty et al. [2009]), it was determined that task execution times showed a high positive correlation with HDFS bytes read, HDFS bytes written, and map input bytes. Moreover, there was a relatively low but positive correlation with combine input records, local bytes read, local bytes written, map output bytes, map input records, and map output records. Two other important signals were determined to be the cluster utilization intervals and input file name. The intuition behind including these signals was that generally tasks operating on the same file and in similar utilization intervals tend to have similar execution times.

Almost every machine learning based prediction algorithm is a trade-off between speed and accuracy (Mair et al. [2000]). The prediction algorithm used for Hadoop systems is required to be extremely fast and fairly accurate. This is because the prediction algorithm is invoked each time a job arrives to the system. The analysis performed on various logs from Facebook and Yahoo! Hadoop clusters shows that at any time, there can be anywhere between 0 to 3000 tasks running on the cluster. This requires that the prediction scheduler should be able to make a decision within a matter of microseconds, with fairly high accuracy. To achieve this goal, *Chronos* consists of two parts: the first part, *chrond*, refers to the *Chronos* daemon that runs in the background. It is responsible for analyzing Hadoop history log files as well as monitoring cluster utilizations every specified time interval. For example, for Facebook and Yahoo! workloads, an interval of every six hours could provide the desired accuracy (Mair et al. [2000]).

Algorithm 1 Refined Predictor for COSHH

```
When a new Job J arrives:
  for Each resource r in the system do
    Get the execution rate r_i of slots in r
    Classify r_i into R_c
    for Each map task t_i of J do
      Get the following information:
        c_i: combine input records t_i requires to read
        h_i: HDFS bytes t_i requires to read
        f_i: the input file of t_i
      Classify (c_i, h_i) into Cluster L_c;
      Construct S= Set of all task durations which operate on f_i, and belong to Clusters L_c, R_c;
      if S = ∅ then
        Construct S= Set of all task durations which operate on f_i, and belong to Cluster R_c;
      end if
      if S = ∅ then
        Construct S= Set of all task durations which operate on f_i;
      end if
      if S = ∅ then
        return −1;
      end if
      if S ≠ ∅ then
        use median(S) as execution time of t_i on each slot
      end if
    end for
    Using:
     Execution time of each task of J on each slot in r
     Number and order of all tasks in J
     Number of slots in r
    Compute the execution time of J on r
  end for
  return the execution time of J on all resources
```

Periodically, *Chronos* applies k-means clustering (Ethem [2004]) on this data and keeps track of the cluster boundaries. On the other hand, the second part, *Predictor*, is an on-the-spot decision engine. Whenever a new job arrives, the *Predictor* classifies its tasks into various categories depending on the file they operate on, the total cluster utilization at that point in time, and the input bytes they read, by consulting the lookup table populated by *chrond*. Finally, it returns the median task durations. The refined *Predictor* algorithm for COSHH is provided in Algorithm 1.

Accuracy experiments for *Chronos* are provided in (Agarwal and Ananthanarayanan [2010]) on 46 GB of Yahoo! and Facebook cluster logs. The results confirm that around 90% of map tasks could be predicted within 80% accuracy. Further, about 80% of reduce tasks are predicted within 80% accuracy. Most importantly, the addition of *Chronos* to the existing Hadoop Delay Scheduler did not result in any significant performance degradation (Agarwal and Ananthanarayanan [2010]).

6.3 Queuing Process

Figure 6.2 shows different stages of the queuing process. The two main components in the queuing process are a classifier (Section 6.3.1), and an optimizer (Section 6.3.2). At a high level, when a new job arrives, the classification approach specifies the job class, and stores the job in the corresponding queue. If the job does not fit in any of the current classes, the list of classes is updated with a new class for the incoming job. The optimizer finds an appropriate matching of job classes and resources, by solving an optimization problem that is defined based on the properties of the job classes and features of the resources. The result of the queuing process, which is sent to the routing process, contains the list of job classes and the suggested set of classes for each resource. The classifier and optimizer methods used in COSHH can reduce the search space in finding an appropriate matching of resources and jobs. Moreover, these methods are used to consider the system heterogeneity in the proposed scheduler, which leads to reducing the completion times. However, the classification and optimization processes can add overhead to the scheduling process. To prevent a large overhead, the proposed scheduler limits the number of times that these steps are performed, as well as using efficient classification and optimization methods with low overheads.

6.3.1 COSHH Classification

Investigations on real Hadoop workloads show that it is possible to determine classes of "common jobs" (Chen et al. [2011]). The well-known k-means clustering method is used for classifying jobs in real Hadoop workloads (Chen et al. [2011]). Accordingly, the proposed COSHH scheduler uses this method in its classifier.

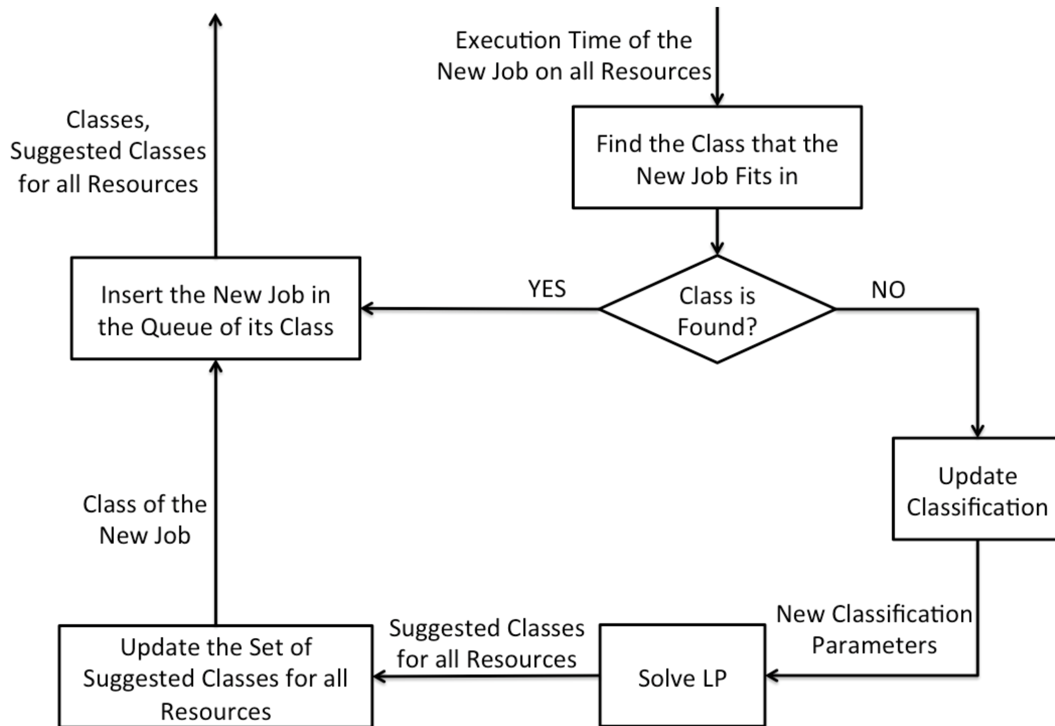


Figure 6.2. The Queuing Process

This thesis designed a Hadoop scheduler based on the fact that there are two critical criteria with different levels of importance in the Hadoop system. The first criterion, imposed by the Hadoop provider is satisfying the minimum shares. The Hadoop providers guarantee that a user's minimum share will be provided at all points in time while their jobs are executing. The second criterion, important to improve the overall system performance, is fairness. Considering fairness prevents starvation of any user, and divides the resources among the users in a fair way. Minimum share satisfaction has higher criticality than fairness. Therefore, COSHH has two classifications, to consider these issues for first minimum share satisfaction, and then for fairness. The primary classification (for minimum share satisfaction), targets only jobs with $min_share > 0$, while in the secondary classification (for fairness) all of the jobs in the system are considered. As a result, jobs with $min_share > 0$ are considered in both classifications. This is due to the fact that when a user asks for more than her minimum share, first her minimum share is given immediately through the primary classification. Then, the extra share should be given in a fair way by considering all users through the secondary classification.

In both classifications, jobs are classified based on their features (i.e, priority, mean execution rate on the resources ($mean_execRate(J_i, R_j)$), and mean arrival rate). The set of classes generated in the primary classification is defined as $JobClasses1$, where an individual class is denoted

by C_i. Each class C_i has a given priority, which is equal to the priority of the jobs in this class. The estimated mean arrival rate of jobs in class C_i is denoted by σ_i, and the estimated mean execution rate of the jobs in class C_i on resource R_j is denoted by $\psi_{i,j}$. Hence, the heterogeneity of resources is completely addressed with $\psi_{i,j}$. The total number of classes generated with this classification is assumed to be B, i.e.

$$JobClasses1 = \{C_1, \ldots, C_B\}.$$

The secondary classification generates a set of classes defined as $JobClasses2$. Similar to the priority classification, each secondary class, denoted by C'_i, has priority equal to the priority of the jobs in this class. The mean arrival rate of the jobs in class C'_i is equal to σ'_i, and the mean execution rate of jobs in class C'_i on resource R_j is denoted by $\psi'_{i,j}$. The total number of classes generated with this classification is B', i.e.

$$JobClasses2 = \{C'_1, \ldots, C'_{B'}\}.$$

As an example of a system with multiple classes, Yahoo! uses the Hadoop system in production for a variety of products (job types) (Bodkin [2010]): Data Analytics, Content Optimization, Yahoo! Mail Anti-Spam, Ad Products, and several other applications.

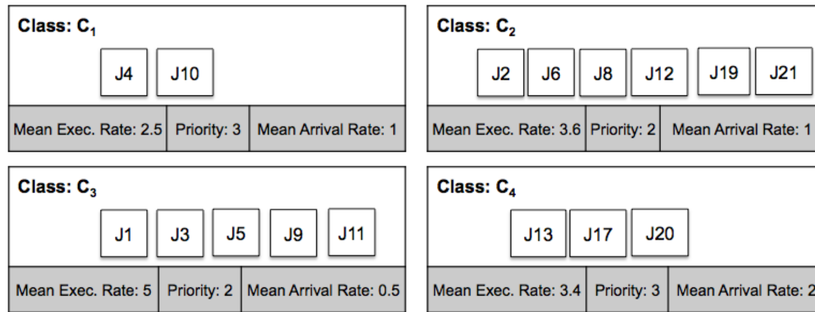


Figure 6.3. The primary classification of the jobs in Exp1 system at time t

User	Job Type	min_share	priority
User1	Advertisement Products	50	3
User2	Data Analytics	20	2
User3	Advertisement Targeting	40	3
User4	Search Ranking	30	2
User5	Yahoo! Mail Anti-Spam	0	1
User6	User Interest Prediction	0	2

Table 6.1. The Hadoop System Example (Exp1)

User	Job Queue
User1	$\{J_4, J_{10}, J_{13}, J_{17}\}$
User2	$\{J_1, J_5, J_9, J_{12}, J_{18}\}$
User3	$\{J_2, J_8, J_{20}\}$
User4	$\{J_6, J_{14}, J_{16}, J_{21}\}$
User5	$\{J_7, J_{15}\}$
User6	$\{J_3, J_{11}, J_{19}\}$

Table 6.2. The job queues in Exp1 at time t

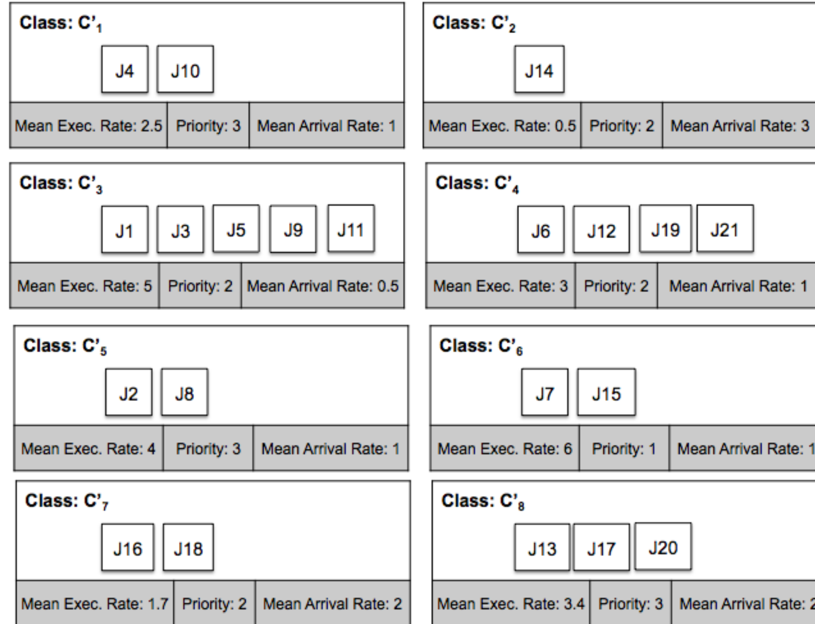


Figure 6.4. The secondary classification of the jobs in Exp1 system at time t

Typically, the Hadoop system defines a user for each job type, and the system assigns a minimum share and a priority to each user. For example, assume a Hadoop system (called Exp1) with the parameters in Table 6.1. The corresponding jobs at a given time t are given in Table 6.2, where the submitted jobs of a user are based on the user's job type (e.g., J_4, submitted by User1, is an advertisement product job, while job J_5 is a search ranking job). The primary classification of the jobs in the Exp1 system at time t is presented in Figure 6.3. Note that this example has one resource in the system. The secondary classification of system Exp1 at time t is shown in Figure 6.4.

6.3.2 COSHH Optimization

After classifying the incoming jobs, and storing them in their appropriate classes, the scheduler finds a matching of job classes and resources. The optimization approach used in the proposed

scheduler first constructs an LP which considers properties of the job classes and features of the resources. The scheduler then solves this LP to find a set of suggested classes for each resource.

An LP is defined for each of the two classifications. The first LP is defined for classes in the set $JobClasses1$ as follows:

$$\max \gamma$$

$$s.t. \sum_{j=1}^{M} \psi_{i,j} \times \theta_{i,j} \geq \gamma \times \sigma_i, \ for\ all\ i = 1, \ldots, B, \quad (6.1)$$

$$\sum_{i=1}^{B} \theta_{i,j} \leq 1, \ for\ all\ j = 1, \ldots, M, \quad (6.2)$$

$$\theta_{i,j} \geq 0, \ for\ all\ i = 1, \ldots, B, \ and\ j = 1, \ldots, M. \quad (6.3)$$

Here γ is interpreted as the maximum capacity of the system, and $\theta_{i,j}$ is the proportion of resource R_j allocated to class C_i. As a reminder, M is the total number of resources, and B is the total number of classes generated in the primary classification (i.e, $|JobClasses1|$). This optimization problem increases the arrival rates of all the classes by a fixed proportion, to minimize the load in the system, while constraint (6.1) keeps the system stable. The solution of this LP provides the allocation matrix θ, whose (i, j) element is $\theta_{i,j}$. Based on the results of this LP, the set SC_j is defined for each resource R_j as

$$SC_j = \{C_i : \theta_{i,j} \neq 0\}.$$

For example, consider a system with two classes of jobs, and two resources ($M = 2$, $B = 2$), in which the arrival and execution rates are $\sigma = \begin{bmatrix} 2.45 & 2.45 \end{bmatrix}$ and $\psi = \begin{bmatrix} 9 & 5 \\ 2 & 1 \end{bmatrix}$, respectively. Solving the above LP gives $\gamma = 1.0204$ and $\theta = \begin{bmatrix} 0 & 0.5 \\ 1 & 0.5 \end{bmatrix}$. Therefore, the sets SC_1 and SC_2 for resources R_1 and R_2 will be $\{C_2\}$ and $\{C_1, C_2\}$, respectively. These two sets define the suggested classes for each resource, i.e. upon receiving a heartbeat from resource R_1, a job from class C_2 should be selected. However, upon receiving a heartbeat from resource R_2, either a job from class C_1 or C_2 should be chosen. Even though resource R_1 has the fastest rate for class C_1, the algorithm does not assign any jobs of class C_1 to it. It is observed in the experiments that when the system is highly loaded, the average job completion time decreases if resource R_1 is dedicated to jobs in class C_2.

The second optimization problem is used for the secondary classification. The scheduler defines an LP similar to the previous one, for classes in the set $JobClasses2$. In this LP the parameters γ, $\psi_{i,j}$, $\theta_{i,j}$, σ_i, and B are replaced by γ', $\psi'_{i,j}$, $\theta'_{i,j}$, σ'_i, and B', respectively:

$$\max \gamma'$$

$$s.t. \sum_{j=1}^{M} \psi'_{i,j} \times \theta'_{i,j} \geq \gamma' \times \sigma'_i, \ for\ all\ i = 1, \ldots, B', \tag{6.4}$$

$$\sum_{i=1}^{B'} \theta'_{i,j} \leq 1, \ for\ all\ j = 1, \ldots, M, \tag{6.5}$$

$$\theta'_{i,j} \geq 0, \ for\ all\ i = 1, \ldots, B', \ and\ j = 1, \ldots, M. \tag{6.6}$$

The solution of this LP yields the matrix θ', whose (i, j) element is $\theta'_{i,j}$. The set SC'_j is defined for each resource R_j as the set of classes allocated to this resource, where $SC'_j = \{C'_i : \theta'_{i,j} \neq 0\}$.

The COSHH scheduler uses the sets of suggested classes SC_R and SC'_R for both making scheduling decisions and improving locality in the Hadoop system. The scheduling decision is made by the routing process, and locality can be improved by replicating input data on multiple resources in the Hadoop system. Most current Hadoop schedulers randomly choose three resources for replication of each input data (Zaharia et al. [2009, 2010]). However, COSHH uses the sets of suggested classes, SC_R and SC'_R, to choose replication resources. For each input data, the initial incoming jobs using this data are considered, and from all the suggested resources for these jobs, three of them are randomly selected for storing replicas of the corresponding input data. As this research evaluates the proposed scheduler on small Hadoop clusters, only the initial incoming jobs

Algorithm 2 Queuing Process

When a new Job (say J) arrives
Get execution time of J from Task Scheduling Process
  if J fits in any class (say C_i) **then**
    add J to the queue of C_i
  else
    use k-means clustering to update the job classification
    find a class for J (say C_j) , and add J to its queue
    solve optimization problems, and get two sets of suggested classes, SC_R and SC'_R
  end if
  send SC_R, SC'_R and both sets of classes ($JobClasses1$ and $JobClasses2$) to the routing process

are considered for determining the replication resources. However, in large Hadoop clusters with a high variety of available network bandwidths, developing the proposed replication method to consider the updates caused by later incoming jobs, could lead to significant improvement in the locality. This is left as future work.

The IBM ILOG CPLEX optimizer (IBM ILOG CPLEX Optimizer [2010]) is used to solve the LPs. A key feature of this optimizer is its high performance in delivering the power needed to solve very large optimization problems, and the speed required for highly interactive analytical decision support applications (IBM ILOG CPLEX Optimizer [2010]). As a result, solving the optimization problems in COSHH does not add considerable overhead. The proposed queuing process is presented in Algorithm 2.

6.4 Routing Process

When the scheduler receives a heartbeat message from a free resource, say R_j, it triggers the routing process. The routing process receives the sets of suggested classes SC_R and SC'_R from the queuing process, and uses them to select a job for the current free resource. This process selects a job for each free slot in the resource R_j, and sends the selected job to the task scheduling process. The task scheduling process chooses a task of the selected job, and assigns the task to its corresponding slot.

Here, it should be noted that the scheduler does not limit each job to just one resource. When a job is selected, the task scheduling process assigns a number of appropriate tasks of this job to available slots of the current free resource. If the number of available slots is less than the number of uncompleted tasks in the selected job, the job will remain in the waiting queue. Therefore, at the next heartbeat message from a free resource, this job will be considered in making the scheduling decision; however, tasks already assigned are no longer considered. When all tasks of a job are assigned, the job will be removed from the waiting queue.

Algorithm 3 presents the routing process. There are two stages in this algorithm to select jobs for the available slots of the current free resource. In the first stage, the jobs of classes in SC_R are considered, where the jobs are selected in the order of their minimum share satisfaction. A user who has the highest distance to achieve her minimum share will get a resource share sooner. However, in the second stage, jobs of classes in SC'_R are considered, and the jobs are selected in the order defined by the current shares and priorities of their users. In this way, the scheduler addresses fairness amongst the users. In each stage, ties are broken randomly.

Algorithm 3 Routing Process

When a heartbeat message is received from a resource (say R)
N_{FS} = number of free slots in R

while $N_{FS} \neq 0$ and there is a job (J) whose
$U.min_share - U.currentShare > 0$
and
$class \in SC_R$
and
$((U.min_share - U.currentShare) \times U.weight)$ is maximum **do**
add J to the set of selected jobs ($J_{selected}$)
$N_{FS} = N_{FS} - 1$
end while
while $N_{FS} \neq 0$ and there is a job (J) whose
$class \in SC'_R$
and
$(U.currentShare/weight)$ is minimum
do
add J to the set of selected jobs ($J_{selected}$)
$N_{FS} = N_{FS} - 1$
end while
send the set $J_{selected}$ to the Task Scheduling Process to choose a task for each free slot in R.

6.5 Hadoop Performance Metrics

There is a range of performance metrics that are of interest to both users and Hadoop providers. This section introduces the Hadoop performance metrics considered in this PhD thesis. Two functions are used to define these performance metrics: the function $Demand(U, t)$ returns the set of unassigned tasks for user U at time t; and the function $AssignedSlots(U, t)$ returns the set of slots executing tasks from user U at time t. The run time of an experiment is assumed to be T. Using these definitions, five Hadoop performance metrics are defined as follows:

1. *Average Completion Time*: the average time to complete all submitted jobs.

2. $Minimum\ Share\ Dissatisfaction$: measures to what degree the scheduling algorithm is successful in satisfying the minimum share requirements of the users. A user whose current demand is not zero ($|Demand(U, t)| > 0$), and whose current share is less than her minimum share ($|AssignedSlots(U, t)| < U.min_share$), has the following $UserDissatisfaction$:

$$UserDissatisfaction(U, t) = \frac{U.min_share - |AssignedSlots(U, t)|}{U.min_share} \times U.weight,$$

   where $U.weight$ denotes the weight, and $U.min_share$ is the minimum share of user U. The total distance of all users from their min_share is defined as follows:

$$Dissatisfaction(t) = \sum_{\forall U \in Users} UserDissatisfaction(U, t).$$

3. $Fairness$: measures how fair a scheduler is in dividing the resources among users. A fair scheduler gives the same share of resources to users with equal priority. However, when the priorities are not equal, then the user's share should be proportional to their weight.

   The number of slots assigned to each user beyond her minimum share is computed as $\Delta(U, t) = Assigned\ Slots(U, t) - U.min_share$. The set $Users_w = \{U | U \in Users \wedge U.weight = w\}$ includes all the users with the same weight w. The average additional share of each set $Users_w$ is defined as:

$$avg(w, t) = \frac{\sum_{U \in Users_w} \Delta(U, t)}{|Users_w|}.$$

$\overline{Fairness}(t)$ is computed by the sum of the distances of users in each set $Users_w$ from the average user share ($avg(w,t)$) in the corresponding set:

$$\overline{Fairness}(t) = \sum_{w \in weights} \sum_{U \in Users_w} |\Delta(U,t) - avg(w,t)|.$$

Comparing two algorithms, the algorithm which has lower $\overline{Fairness}(T)$ achieves better performance.

4. *Locality*: is defined as the proportion of tasks which are running on the same resource as where their stored data are located. A *map task* is defined to be local on a resource R if it is running on resource R and its required slice is also stored on resource R. Since the input data size is large, and the *map task*s of one job are required to send their results to the *reduce task*s of that job, the communication cost can be quite significant. Comparing two scheduling algorithms, the algorithm which has larger $Locality(T)$ delivers better performance.

5. *Scheduling Time*: is the total time spent for scheduling all of the incoming jobs. This measures the overhead of each Hadoop scheduler.

6.6 Experimental Results - Synthetic Workload

This section evaluates the proposed scheduler using synthetic Hadoop workloads. First the implemented evaluation environment is described, and later the experimental results are provided.

6.6.1 Experimental Environment

The experimental environment in this section consists of a cluster of six heterogeneous resources with the features presented in Table 6.3. The bandwidth between the resources is 100Mbps. In these experiments an extreme case of heterogeneity is used to identify the boundaries, limitations, and advantages of each algorithm. A moderate case of heterogeneity is used for evaluations on a real Hadoop cluster in Chapter 9. MRSIM (Hammoud et al. [2010]), a MapReduce simulator, is used to simulate a Hadoop cluster and evaluate the proposed scheduler. This simulator is based on discrete event simulation, and accurately models the Hadoop environment.

This thesis extends the MRSIM simulator to measure the five main Hadoop performance metrics defined in Section 6.5. The extended simulator includes scheduler component and a performance

Resources	Slot		Mem	
	$slot\#$	$execRate$	$Capacity$	$RetrieveRate$
R_1	1	$500MHz$	$4GB$	$40Mbps$
R_2	1	$500MHz$	$4TB$	$100Gbps$
R_3	1	$500MHz$	$4TB$	$100Gbps$
R_4	8	$500MHz$	$4GB$	$40Mbps$
R_5	8	$500MHz$	$4GB$	$40Mbps$
R_6	8	$4.2GHz$	$4TB$	$100Gbps$

Table 6.3. Experiment resources

evaluator component (to calculate the desired performance metrics). Moreover, a job submission component is added to the simulator. Using this component it is possible to define various users with different minimum shares and priorities. Each user can submit various types of jobs with different arrival rates. Moreover, a scheduler component is developed in the simulator to receive the incoming jobs and store them in the queues chosen by the scheduler. Also, upon receiving a heartbeat message, the simulated scheduler sends a task to the free slot of the corresponding resource. COSHH implementation details are described in Chapter 10.

The workloads in this section are defined using a Loadgen example job in Hadoop, which is used in the Gridmix (Apache Hadoop Foundation [2010a]) Hadoop benchmark. Loadgen is a configurable job, in which choosing various percentages for keepMap and keepReduce, can make the job equivalent to various workloads used in Gridmix, such as sort and filter. Four types of jobs are generated in the system:

- Small jobs: small I/O and CPU requirements (1 map and 1 reduce task),
- I/O-heavy jobs: large I/O and small CPU requirements (10 map and 1 reduce tasks),
- CPU-heavy jobs: small I/O and large CPU requirements (1 map and 10 reduce tasks), and
- Large jobs: large I/O and large CPU requirements (10 map and 10 reduce tasks).

Using these jobs, three workloads are defined: an I/O-Intensive workload, in which all jobs are I/O-bound; a CPU-Intensive workload; and a mixed workload, which includes all job types. The workloads are given in Table 6.4.

Considering various arrival rates for the jobs in each workload, three benchmarks are defined for each workload in Table 6.5. Here, $BM_{i,j}$ shows the benchmark j of workload i; for instance, $BM_{1,1}$ is a benchmark including I/O-Intensive jobs, where the arrival rate of smaller jobs is higher than the arrival rate of larger jobs. In total, nine benchmarks are defined to run in the simulated Hadoop environment.

Workloads	Workload Type	Jobs Included
W_1	$I/O\text{-}Intensive_i$	$small, I/O\text{-}heavy$
W_2	$CPU\text{-}Intensive_i$	$small, CPU\text{-}heavy$
W_3	$Mixed_i$	$All\ jobs$

Table 6.4. Experimental workloads

Benchmarks	Arrival rate Ordering
$BM_{i,1}$	Smaller jobs have higher arrival rates
$BM_{i,2}$	Arrival rates are equal for all jobs
$BM_{i,3}$	Larger jobs have higher arrival rates

Table 6.5. Experiment benchmarks

Each experiment submits 100 jobs to the system, which is sufficient to contain a variety of the behaviours in a Hadoop system, and is the same number of jobs used in evaluating most Hadoop scheduling systems (Zaharia et al. [2009]). The Hadoop block size is set to 64MB, which is the default size in Hadoop 0.20. In these experiments, job input data sizes are generated similar to the workload used in (Zaharia et al. [2009], Zaharia et al. [2010]), which is driven from a real Hadoop workload. The input data of a job is defined based on the number of map tasks and considering the corresponding data set size (there is one map task per 64MB input block). There are four users, where each user submits a mixture of different job types. The user properties are presented in Table 6.6.

$Users$	$MinimumShare$	$Priority$
U_1	10	1
U_2	30	2
U_3	10	2
U_4	0	1

Table 6.6. User properties

6.6.2 Compared Schedulers

In this thesis, the COSHH scheduler is evaluated from different critical aspects such as heterogeneity, scalability, and the main Hadoop performance metrics. Therefore, the compared schedulers in the evaluations are selected by considering these issues: (1) simplicity, (2) required state information, (3) consideration of the main Hadoop performance metrics, (4) heterogeneity, (5) scalability, and (6) approach with respect to the minimum shares.

There are three schedulers compared in the evaluations: FIFO, Fair Sharing, and COSHH. As these schedulers have completely different approaches in terms of the above six issues, they can

provide an overall picture of the effectiveness of different approaches. The FIFO and the Fair Sharing schedulers are used as the basis of a majority of Hadoop schedulers (Zaharia et al. [2010], Sandholm and Lai [2010], Apache [2007], Ghodsi et al. [2011]). These two schedulers are the most widely used Hadoop schedulers, and are implemented in the default Hadoop package (Apache Hadoop Foundation [2010b]).

FIFO is a simple and fast algorithm provided as the default scheduler in the Hadoop package (Apache Hadoop Foundation [2010b]). It only requires a small amount of state information (only the arrival times of the jobs) to make scheduling decisions. In terms of the Hadoop performance metrics, this scheduler does not take into account the main metrics such as fairness, minimum share satisfaction, and average completion time. FIFO neglects heterogeneity at the user, job and cluster levels. However, it does scale as the number of jobs and resources increases. Due to the simple implementation of this scheduler, it is widely being used in various current Hadoop clusters.

The Fair Sharing scheduler is more complex than the FIFO scheduler, but simpler than the COSHH scheduler. It requires state information about job progress and user status; however, it does not require any information about job execution times. This scheduler considers user heterogeneity, but neglects resource and job heterogeneity. Therefore, it has partial heterogeneity consideration. Fair Sharing considers minimum share satisfaction as its main performance goal. Therefore, it is sensitive to changes in the minimum share settings. This algorithm is the backbone of various other Hadoop schedulers (Zaharia et al. [2010], Apache Hadoop Capacity Scheduler [2010], Wolf et al. [2010], Isard et al. [2009]), and is used in various real Hadoop clusters implemented at companies such as Facebook and Cloudera.

6.6.3 Results and Analysis

The experiments in this section compare the proposed scheduler with the FIFO scheduler and an implemented version of the Fair Sharing scheduler presented in (Zaharia et al. [2009]). The comparison is based on the dissatisfaction, fairness, locality, and average completion time performance metrics. Each experiment was repeated 30 times to construct 95%-confidence intervals.

Figures 6.5, 6.6, and 6.7 present the dissatisfaction metric for the schedulers running the benchmarks of the I/O-Intensive, CPU-Intensive, and Mixed workloads, respectively. The lower and upper bounds of the confidence intervals are presented with lines on each bar.

Based on these results, the proposed scheduler can lead to considerable improvement in the dissatisfaction performance metric. There are a couple of reasons for this improvement. First, the COSHH scheduler considers the minimum share satisfactions of the users as its initial goal. When receiving a heartbeat from a resource, the highest priority user who has not yet received

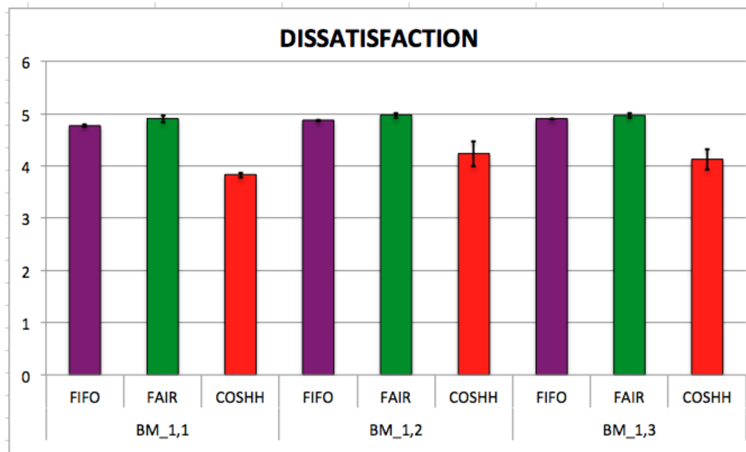


Figure 6.5. Dissatisfaction performance metric of the schedulers for I/O-Intensive workload

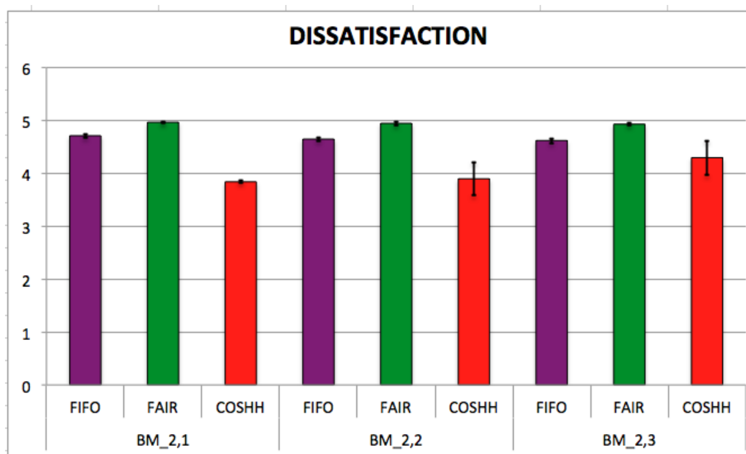


Figure 6.6. Dissatisfaction performance metric of the schedulers for CPU-Intensive workload

her minimum share will be considered first. However, as the scheduler considers the product of the remaining minimum share and the priority of the user, it does not let a high priority user with high minimum share starve lower priority users with smaller minimum shares. Similar to COSHH, the Fair Sharing scheduler has the initial goal of satisfying the minimum shares.

Figures 6.8, 6.9, and 6.10 present the average completion time of the schedulers running the benchmarks of the I/O-Intensive, CPU-Intensive, and Mixed workloads, respectively. The results show the significant improvement in average completion time using the COSHH schedulers in all of the benchmarks. This significant improvement can be explained by the COSHH scheduler considering heterogeneity to make appropriate scheduling decisions.

Table 6.7 presents the fairness metric of the schedulers for the defined benchmarks. Comparing the schedulers, the Fair Sharing algorithm has the best fairness. This is as expected, because the main goal of this algorithm is minimizing the fairness metric. In some benchmarks, the COSHH

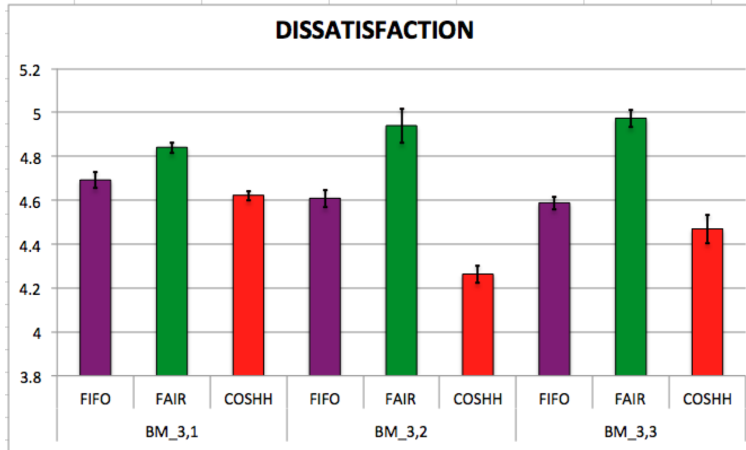


Figure 6.7. Dissatisfaction performance metric of the schedulers for Mixed workload

Benchmarks	FIFO	FAIR	COSHH
$BM_{1,1}$	$(14.88, 15.05)$	$(11.59, 11.65)$	$(14.68, 16.08)$
$BM_{1,2}$	$(14.93, 15.00)$	$(11.57, 11.72)$	$(12.68, 14.60)$
$BM_{1,3}$	$(14.63, 15.26)$	$(11.59, 11.76)$	$(17.23, 17.65)$
$BM_{2,1}$	$(14.77, 15.22)$	$(11.63, 11.98)$	$(11.99, 12.34)$
$BM_{2,2}$	$(14.83, 15.09)$	$(11.81, 12.12)$	$(13.99, 14.36)$
$BM_{2,3}$	$(14.42, 15.73)$	$(11.81, 11.94)$	$(17.37, 17.72)$
$BM_{3,1}$	$(14.94, 15.37)$	$(11.47, 12.71)$	$(14.11, 15.05)$
$BM_{3,2}$	$(14.73, 15.62)$	$(11.72, 12.46)$	$(14.41, 14.98)$
$BM_{3,3}$	$(15.00, 15.44)$	$(11.89, 12.07)$	$(12.11, 13.31)$

Table 6.7. Fairness performance metric of the schedulers for all workloads

scheduler leads to an increase in the fairness metric. However, a small increase in fairness may be considered acceptable for most Hadoop systems, if it results in better satisfaction of the minimum shares, and significant reduction in average completion time.

Table 6.8 presents the locality metric for the defined benchmarks. For each benchmark, the table shows the 95%-confidence interval for locality when the corresponding scheduling algorithm is used. The locality of the proposed scheduler is very competitive with the Fair Sharing scheduler. This can be explained by the fact that the COSHH scheduler chooses the replication locations based on the suggested classes for each resource. Therefore, it tends to assign the job to the same resource, as where its required data is stored.

It should be noted here that although COSHH uses sophisticated approaches, it does not add considerable overhead to the system. This is due to the limited number of times required to perform classification. Moreover, as in a typical Hadoop system, there are several jobs submitted multiple times, the scheduler's classification does not need to change each time that these jobs

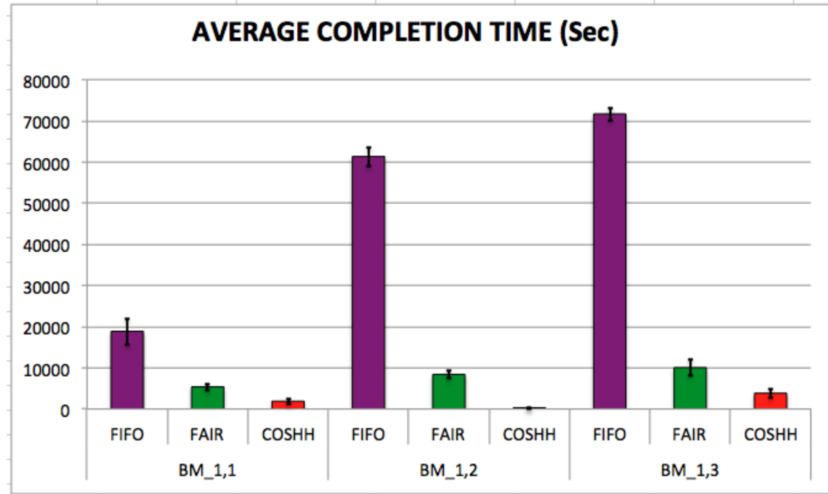


Figure 6.8. Average completion time performance metric of the schedulers for I/O-Intensive workload

Benchmarks	FIFO	FAIR	COSHH
$BM_{1,1}$	$(96.60, 98.03)$	$(98.12, 99.08)$	$(98.62, 99.98)$
$BM_{1,2}$	$(47.39, 57.81)$	$(89.84, 91.76)$	$(93.82, 95.38)$
$BM_{1,3}$	$(62.93, 65.07)$	$(71.43, 74.57)$	$(66.44, 71.55)$
$BM_{2,1}$	$(90.38, 94.42)$	$(97.12, 98.08)$	$(98.56, 99.87)$
$BM_{2,2}$	$(68.65, 82.15)$	$(93.93, 96.87)$	$(91.78, 95.42)$
$BM_{2,3}$	$(78.73, 84.07)$	$(94.14, 97.86)$	$(93.78, 97.42)$
$BM_{3,1}$	$(73.48, 86.92)$	$(78.77, 83.63)$	$(99.12, 100.00)$
$BM_{3,2}$	$(92.36, 95.24)$	$(81.27, 87.13)$	$(95.11, 99.69)$
$BM_{3,3}$	$(79.23, 88.37)$	$(78.02, 86.37)$	$(66.86, 76.73)$

Table 6.8. Locality performance metric of the schedulers for all workloads

are submitted.

6.7 Experimental Results - Real Hadoop Workload

The previous section evaluated the COSHH scheduler under various synthetic computation and I/O intensive workloads. To complete the analysis and confirm the obtained results in the previous section for real Hadoop workloads, this section extends the evaluations using traces of workloads from real Hadoop systems. Various challenges and features of real workloads (such as large sized jobs with high arrival rates) can further evaluate the applicability of the COSHH scheduler for current Hadoop systems.

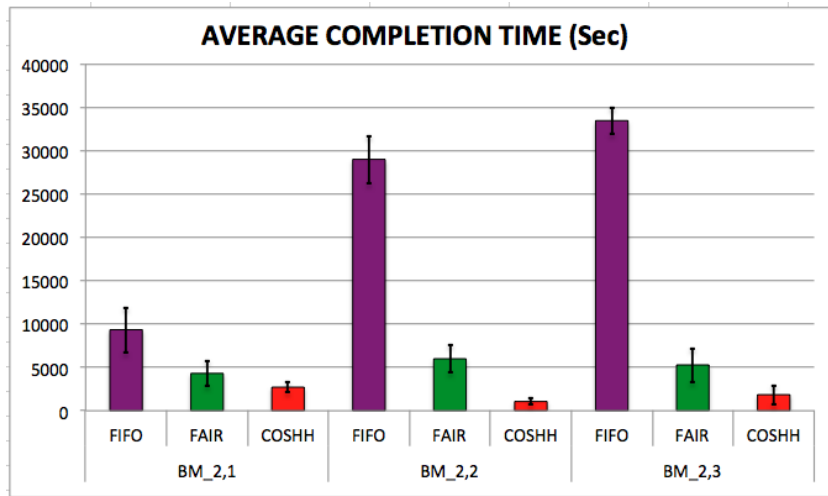


Figure 6.9. Average completion time metric of the schedulers for CPU-Intensive workload

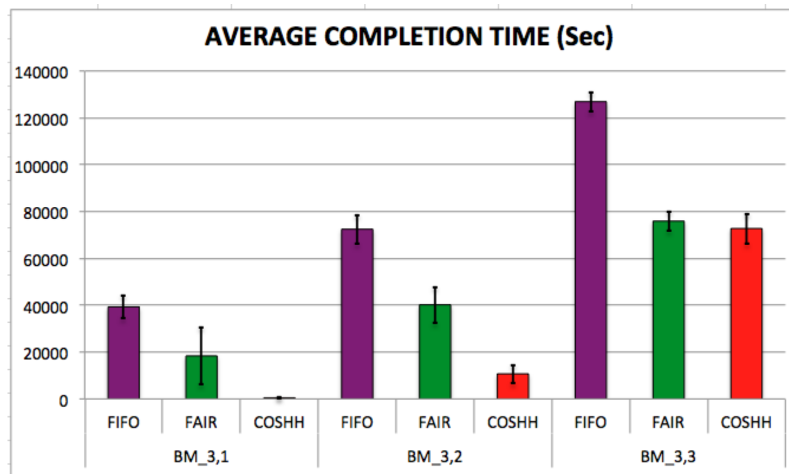


Figure 6.10. Average completion time metric of the schedulers for Mixed workload

6.7.1 Experimental Environment

This section performs experiments on a cluster of six heterogeneous resources. The resource features are presented in Table 6.9. The bandwidth between the resources is 100Mbps. It should be noted that these experiments use rather extreme settings in terms of heterogeneity to clearly expose the differences among schedulers.

Two production Hadoop MapReduce traces, presented in (Chen et al. [2011]), are used in the experiments. One trace is from a cluster at Facebook, spanning six months from May to October 2009. The other trace is from a cluster at Yahoo!, covering three weeks in late February/early March 2009. Both traces contain a list of job submission and completion times, data sizes of the input, shuffle and output stages, and the running times of map and reduce functions. The arrival

Resources	Slot		Mem	
	slot#	*execRate*	*Capacity*	*RetrieveRate*
R_1	2	$5MHz$	$4TB$	$9Gbps$
R_2	16	$500MHz$	$400KB$	$40Kbps$
R_3	16	$500MHz$	$4TB$	$9Gbps$
R_4	2	$5MHz$	$4TB$	$9Gbps$
R_5	16	$500MHz$	$400KB$	$40Kbps$
R_6	2	$5MHz$	$400KB$	$40Kbps$

Table 6.9. Experimental resources

time of the jobs in our experiments are defined by considering the number of jobs in each Facebook and Yahoo! trace, and the total number of submitted jobs in each experiment. The proportion of the number of each job class to whole number of jobs in our workload is the same as the actual trace. However, the arrival time of the jobs in the traces are reduced with the same amount to include all types of the jobs in our workload. The analysis in (Chen et al. [2011]) provides classes of "common jobs" for each of the Facebook and Yahoo! traces using k-means clustering. The details of the Facebook and Yahoo! workloads are provided in Table 6.10 (Chen et al. [2011]).

Job Categories	Duration (sec)	Job	Input	Shuffle	Output	Map Time	Reduce Time
Facebook trace							
Small jobs	32	126	$21KB$	0	$871KB$	20	0
Fast data load	1260	25	$381KB$	0	$1.9GB$	6079	0
Slow data load	6600	3	$10KB$	0	$4.2GB$	26321	0
Large data load	4200	10	$405KB$	0	$447GB$	66657	0
Huge data load	18300	3	$446KB$	0	$1.1TB$	125662	0
Fast aggregate	900	10	$230GB$	$8.8GB$	$491MB$	104338	66760
Aggregate and expand	1800	6	$1.9TB$	$502MB$	$2.6GB$	348942	76736
Expand and aggregate	5100	2	$418GB$	$2.5TB$	$45GB$	1076089	974395
Data transform	2100	14	$255GB$	$788GB$	$1.6GB$	384562	338050
Data summary	3300	1	$7.6TB$	$51GB$	$104KB$	4843452	853911
Yahoo! trace							
Small jobs	60	114	$174MB$	$73MB$	$6MB$	412	740
Fast aggregate	2100	23	$568GB$	$76GB$	$3.9GB$	270376	589385
Expand and aggregate	2400	10	$206GB$	$1.5TB$	$133MB$	983998	1425941
Transform expand	9300	5	$806GB$	$235GB$	$10TB$	257567	979181
Data summary	13500	7	$4.9TB$	$78GB$	$775MB$	4481926	1663358
Large data summary	30900	4	$31TB$	$937GB$	$475MB$	33606055	31884004
Data transform	3600	36	$36GB$	$15GB$	$4.0GB$	15021	13614
Large data transform	16800	1	$5.5TB$	$10TB$	$2.5TB$	7729409	8305880

Table 6.10. Job categories in both traces. Map time and Reduce time are in Task-seconds, e.g., 2 tasks of 10 seconds each is 20 Task-seconds (Chen et al. [2011]).

It should be clarified that the job sizes are defined based on their mean execution times reported in (Chen et al. [2011]). Heterogeneous users with different minimum shares and priorities are defined. Table 6.11 defines the users in the Facebook and Yahoo! experiments. The minimum share of each user is defined based on its submitted job size, where each user submits jobs from one of the job classes in Table 6.10.

Users	$MinimumShare$	$Priority$
Facebook experiments		
U_1	5	1
U_2	0	2
U_3	0	2
U_4	5	1
U_5	10	2
U_6	15	1
U_7	4	2
U_8	10	1
U_9	10	1
U_{10}	15	1
Yahoo! experiments		
U_1	5	1
U_2	0	2
U_3	10	2
U_4	15	1
U_5	10	1
U_6	15	2
U_7	10	1
U_8	15	1

Table 6.11. User properties in experiments for both workloads

In each experiment 100 jobs are submitted to the system. The Hadoop block size is set to 128MB, which is the default size in Hadoop 0.21. The data replication number is set to three for all schedulers.

6.7.2 Results and Analysis

In each experiment the COSHH scheduler, the FIFO scheduler, and the version of the Fair Sharing scheduler presented in (Zaharia et al. [2009]) are compared. The comparison is based on the performance metrics introduced in Section 6.5.

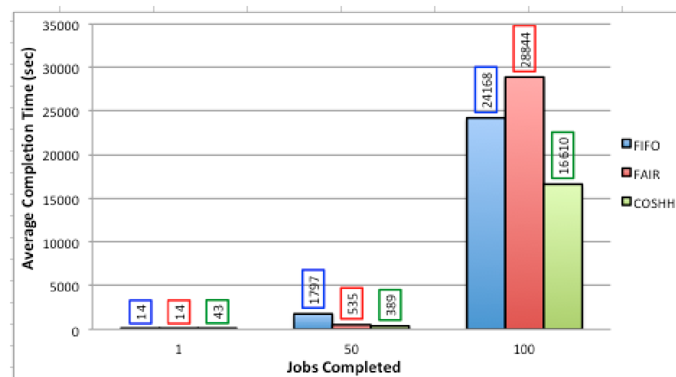


Figure 6.11. Average completion time for Facebook workload

Figures 6.11 and 6.12 present the average completion times of the schedulers running the Facebook and Yahoo! workloads, respectively. The results confirm that compared to the other schedulers, COSHH achieves the best average completion time in both workloads, which is caused by considering heterogeneity in its scheduling decisions. On the other hand, the Fair Sharing scheduler has the highest average completion time compared to the other two schedulers. Both the Facebook and Yahoo! workloads are heterogeneous, in which the arrival rates of small jobs are higher. In a heterogeneous Hadoop workload, jobs have different mean execution times (job sizes). For such workloads, as the FIFO algorithm does not take into account job sizes, it causes the problem that small jobs potentially get stuck behind large ones. Therefore, when the Hadoop workload is heterogeneous, the FIFO algorithm can significantly increase the completion time of small jobs. The Fair Sharing and the COSHH algorithms do not have this problem. Fair Sharing puts the jobs in different pools based on their sizes, and assigns a fair share to each pool. As a result, the Fair Sharing algorithm executes different size jobs in parallel. The COSHH algorithm assigns the jobs to the resources based on the size of the jobs and the execution rates of the resources. As the Fair Sharing scheduler first satisfies the minimum shares, it executes most of the small jobs after satisfying the minimum shares of the larger jobs. Therefore, the completion times of the small jobs (the majority of the jobs in this workload) are increased.

In the Yahoo! workload, the COSHH scheduler leads to 74.49% and 79.73% improvement in average completion time over the FIFO scheduler, and the Fair Sharing scheduler, respectively. Moreover, for the Facebook workload, the COSHH scheduler results in 31.27% and 42.41% improvement in the average completion time over the FIFO scheduler, and the Fair Sharing scheduler, respectively. It should be noted that the COSHH scheduler leads to a more substantial improvement for average completion time in the Yahoo! workload than in the Facebook workload. The reason is that the jobs in the Facebook workload are smaller and less heterogeneous than the

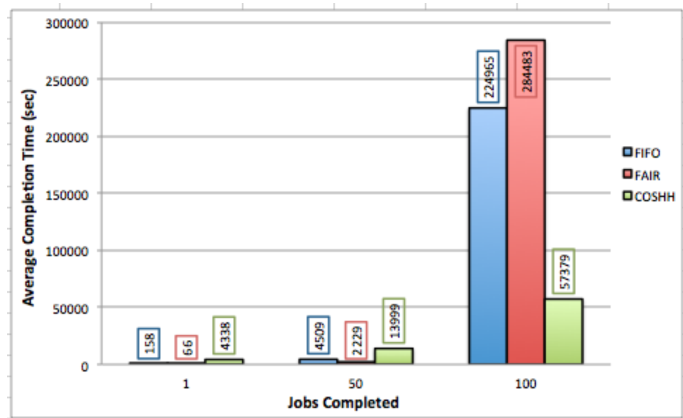


Figure 6.12. Average completion time for Yahoo! workload

jobs in the Yahoo! workload. As a result, taking the heterogeneity into account in the Yahoo! workload leads to more improvement. Moreover, as the experimental environment here is set to be extremely heterogeneous, it leads to a considerable level of improvement in the COSHH scheduler over the other two schedulers. This chapter defines the experimental environment such that the differences between the schedulers are emphasized. In the following chapters, the schedulers are further compared in other experimental environments.

The overheads of the scheduling algorithms are presented in Figures 6.13 and 6.14 for the Yahoo! and the Facebook workloads, respectively. The overheads show that the improvement for average completion time in the COSHH scheduler is achieved at the cost of increasing the overhead of scheduling. However, the additional 5 second overhead for the COSHH algorithm, compared to its improvement for average completion time (which is more than 10000 seconds) is negligible. The time spent for classification and solving the LP at the beginning of the COSHH scheduler leads to a higher scheduling time. The scheduling times for both the Fair Sharing and the COSHH schedulers increase considerably at around the point of scheduling the 50th to 60th jobs. These schedulers need to sort the users based on their shares to consider fairness and minimum share satisfaction. In the first stage of satisfying the minimum shares, they need to sort a smaller number of users. However, after satisfying the minimum shares, the number of users to be sorted is increased. Also, the order of users is changed based on the fairness. As a result, the process of sorting users takes longer, and causes an increase in the scheduling time for both algorithms. The FIFO algorithm has the least overhead.

Fairness, dissatisfaction, and the locality of the algorithms are presented in Tables 6.12 and 6.13 for the Yahoo! and the Facebook workloads, respectively. The results for both workloads show that COSHH has competitive dissatisfaction and fairness with the Fair Sharing algorithm. Because the COSHH scheduler has two stages to consider minimum share satisfaction and fairness, it is successful in reducing the dissatisfaction along with improving the fairness. First the scheduler only satisfies the minimum shares based on the priority of the users, and then it focuses on improving fairness. As COSHH considers the weights of the users, it does not allow starvation of low priority users who have small minimum shares.

Metrics	FIFO	Fair	COSHH
$Dissatisfaction$	8.618	$7.16E-04$	1.209
$Fairness$	4.974	0.965	2.779
$Locality(\%)$	95.6	97.4	96.5

Table 6.12. Dissatisfaction, fairness, and locality for Yahoo! workload

Based on the results, the locality of COSHH is competitive with the Fair Sharing scheduler. As the experimental environment is a small Hadoop cluster, the scheduler's replication method can

Metrics	FIFO	Fair	COSHH
$Dissatisfaction$	10.782	$8.31E-02$	0.294
$Fairness$	6.646	2.537	0.663
$Locality(\%)$	97.7	95.7	95.0

Table 6.13. Dissatisfaction, fairness, and locality for Facebook workload

not lead to considerable improvement in the locality. However, the advantages of using COSHH's replication method could be significant on large Hadoop clusters. This is left for future research.

6.8 Discussion

The priorities and the minimum shares of users are usually defined by the Hadoop providers. The minimum shares may be defined without taking the job sizes into account. The minimum shares can help in improving average completion times, in particular when the minimum shares are defined based on the job sizes. Therefore, to complete the COSHH analysis, the performance of the schedulers is also studied with no minimum shares assigned to the users. The experimental environment in this section is the same as the previous section, except that the users are homogeneous with zero minimum shares, and priorities equal to one.

Figures 6.15 and 6.16 present the average completion time metric for the schedulers running the Facebook and Yahoo! workloads, respectively. The results show that when the users are homogeneous, and no minimum share is defined, the average completion time of the FIFO algorithm is higher than the Fair Sharing algorithm. Unlike the FIFO algorithm, the Fair Sharing algorithm does not have the problem of small jobs stuck behind large ones. In addition, the minimum share satisfaction of large jobs will not defer the scheduling of smaller jobs. In the Yahoo! workload, the Fair Sharing algorithm achieves a 37.72% smaller average completion time than the FIFO

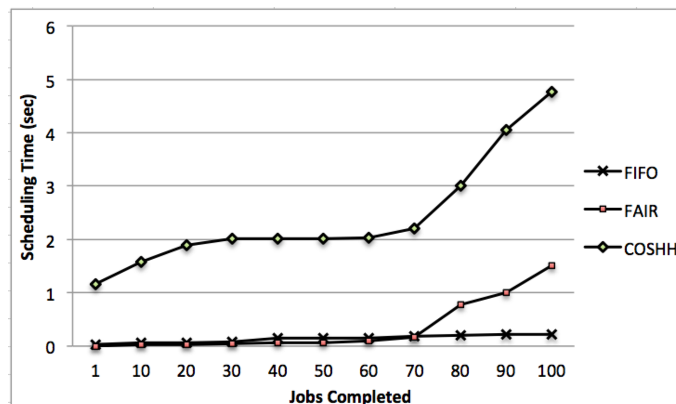


Figure 6.13. Scheduling overheads in Yahoo! workload

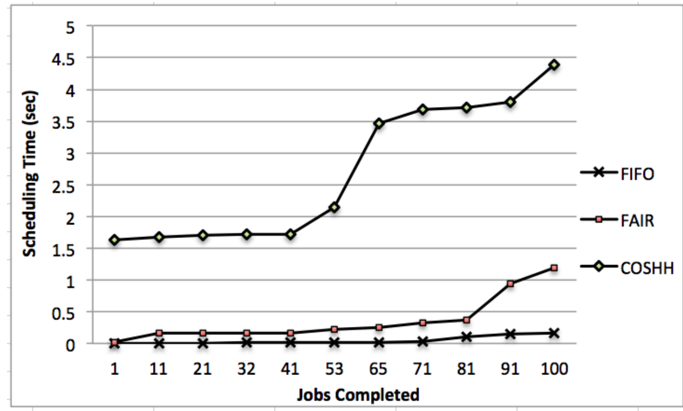


Figure 6.14. Scheduling overheads in Facebook workload

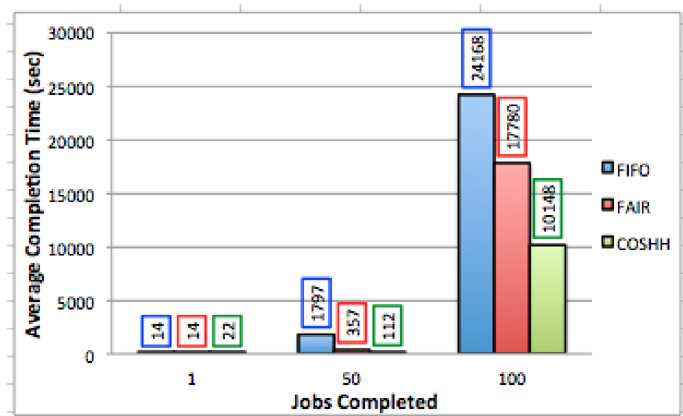


Figure 6.15. Average completion time for Facebook workload

algorithm, and the COSHH algorithm reduces the average completion time of the Fair Sharing algorithm by 75.92%. Moreover, in the Facebook workload, the Fair Sharing algorithm achieves a 26.42% smaller average completion time than the FIFO algorithm, and the COSHH algorithm reduces the average completion time of the Fair Sharing algorithm by 42.97%. Because of the job sizes, and level of heterogeneity, the average completion time improvement of COSHH for the Yahoo! workload is higher than for the Facebook workload.

The overheads of the scheduling algorithms are presented in Figures 6.17 and 6.18 for the Yahoo! and the Facebook workloads, respectively. Because most of the jobs in this workload are small, and have fewer tasks, the scheduling overheads are low. The classification and LP solution time in the COSHH algorithm lead to a large initial scheduling time. The overheads of both the Fair Sharing and the COSHH algorithms are lower when the users are homogeneous. In this case these algorithms no longer have the minimum share satisfaction stages.

Fairness and locality of the algorithms are presented in Tables 6.14 and 6.15 for the Yahoo!

and the Facebook workloads, respectively. Because there is no minimum share defined in these experiments, the amount of dissatisfaction for all schedulers is zero. The results show that the Fair Sharing algorithm has the best fairness.

Metrics	FIFO	Fair	COSHH
$Fairness$	1.032	0.504	0.856
$Locality(\%)$	94.9	97.9	98.1

Table 6.14. Fairness and locality for Yahoo! workload

Metrics	FIFO	Fair	COSHH
$Fairness$	1.188	0.429	0.926
$Locality(\%)$	93.4	95.2	98.3

Table 6.15. Fairness and locality for Facebook workload

The locality of COSHH is close to, and in most cases is better than the Fair Sharing algorithm. This can be explained by the fact that COSHH chooses the replication places based on the suggested classes for each resource.

6.9 Sensitivity Analysis

The COSHH scheduler uses the estimated job mean execution times, which makes it dependent on the accuracy of the estimation method. It is important for the COSHH scheduler to tolerate some level of inaccuracy in the estimated mean execution times. This section evaluates the COSHH scheduler from the sensitivity point of view.

The task scheduling process component in the COSHH scheduler provides an estimate of the mean execution time of an incoming job. To measure how much the performance of the proposed

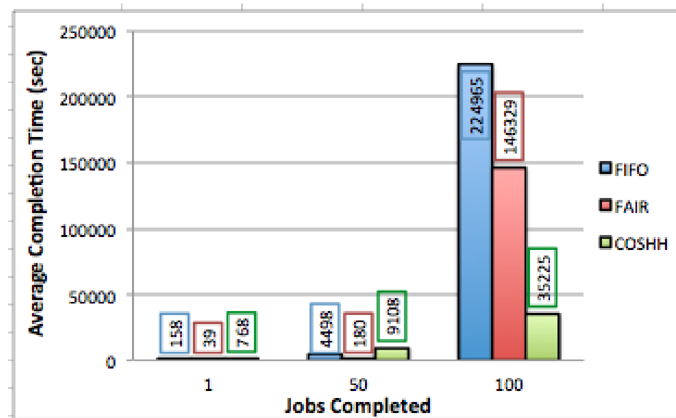


Figure 6.16. Average completion time for Yahoo! workload

scheduler is dependent on the estimation error, the scheduler is evaluated with various workloads under different levels of error. In order to completely study the robustness of the proposed scheduler, we examine cases that have 0% to 40% error in the estimates; however, typically these errors are on the order of 10% (Akioka and Muraoka [2004]). The error models discussed in (Iosup et al. [2008]) are used for estimating the mean execution times, which are the same error models used in Section 3.6 for estimation errors in Computational Grid systems. Generally the error model for these estimates in Hadoop is an *Over and Under Estimation Error* model, which is as follows. Define the actual execution time of job i on Resource j to be $L(i,j)$. Let $\hat{L}(i,j)$ denote the (corresponding) estimated mean execution time. In the simulations, $\hat{L}(i,j)$ is obtained from $\hat{L}(i,j) = L(i,j) \times (1 + E_r)$. Here, E_r is the error for estimating the job mean execution time, which is sampled from the uniform distribution $[-I, +I]$, where I is the maximum error.

First, the proposed scheduler is evaluated in an environment with accurate estimated mean execution times, and then the amount of error in estimating the mean execution times is increased. Figures 6.19 and 6.20 present the average completion times for the Yahoo! and the Facebook workloads, respectively. Different error levels are considered in these Figures, where COSHH-n denotes the COSHH scheduler with n% error in estimating the mean execution times. For each experiment, 30 replications are run to construct 95%-confidence intervals. The lower and upper bounds of the confidence intervals are represented with lines on each bar.

Based on the results, up to 40 percent error in estimation does not significantly affect the average completion time of the proposed scheduler. The reason is that the COSHH scheduler uses the estimated mean execution times to provide suggestions for each resource; the estimates themselves are not directly used in the final scheduling decisions. Small amounts of error lead to only a slight change in job classes. As a result, the average completion time is not increased for 10% error levels. When the error level is higher, it leads to more changes in the job classes, which

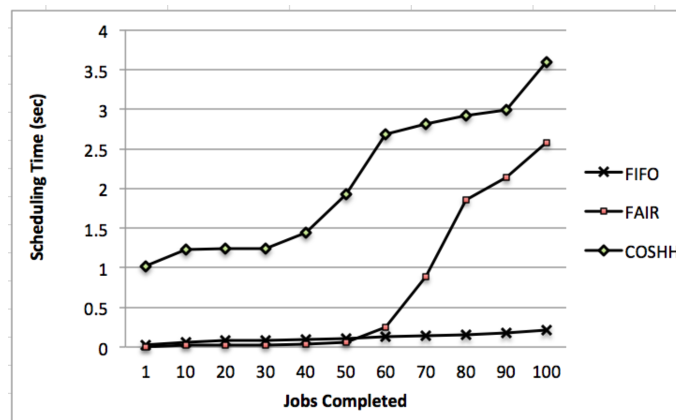


Figure 6.17. Scheduling overheads in Yahoo! workload

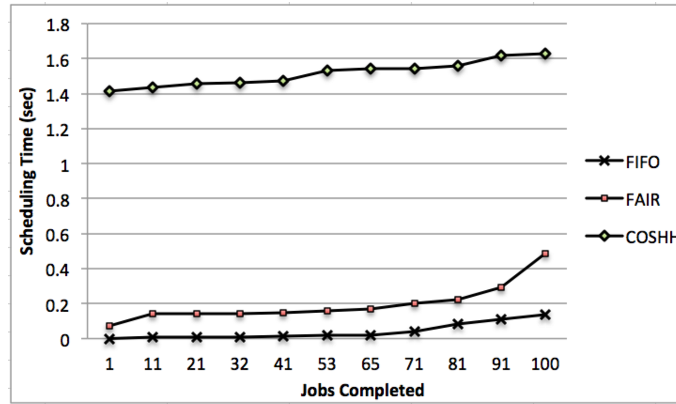


Figure 6.18. Scheduling overheads in Facebook workload

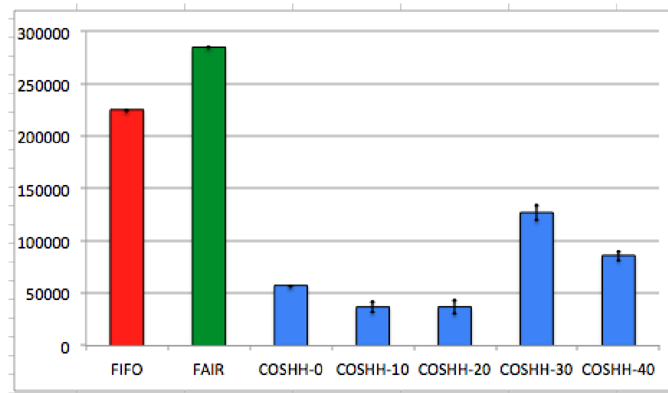


Figure 6.19. Average completion time for Yahoo! workload - Sensitivity to error in estimation

modifies the suggested set of classes considerably, and affects the average completion time. The results show that the COSHH algorithm is not very sensitive to the estimated mean execution times, and it maintains better average completion time than the other algorithms in up to 40% error in estimating the mean execution times.

Figures 6.21 and 6.22 present the scheduling times for the Yahoo! and the Facebook workloads, respectively. The error level causes a slight increase in scheduling time, due to longer classification processes. When there are estimation errors, the k-means clustering method may need to run more steps to reach the appropriate classification.

6.10 Related Work

The scheduler is the centrepiece of MapReduce and Hadoop systems. Desired performance levels can be achieved by an appropriate submission of jobs to resources based on the system parameters and the performance metrics. The following presents current Hadoop schedulers categorized based

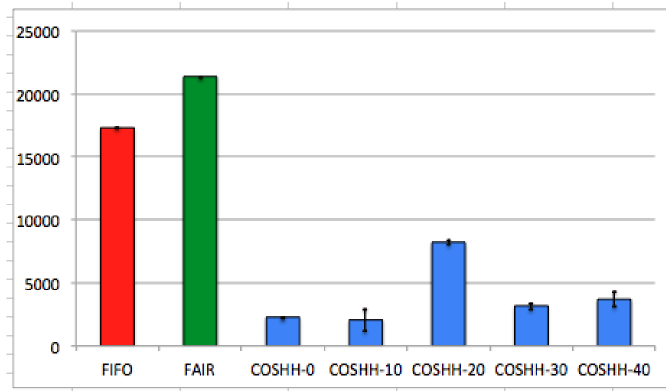


Figure 6.20. Average completion time for Facebook workload - Sensitivity to error in estimation

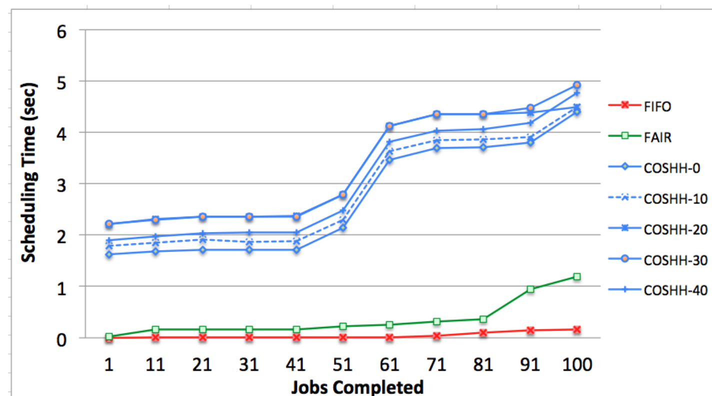


Figure 6.21. Scheduling time for Yahoo! workload - Sensitivity to estimation error

on their main features.

6.10.1 Simple schedulers

MapReduce was initially designed in (Dean and Ghemawat [2008]) for performing large batch jobs in small teams. A simple scheduling algorithm like FIFO, which makes fast scheduling decisions with low overhead could achieve an acceptable performance level in initial Hadoop systems. However, experience from deploying Hadoop in large systems shows that basic scheduling algorithms such as FIFO can cause severe performance degradation; particularly in systems that share data among multiple users (Zaharia et al. [2009]).

The next generation of Hadoop schedulers, called Hadoop on Demand (HOD) (Apache [2007]), addresses cluster sharing issues by setting up private Hadoop clusters based on user demands. HOD allows users to share a common file system while owning private Hadoop clusters on their allocated nodes. The FIFO scheduler is used for allocating the jobs in each of these private Hadoop clusters. This approach failed in practice because it violates the data locality design of the original

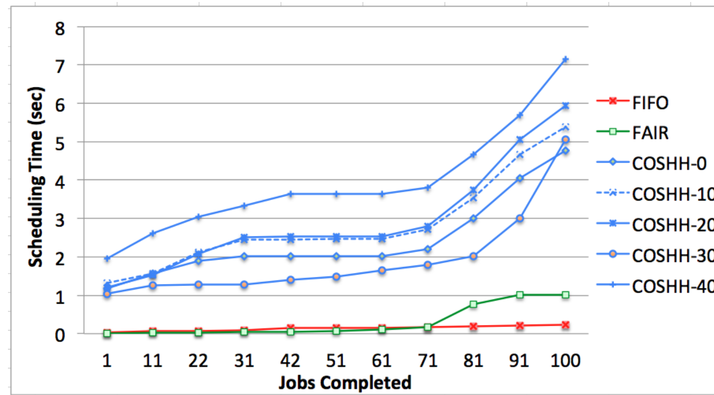


Figure 6.22. Scheduling time for Facebook workload - Sensitivity to estimation error

MapReduce schedulers. As the processing nodes only cover a subset of the data nodes, more data transfers were required between data nodes and the compute nodes. Creating small sub-clusters for processing individual user jobs in HOD leads to high maintenance costs, poor scalability, and degraded performance. The poor performance of simple Hadoop schedulers led to the design of more complicated schedulers as a practical solution for Hadoop systems with multiple users.

6.10.2 User Fairness based Schedulers

To address some of the shortcomings of the FIFO algorithm, the Fair Sharing scheduling system is introduced in (Zaharia et al. [2009]). The Fair Sharing scheduler uses system parameters such as user priorities and minimum shares to make scheduling decisions. The Delay scheduler (Zaharia et al. [2010]) is a complementary algorithm for Fair Sharing to improve data locality. To achieve this goal the Delay scheduler uses additional state information such as the input data locations for each job. However, both the Fair Sharing and Delay schedulers neglect system heterogeneity in their scheduling decisions.

The Capacity scheduler proposed in (Apache Hadoop Capacity Scheduler [2010]) was originally developed at Yahoo! to address the case where the number of users is large, and the users should receive fair allocations of resources. The scheduler defines multiple queues with configurable numbers of map and reduce slots. The jobs in a queue are given the configured capacity of the queue, and the extra capacity is divided among the jobs in other queues. Each queue uses the FIFO algorithm with priorities. The percentage of running tasks per user can be limited to share a cluster equally among the users. Upon receiving a heartbeat message from a resource, the least loaded queue is chosen, from which the oldest remaining job is selected. This scheduler enforces sharing of cluster capacity among users, rather than among jobs. The Capacity scheduler cannot always match the resource allocations with the task demands, especially when these demands

are widely heterogeneous. This mismatch may lead to either low cluster utilization or poor performance due to resource oversubscription.

A flexible scheduling allocation scheme (Wolf et al. [2010]), known as Flex, extends the Fair Sharing scheduler, while avoiding its job starvation problems. The goal of the Flex algorithm is to optimize any variety of specified metrics while ensuring the same minimum and maximum slot guarantees as in the Fair Sharing algorithm. The metrics in Flex can be chosen by the system administrator on a cluster-wide basis, or by individual users on a job-by-job basis. These metrics can be selected from a menu that includes response time, makespan (dual to throughput), and any of several metrics which (reward or) penalize job completion times compared to possible deadlines. Generally, this approach could be considered as negotiating Service Level Agreements (SLAs) with MapReduce users, where the penalty of missing an SLA depends on the achieved service level. Flex defines a speedup function for each job based on its allocated slots. While the speedup function is used to model each job, it is not clear how to derive this function for different jobs and different sizes of input datasets. Flex does not provide a detailed MapReduce performance model, but instead it uses a set of simplifying assumptions about job execution and progress over time. Moreover, no case study is provided to evaluate the proposed job model and the approach in achieving the targeted job deadlines.

The Quincy scheduler (Isard et al. [2009]) is proposed for the Dryad environment (Isard et al. [2007]). The Dryad distributed execution engine (Isard et al. [2007]) has a similar low level computational model as the MapReduce and Hadoop systems. The challenge of scheduling introduced in (Isard et al. [2009]) shows that there is a conflict between satisfying the fairness and locality metrics. Intuitively this is because a strategy that achieves optimal data locality will typically delay a job until its ideal resources are available. On the other hand, for improving fairness, the best available resources should be allocated to a job as soon as possible, even if the communication cost between the computation and data resources is not minimized. This scheduler achieves fairness by modelling the fair scheduling problem as a min-cost flow problem. It maps the scheduling problem to a graph data structure, where edge weights and capacities encode the competing demands of data locality, fairness, and starvation avoidance. Then, a standard solver is used to compute the online schedule according to a global cost model. Quincy does not currently support multi-resource fairness, and it has problems in incorporating multi-resource requirements into the min-cost flow formulation. Similar to the Fair Sharing scheduler, Quincy does not provide any special support for improving job completion times.

6.10.3 Formal Model based Schedulers

The MapReduce scheduling problem is formalized in (Moseley et al. [2011]) as a generalized version of the classical two-stage flexible flow-shop problem with identical machines. It defines a two stage scheduling problem, where there are constraints defined between each map task and reduce task of a job. This scheduling method considers two scenarios: offline and online job arrivals. In the offline scenario, jobs arrive together, and the focus is on optimizing the approximation ratio. In the online scenario, jobs arrive over time, where the scheduler makes decisions without knowing the jobs that are yet to arrive; the focus is on optimizing the competitive ratio. The theoretical study is done in two processing time configurations. First, in the identical machines setting, where all map machines have the same speed, and similarly, all the reduce machines have the same speed. Second, in the unrelated machines setting, in which the processing time for each task is a vector, specifying the task running times on each of the machines. The paper generalizes the flexible flow-shop problem by having a set of map tasks per job that need to be scheduled on the map machines and a set of reduce tasks that are to be scheduled on the reduce machines. However, it does not take into account other common Hadoop performance metrics such as fairness or average completion time. Moreover, as it aims to provide a simple formalization of the MapReduce scheduling problem, it abstracts some of the main requirements in the system such as the relations between reduce and map tasks within a job. The abstractions lead to elimination of the parameters in real MapReduce systems, which often have a significant affect on the performance.

In (Aboulnaga et al. [2009]), the authors introduce a scheduling algorithm for MapReduce systems to minimize the total completion time, while improving the CPU and I/O utilizations of the cluster. The algorithm defines Virtual Machines (VM), and decides how to allocate the VMs to each Hadoop job, and to the physical Hadoop resources. For this purpose, a constrained optimization problem is formulated and solved. To define the optimization problem, a mathematical performance model is required for different jobs in the system. The algorithm first runs all job types in the Hadoop system to build corresponding performance models. Then, assuming these jobs will be submitted repeatedly, scheduling decisions for each job are made based on the defined optimization problem's solution. The algorithm assumes that job characteristics will not vary between runs, and also that when a job will be executed on a resource, all its required data is located on that resource. The problem with this algorithm is that it can not make any decisions when a new job is submitted. Moreover, the assumption that all of the job's required data is available on the running resource, without considering the overhead of transmitting the data is not realistic. Furthermore, virtualization can add significant overhead to the scheduling system.

The work of (Phan et al. [2010]) explores the feasibility of enabling real-time scheduling of

MapReduce jobs. They present a formal model for capturing real-time MapReduce applications and the Hadoop system. Using this model, they formulate the offline scheduling of real-time MapReduce jobs on a heterogeneous distributed Hadoop architecture as a constraint satisfaction problem, and introduce search strategies for the formulation. Finally, they propose an enhancement of the MapReduce execution model and a range of heuristic techniques for online scheduling. However, all the proposed heuristics can only perform in homogeneous systems. Tian et al. [2009] suggest the TripleQueue scheduler that seeks better system utilization via scheduling jobs based on (predicted) resource usage. Ganapathi et al. [2010] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job.

6.10.4 Job Deadline based Schedulers

The ARIA framework, proposed in (Verma et al. [2011]), aims to allocate the appropriate amount of resources to each job to meet a required Service Level Objective (SLO). First, for a production job that is routinely executed on a new dataset, it builds a job profile that compactly summarizes critical performance characteristics of the underlying application during the map and reduce stages. Second, a MapReduce performance model is constructed for a given job (with a known profile) and its SLO (soft deadline), which estimates the amount of resources required for job completion within the deadline. Finally, a deadline-based scheduler for Hadoop is introduced that determines job ordering and the amount of resources to allocate to each job. This scheduler extracts job profiles from past executions, and provides a variety of bounds-based models for predicting a job's completion time as a function of allocated resources. However, these models apply only to a single MapReduce job.

A deadline constraint scheduler (Kc and Anyanwu [2010]) is defined for jobs with deadlines, and focuses on increasing system utilization. To address the deadlines, a job execution cost model is introduced based on parameters such as map and reduce runtimes, input data sizes, and data distribution. The scheduling algorithm receives user deadlines as its input, and determines whether the incoming job can be finished within the specified deadline or not. Jobs are only scheduled if specified deadlines can be satisfied. If a job can not be finished before its assigned deadline, the scheduler will inform the user to adjust the job deadline. There are some strict assumptions in this scheduler such as all nodes are homogeneous, unit cost of processing for each map or reduce node is equal, and input data is distributed uniformly. The assumptions and requirement of defining deadlines by users makes this scheduling algorithm impractical for most Hadoop systems. Moreover, the assigned deadlines may lead to increasing the average completion time of the jobs.

An online job completion time estimator is introduced by Polo et al. [2010], and is used for adjusting resource allocations for different jobs. The Adaptive MapReduce Scheduler is then introduced to meet user defined performance goals such as deadlines. The provided results show that the adaptive scheduler provides dynamic resource allocation across jobs. However, their estimator tracks the progress of the map tasks alone and has no information or control over the reduce tasks. This thesis does not consider Hadoop systems with specified deadlines for individual jobs. It rather concentrates on overall improvement of job completion times.

6.11 Conclusion

In order to keep Hadoop schedulers simple, minimal system information is typically used in making scheduling decisions, which in some cases results in poor performance. Growing interest in applying the MapReduce programming model in various applications gives rise to greater heterogeneity, and thus must be considered in its impact on performance. However, heterogeneity is for the most part neglected in designing Hadoop scheduling systems. It has been shown that it is possible to estimate Hadoop system parameters. Using the system information, this thesis designed a Hadoop scheduling system which classifies the jobs based on their requirements and finds an appropriate matching of resources and jobs.

The results show that even with around 40% error in estimating job mean execution times, the COSHH scheduler provides a significant improvement in the average completion times. The proposed scheduler (COSHH) is adaptable to variations in the system parameters. The classification part detects changes and modifies the classes based on the new system parameters. Also, the job mean execution times are estimated when a new job is submitted to the system, which makes the scheduler adaptable to changes in job mean execution times. The scheduling times of different schedulers are provided to evaluate their overheads. Based on the results, the COSHH scheduler improves the average completion time at the cost of increasing scheduling overhead. However, compared to the improvement in average completion time, the additional overhead of the COSHH scheduler is in most cases negligible.

This chapter evaluated the COSHH scheduler in a heterogeneous Hadoop system with different synthetic and real Hadoop workloads. However, the considered heterogeneous Hadoop systems were only one possible case of heterogeneity in Hadoop. In the next chapter various possible cases of Hadoop heterogeneity are considered, and performance of different schedulers are evaluated in these cases. The goal is to provide a more detailed analysis of the root causes of performance degradation that may occur due to heterogeneity and a more thorough study of how different schedulers perform in the face of heterogeneity in different aspects of a Hadoop system. As such, this chapter can be seen as a "proof of concept" of the COSSH scheduler.

Chapter 7

Guidelines for Selecting Hadoop Schedulers based on System Heterogeneity [1]

A Hadoop system can be specified using three main factors: cluster, workload, and user, where each can be either heterogeneous or homogeneous. A Hadoop cluster is a group of linked resources. Organizations could use existing resources to build Hadoop clusters - small companies may use their available (heterogeneous) resources to build a Hadoop cluster, or a large company may specify a number of (homogeneous) resources for setting up its Hadoop cluster. There can be a variety of users in a Hadoop system who are differentiated based on features such as priority, usage, and guaranteed shares. Similarly, workload in the Hadoop system may have differing numbers of user jobs and corresponding requirements. Heterogeneity in Hadoop is defined based on the level of heterogeneity in the Hadoop factors.

To increase the utilization of a Hadoop cluster, different types of applications may be assigned to one cluster, which leads to increasing the level of heterogeneity in the workload. However, there are situations where a company assigns a Hadoop cluster to specific jobs as the jobs are critical, confidential, or highly data or computation intensive. Accordingly, the types of applications assigned by different users to a Hadoop cluster define the heterogeneity levels of the workload and users in the corresponding Hadoop system. Similarly, the types of resources define the heterogeneity of Hadoop clusters.

The heterogeneity level of each Hadoop factor potentially has a significant effect on performance. It is critical to select a scheduling algorithm by considering the Hadoop factors, and the desired

[1] This chapter is mostly based on the following paper: A. Rasooli and D. G. Down, *Guidelines for Selecting Hadoop Schedulers based on System Heterogeneity.* **Journal of Grid Computing**, 2012, (Submitted).

performance level. This chapter provides guidelines for selecting scheduling algorithms based on Hadoop factors, and their heterogeneity levels. The performed analysis and proposed guidelines are based on three Hadoop schedulers: FIFO, Fair Sharing, and COSHH. These algorithms are selected as representatives of schedulers which consider heterogeneity at different levels. The FIFO scheduler does not consider heterogeneity in its scheduling decisions. However, the Fair Sharing and COSHH algorithms consider partial and complete heterogeneity, respectively.

In this chapter, performance issues for Hadoop schedulers are introduced (Section 7.1). To reduce the dimensionality of the space of heterogeneity factors, this research performs a categorization of Hadoop systems (Section 7.2). The scheduler performance in each category is also experimentally analyzed and discussed. Section 7.3 presents the experimental framework. Sections 7.4 and 7.5 provide the experimental results and analysis for homogeneous and heterogeneous Hadoop environments, respectively. Finally, using the experiments and discussions, Hadoop scheduler selection guidelines are proposed in Section 7.6. The guidelines are evaluated using different Hadoop systems. Related work is discussed in Section 7.7, and Section 7.8 provides a conclusion.

7.1 Performance Issues

This section analyzes the main drawbacks of each scheduler for various heterogeneity levels in the Hadoop factors. For this purpose, two example systems are defined, one heterogeneous and one homogeneous. Example A is used to define problems I, II, and III, and Example B illustrates problem IV. The choice of system sizes in these examples are only for ease of presentation, the same issues arise in larger systems.

7.1.1 Problem I: Small Jobs Starvation

Example A- Heterogeneous Hadoop System: includes four heterogeneous resources and three users with the following characteristics:

- Task1, Task2, and Task3 represent three heterogeneous task types with the following mean execution times. Here, $m_t(T_i, R_j)$ is the mean execution time of task T_i on resource R_j.

$$m_t = \begin{bmatrix} 2.5 & 2.5 & 10 & 10 \\ 2.5 & 2.5 & 5 & 5 \\ 10 & 10 & 2.5 & 2.5 \end{bmatrix}$$

- Three users submit three jobs to the system, where each job consists of a number of similar tasks. Jobs arrive to the system in the following order: Job1, Job2, and Job3.

- Users are homogeneous with zero minimum share and priority equal to one. Each user submits one job to the system as follows:

  User1: Job1 (consists of 10 Task1)
  User2: Job3 (consists of 10 Task3)
  User3: Job2 (consists of 5 Task2)

Figure 7.1 shows the job assignments for the FIFO, Fair Sharing, and COSHH schedulers. The completion time of the last task in each job is highlighted to show the overall job completion times. The FIFO algorithm assigns incoming jobs to the resources based on their arrival times (Figure 7.1a). Consequently in the FIFO scheduler, execution of the smaller job (Job2) will be delayed significantly. In a heterogeneous Hadoop workload, jobs have different execution times. For such workloads, as the FIFO algorithm does not take into account job sizes, it has the problem that small jobs potentially get stuck behind large ones.

The Fair Sharing and the COSHH algorithms do not have this problem. Fair Sharing puts the jobs in different pools based on their sizes, and assigns a fair share to each pool. As a result, the Fair Sharing algorithm executes different size jobs in parallel. The COSHH algorithm assigns jobs to resources based on the job sizes and the execution rates of resources. As a result, it can avoid this problem.

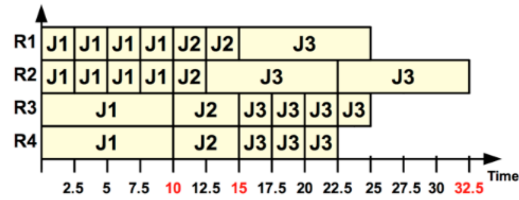
7.1.2 Problem II: Sticky Slots

Figure 7.1b shows the job-resource assignment for the Fair Sharing algorithm in Example A. As the users are homogeneous, the Fair Sharing scheduler searches through all of the users' pools, and assigns a slot to one user at each heartbeat. Upon completion of a task, the free slot is assigned to a new task of the same user to preserve fairness among users.

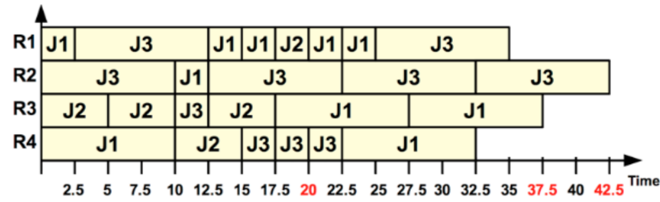
Resource2 is an inefficient choice for Job3 with respect to completion time, but the Fair Sharing scheduler assigns this job to this resource multiple times. There is a similar problem for Job1 assigned to Resource3 and Resource4. Consequently, the average completion time will be increased.

This problem arises when the scheduler assigns a job to the same resource at each heartbeat. This issue is first mentioned in (Zaharia et al. [2010]) for the Fair Sharing algorithm, where the authors considered the effect of this problem on locality. However, our example shows Sticky Slots can also significantly increase average completion times, when an inefficient resource is selected for a job.

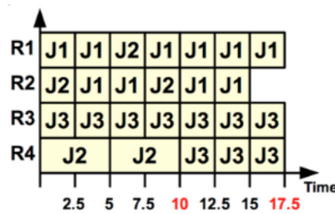
The FIFO algorithm does not have this problem because it only considers arrival times in making scheduling decisions. The COSHH algorithm has two levels of classification, which helps avoid



(a) FIFO Scheduler



(b) Fair Sharing Scheduler



(c) COSHH Scheduler

Scheduler	Job	Completion Time	Average Completion Time
FIFO	J1 J2 J3	10 15 32.5	19.17
Fair Sharing	J1 J2 J3	37.5 20 42.5	33.33
COSHH	J1 J2 J3	17.5 10 17.5	15

(d) Average Completion Time of Schedulers

Figure 7.1. Job assignment by a) FIFO, b) Fair Sharing, and c) COSHH schedulers, and their average completion times in Example A.

the Sticky Slot problem. Even when the same resource is assigned for a job in different rounds, the optimizations in the COSHH algorithm guarantee an appropriate selection of resource for the corresponding job.

7.1.3 Problem III: Resource and Job Mismatch

In a heterogeneous Hadoop system, resources can have different features with respect to their computation or storage units. Moreover, jobs in a heterogeneous workload have different requirements. To reduce the average completion time, it is critical to assign the jobs to resources by considering resource features and job requirements.

As the FIFO and the Fair Sharing algorithms do not consider heterogeneity when making scheduling decisions, they both have the problem of Resource and Job Mismatch. On the other hand, the COSHH algorithm has the advantage of appropriate matching of jobs and resources, applying the following process (Figure 7.1c). The COSHH algorithm classifies the jobs into three classes: Class1, Class2, and Class3, which contain Job1, Job2, and Job3, respectively. This scheduler solves an LP to find the best set of suggested job classes for each resource, as follows:

Resource1: {Class1, Class2}
Resource2: {Class1, Class2}
Resource3: {Class2, Class3}
Resource4: {Class2, Class3}

After computing the suggested sets, the COSHH scheduler considers fairness and minimum share satisfaction to assign a job to each resource. Although the COSHH algorithm assigns Job1 exclusively to Resource1 (Sticky Slot Problem), it does not increase the completion time of Job1. This is one of the main advantages of the COSHH algorithm over FIFO and Fair Sharing in a heterogeneous system.

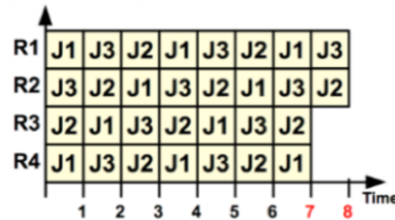
7.1.4 Problem IV: Scheduling Complexity

Example B - Homogeneous Hadoop System: includes four homogeneous resources, and three homogeneous users as follows:

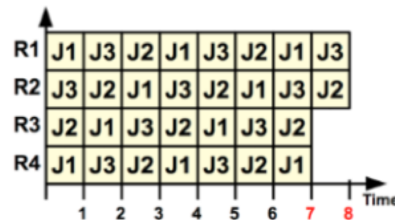
- There is one task type, and one job class, where each job consists of 10 tasks. Tasks are homogeneous, and they have execution time of 1 second on all resources.
- There are three homogeneous users similar to Example A.
- Users submit three jobs to the system, where each job consists of a number of similar tasks. Jobs arrive to the system in this order: Job1, Job2, and Job3.

The scheduling and completion times for the schedulers are presented in Figure 7.2. Figures 7.2a, 7.2b, and 7.2c show the job assignments in the FIFO, Fair Sharing, and COSHH schedulers, respectively.

(a) FIFO Scheduler



(b) Fair Sharing Scheduler



(c) COSHH Scheduler

Scheduler	Job	Completion Time	Average Completion Time
FIFO	J1 J2 J3	3 5 8	5.33
Fair Sharing	J1 J2 J3	7 8 8	7.67
COSHH	J1 J2 J3	7 8 8	7.67

(d) Average Completion Time of Schedulers

Figure 7.2. Job assignment by a) FIFO, b) Fair Sharing, and c) COSHH schedulers, and their average completion times in Example B.

In a fully homogeneous Hadoop system, a simple algorithm like FIFO, which quickly multiplexes jobs to resources, leads to the best average completion time compared to the other, more complex algorithms (Figure 7.2a). As the users are homogeneous, at each heartbeat the Fair Sharing algorithm assigns a task of a user to the current free resource (Figure 7.2b). In the case of the Fair Sharing scheduler, at each heartbeat there is the overhead of comparing previous usage of resources

to make a fair assignment. Finally, for the COSHH scheduler, as the system is homogeneous, and there is only one job class (i.e., jobs have similar features), solving the LP suggests all job classes for all resources. In this system, the scheduling decisions of the COSHH algorithm are identical to those of the Fair Sharing algorithm (Figure 7.2c). However, its scheduling complexity is greater than the Fair Sharing algorithm. The complexity of scheduling algorithms can result from different features, such as gathering more system parameters and state information, and considering various factors in making scheduling decisions. In a homogeneous system such as Example B, collecting system parameters and state information and a complex scheduling process add more complexity to the system which can not be compensated for by the more precise job and resource assignment. As a result, a simple and fast algorithm like FIFO can achieve better performance.

7.2 Heterogeneity in Hadoop

Reported analysis on several Hadoop systems found their workloads extremely heterogeneous with very different execution times (Chen et al. [2012]). However, due to privacy issues, most companies are unable or unwilling to release information about their Hadoop systems. Therefore, reported information about heterogeneity in Hadoop may be only partial. This PhD research considers various possible settings of heterogeneity for the Hadoop factors to provide a complete performance analysis of Hadoop schedulers in terms of system heterogeneity.

In a Hadoop system, the incoming jobs can be heterogeneous with respect to various features such as number of tasks, data and computation requirements, arrival rates, and execution times. Also, Hadoop resources may differ in capabilities such as data storage and processing units. The assigned priorities and minimum share requirements may differ between users. Moreover, the type and number of jobs assigned by each user can be different. As the mean execution time for a job on a resource, $mean_execTime(J_i, R_j)$, reflects the heterogeneity of both workload and cluster factors, this chapter considers four possible cases of heterogeneity of users and mean execution times, as follows:

- Homogeneous System: both the workload and cluster are homogeneous, and the users can be either homogeneous or heterogeneous. In these systems, the job sizes can significantly affect the performance of the Hadoop schedulers. Therefore, two case studies are defined for this category:

  - *Homogeneous-Small* (all jobs are small).
  - *Homogeneous-Large* (all jobs are large).

- Heterogeneous System: both the workload and cluster are heterogeneous, and users are either homogeneous or heterogeneous. In this system, the challenging issue for the schedulers is the arrival rates of different sized jobs. There are three case studies in this category:
  - *Heterogeneous-Small* (higher arrival rate for small jobs).
  - *Heterogeneous-Equal* (equal arrival rates for all job sizes).
  - *Heterogeneous-Large* (higher arrival rate for large jobs).

Overall, five case studies are defined which are evaluated in Sections 7.4 and 7.5. The real Hadoop workload experiments provided in Section 6.7 were samples of the Heterogeneous-Small case study.

7.3 Evaluation: Settings

This section defines the experimental environment and performance metrics. As the previous chapter performed evaluations that correspond to the Heterogeneous-Small case study, for ease of comparison, the same experimental environment is used for Heterogeneous-Small evaluations in this Chapter.

7.3.1 Experimental Environment

The general settings of the Hadoop factors in our experiments are defined as follows:

1. Workload: jobs are selected from Yahoo! production Hadoop MapReduce traces, presented in Chapter 6. The details of the workloads used for evaluating the schedulers are provided in Table 7.1 (which is similar to the workload in Table 6.10).

Job Categories	Duration (sec)	Job	Input	Shuffle	Output	Map Time	Reduce Time
Small jobs	60	114	$174MB$	$73MB$	$6MB$	412	740
Fast aggregate	2100	23	$568GB$	$76GB$	$3.9GB$	270376	589385
Expand and aggregate	2400	10	$206GB$	$1.5TB$	$133MB$	983998	1425941
Transform expand	9300	5	$806GB$	$235GB$	$10TB$	257567	979181
Data summary	13500	7	$4.9TB$	$78GB$	$775MB$	4481926	1663358
Large data summary	30900	4	$31TB$	$937GB$	$475MB$	33606055	31884004
Data transform	3600	36	$36GB$	$15GB$	$4.0GB$	15021	13614
Large data transform	16800	1	$5.5TB$	$10TB$	$2.5TB$	7729409	8305880

Table 7.1. Job categories in the Yahoo! trace. Map time and Reduce time are in Task-seconds, e.g., 2 tasks of 10 seconds each is 20 Task-seconds (Chen et al. [2011]).

The job sizes are defined based on the execution times reported in (Chen et al. [2011]). Wherever it is not defined explicitly, "small jobs" and "large jobs" mean the Small jobs and Large data

summary classes in the Yahoo! workload, respectively. Similar to the experiments in the previous chapter, the default number of jobs is 100, which contains a variety of the behaviours in our Hadoop workload.

2. Clusters: have different configurations for the heterogeneous and homogeneous case studies. Here, the heterogeneous cluster is defined the same as in Section 6.7, where for ease of presentation the information is repeated in Table 7.2. The bandwidth between the resources is 100Mbps. Experiments with a homogeneous cluster use a cluster consisting of six R_3 resources.

Resources	Slot		Mem	
	$slot\#$	$execRate$	$Capacity$	$RetrieveRate$
R_1	2	$5MHz$	$4TB$	$9Gbps$
R_2	16	$500MHz$	$400KB$	$40Kbps$
R_3	16	$500MHz$	$4TB$	$9Gbps$
R_4	2	$5MHz$	$4TB$	$9Gbps$
R_5	16	$500MHz$	$400KB$	$40Kbps$
R_6	2	$5MHz$	$400KB$	$40Kbps$

Table 7.2. Resources in the heterogeneous cluster

3. Users: have two different settings in these experiments. In both settings, each user submits jobs from one of the job classes in Table 7.1. Heterogeneous users are defined with different minimum shares and priorities (Table 7.3). The minimum share of each user is defined to be proportional to its submitted job size. Therefore, the minimum share of U_2 (who is submitting the smallest jobs) is defined to be zero, and the minimum share of U_8 (who is submitting the largest jobs) is set to the maximum amount. In the case of homogeneous users, there are eight users, each with zero minimum share, and priority equal to one.

User	$MinimumShare$	$Priority$
U_1	5	1
U_2	0	2
U_3	10	2
U_4	15	1
U_5	10	1
U_6	15	2
U_7	10	1
U_8	15	1

Table 7.3. Heterogeneous Users

MRSIM (Hammoud et al. [2010]) is again used to simulate the various systems. The Hadoop block size is set to 128MB, and the data replication number is set to the default value of three in all algorithms.

7.4 Evaluation: Homogeneous Hadoop System

This section includes the performance analysis of three schedulers on a homogeneous Hadoop system. An example of these systems is the use of storage and computing power from Amazon Web Services to convert 11 million public domain articles in the New York Times archives from scanned images into PDF format (Gottfrid [2009]).

7.4.1 Case Study 1: Homogeneous-Small

This case study analyzes the performance of Hadoop schedulers for a homogeneous cluster and workload, where all the jobs are of small size. The workload consists of 100 "Small jobs" as defined in Table 7.1. Two experiments are performed. In the first experiment the users are heterogeneous, while in the second they are homogeneous (as defined in Section 7.3.1).

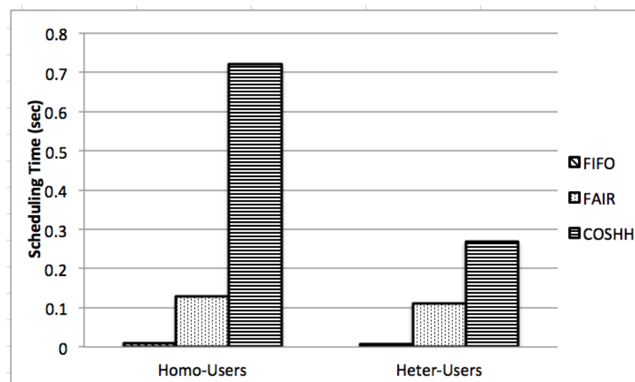


Figure 7.3. Scheduling time - Homogeneous-Small

In this homogeneous environment, the average completion times of all schedulers are almost equal. As the cluster and workload are homogeneous, the COSHH algorithm suggests all resources as the best choices for all job classes. Therefore, its performance is similar to the Fair Sharing algorithm. Moreover, due to the homogeneity in users, the Fair Sharing algorithm defines similar job pools for all users, where each job pool uses the FIFO algorithm to select a job. Therefore, despite the heterogeneity of users, the average completion time of all the algorithms are almost the same (around 98.8 seconds).

The scheduling overheads in the homogeneous-small Hadoop system are presented in Figure 7.3. The Scheduling Complexity problem in the COSHH algorithm leads to higher scheduling time and overhead. This is caused by the classification and LP solving processes. As the job sizes are small, the scheduling overhead of the COSHH algorithm does not lead to a significant increase in its average completion time. The total scheduling overhead here is less than a second, which is negligible compared to the processing times.

Tables 7.4 and 7.5 present the fairness, dissatisfaction, and locality when the users are heterogeneous and homogeneous, respectively. As the main goal of the Fair Sharing algorithm is to improve the fairness and minimum share satisfaction, it leads to better fairness and dissatisfaction.

Metrics	FIFO	Fair	COSHH
Dissatisfaction	8.615	0.801	0.824
Fairness	4.004	2.043	2.198
Locality (%)	97	97	97

Table 7.4. Dissatisfaction, fairness, and locality - heterogeneous users

Metrics	FIFO	Fair	COSHH
Fairness	0.447	0.447	0.447
Locality (%)	97	97	97

Table 7.5. Fairness and locality - homogeneous users

7.4.2 Case Study 2: Homogeneous-Large

Here , an experiment is run in a homogenous cluster and workload in which all the jobs are of large size. A workload consisting of 100 Large data summary jobs was used (Table 7.1) . The evaluation was performed for both homogeneous and heterogeneous users.

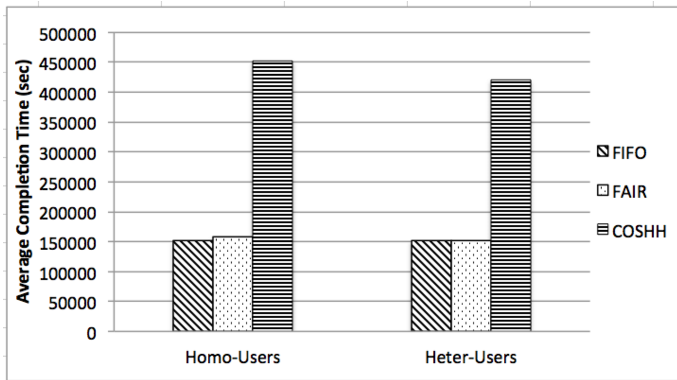


Figure 7.4. Average completion time - Homogeneous-Large

Figure 7.4 presents the average completion times for this system. Because the jobs are large, the scheduling complexity problem in the COSHH and the Fair Sharing algorithms leads to increases in their average completion times. The scheduling overheads are presented in Figure 7.5. As the COSHH algorithm suggests all job classes as the best choice for each resource, and all jobs are large (with a large number of tasks), the sort and search spaces are large. Therefore, the scheduling time

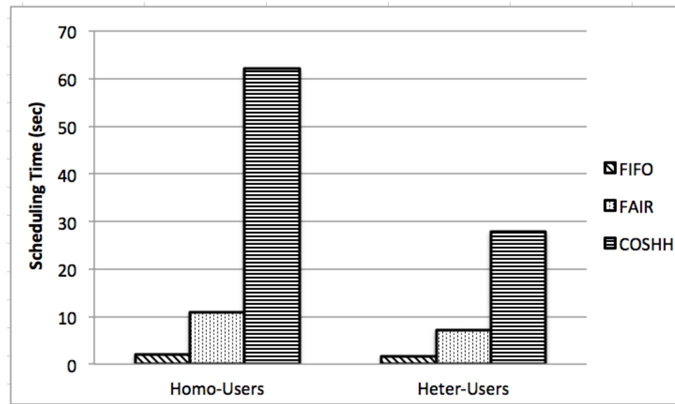


Figure 7.5. Scheduling time - Homogeneous-Large

of the COSHH algorithm is higher compared to COSHH for the small homogeneous workload. This leads to an increase in average completion time for the COSHH algorithm.

Tables 7.6 and 7.7 present the fairness, dissatisfaction, and locality of the large homogeneous Hadoop cluster for heterogeneous and homogeneous users, respectively. The results show the competitive performance of the Fair Sharing and the COSHH algorithms for these metrics.

Metrics	FIFO	Fair	COSHH
Dissatisfaction	8.345	5.470	6.954
Fairness	4.183	1.753	1.870
Locality (%)	97	97	96

Table 7.6. Dissatisfaction, fairness, and locality - heterogeneous users

Metrics	FIFO	Fair	COSHH
Fairness	0.278	0.247	0.322
Locality (%)	97	97	97

Table 7.7. Fairness and locality - homogeneous users

7.5 Evaluation: Heterogeneous Hadoop System

In this section, experimental results are provided to analyze the performance of schedulers in more heterogeneous environments. In these experiments, a cluster of six heterogeneous resources is used as presented in Table 7.2. Each experiment is performed for both heterogeneous and homogeneous users.

7.5.1 Case Study 3: Heterogeneous-Small

This case study evaluates the schedulers when the system is heterogeneous, and the proportion of small jobs is high. This workload is similar to the Yahoo! workload, where the arrival rates of small jobs are higher. This case study is considered for the evaluations in Section 6.7, and here it is further analyzed in both heterogeneous and homogeneous users settings. Figure 7.6 presents the average completion times for this system when the users are either homogeneous or heterogeneous.

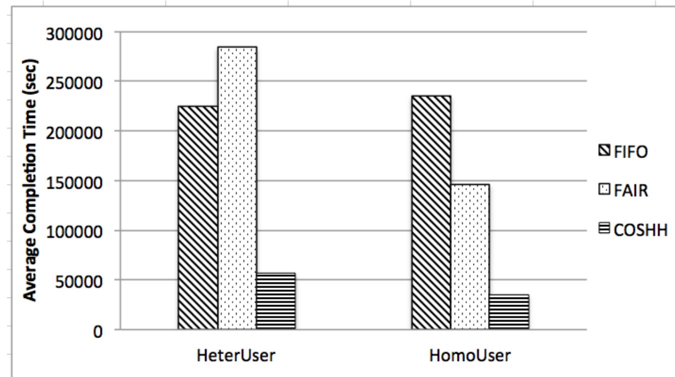


Figure 7.6. Average completion time - Heterogeneous-Small

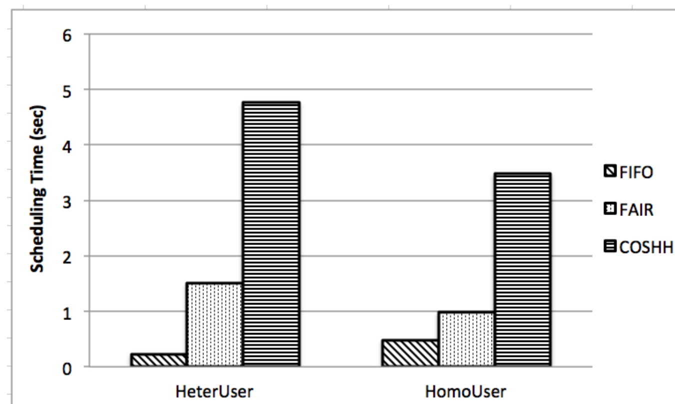


Figure 7.7. Scheduling time - Heterogeneous-Small

As the Fair Sharing algorithm does not have the problem of Small Jobs Starvation, it is expected to have better average completion time for this scheduler than for the FIFO algorithm. However, in the case of heterogeneous users, because the minimum shares are defined based on the job sizes, and the Fair Sharing algorithm first satisfies the minimum shares, it causes Small Jobs Starvation in this scheduler. As a result, most of the small jobs are executed after satisfying the minimum shares of the larger jobs, and the completion times of the small jobs (the majority of the jobs in

this workload) are increased. As the COSHH algorithm solves the Resource and Job Mismatch problem, it leads to 74.49% and 79.73% improvement in average completion time over the FIFO algorithm, and the Fair Sharing algorithm, respectively.

In the case of homogeneous users, where there is no minimum share defined, the Fair Sharing algorithm achieves better average completion time than the FIFO algorithm. The Fair Sharing algorithm achieves a 37.72% smaller average completion time than the FIFO algorithm, and the COSHH algorithm reduces the average completion time of the Fair Sharing algorithm by 75.92%.

The overhead of the scheduling algorithms is presented in Figure 7.7. Because most of the jobs in this workload are small, and they have fewer tasks, the scheduling overheads are low. Fairness, dissatisfaction, and the locality of the algorithms are presented in Tables 7.8 and 7.9 for heterogeneous and homogeneous users, respectively. The results show that the Fair Sharing algorithm has the best performance in these metrics, followed by the COSHH algorithm.

Metrics	FIFO	Fair	COSHH
Dissatisfaction	8.618	$7.16E-04$	1.209
Fairness	4.974	0.965	2.779
Locality (%)	95.6	97.4	96.5

Table 7.8. Dissatisfaction, fairness, and locality - heterogeneous users

Metrics	FIFO	Fair	COSHH
Fairness	1.032	0.504	0.856
Locality (%)	94.9	97.9	98.1

Table 7.9. Fairness and locality - homogeneous users

7.5.2 Case Study 4: Heterogeneous-Large

In this case study, the schedulers are evaluated in a heterogeneous cluster with a greater proportion of large jobs. In this workload, the number of jobs from larger size Yahoo! classes (classes 4 and higher in Table 7.1) is greater than the number from the smaller size job classes. Figure 7.8 presents the average completion times for heterogeneous and homogeneous users. When the users are heterogeneous, the Fair Sharing algorithm achieves the best average completion time. The reason is that this algorithm satisfies the minimum shares first, where the minimum shares are defined based on the job sizes. As a result, minimum share satisfaction helps reduce the average completion time. The COSHH algorithm has the second best average completion time as a result of addressing the Resource and Job Mismatch problem. In this system, the Fair Sharing algorithm reduces the average completion time of the FIFO and the COSHH algorithms by 47% and 22%, respectively.

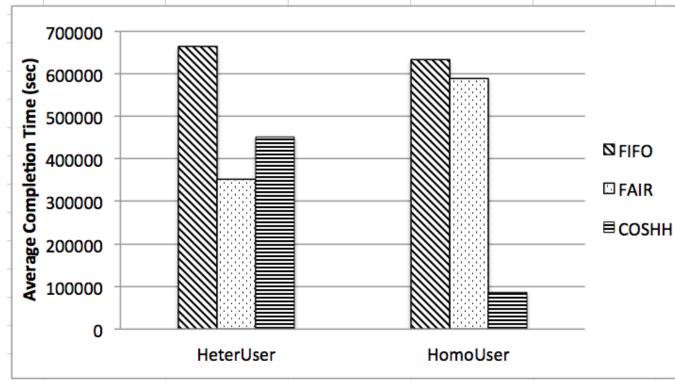


Figure 7.8. Average completion time - Heterogeneous-Large

However, when the users are homogeneous, and no minimum share is defined, the average completion time of the Fair Sharing algorithm becomes higher than the COSHH algorithm. As the Fair Sharing algorithm does not have the problem of Small Jobs Starvation, it achieves better average completion time than the FIFO algorithm. Based on the simulation results, the COSHH algorithm reduces the average completion time of the FIFO and Fair Sharing algorithms by 86.18% and 85.17%, respectively.

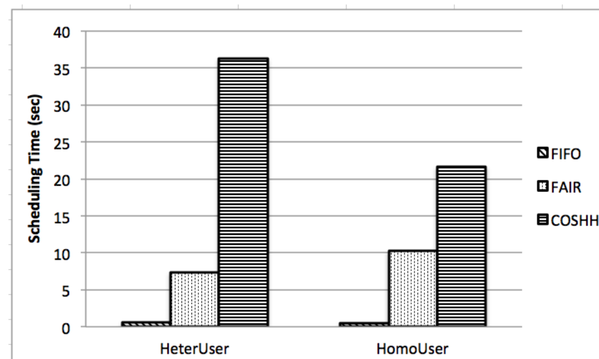


Figure 7.9. Scheduling time - Heterogeneous-Large

As the job sizes of this workload are larger than for the workload in Section 7.5.1, the scheduling times in Figure 7.9 are correspondingly higher. When the users are homogeneous the scheduling overhead of the COSHH algorithm is lower, as the classification and LP solving processes are shorter. When there are no minimum shares defined, the Fair Sharing algorithm has to spend more time on computing the fair shares, and sorting the jobs accordingly. This leads to higher scheduling times.

Tables 7.10 and 7.11 present the dissatisfaction, fairness, and locality when the users are heterogeneous and homogeneous, respectively. The COSHH algorithm has competitive performance with the Fair Sharing algorithm with respect to all three metrics.

Metrics	FIFO	Fair	COSHH
Dissatisfaction	8.223	$3.12E-04$	0.141
Fairness	6.523	1.651	0.697
Locality (%)	93.5	97.4	97.6

Table 7.10. Dissatisfaction, fairness, and locality - heterogeneous users

Metrics	FIFO	Fair	COSHH
Fairness	2.152	1.264	0.605
Locality (%)	95.3	98.0	98.1

Table 7.11. Fairness and locality - homogeneous users

7.5.3 Case Study 5: Heterogeneous-Equal

In this case study, an equal number of jobs are submitted from each of the Yahoo! classes. Figure 7.10 shows the average completion times when the users are homogeneous and heterogeneous. As the jobs and resources are all heterogeneous, it is important to consider the Resource and Job Mismatch problem in making scheduling decisions.

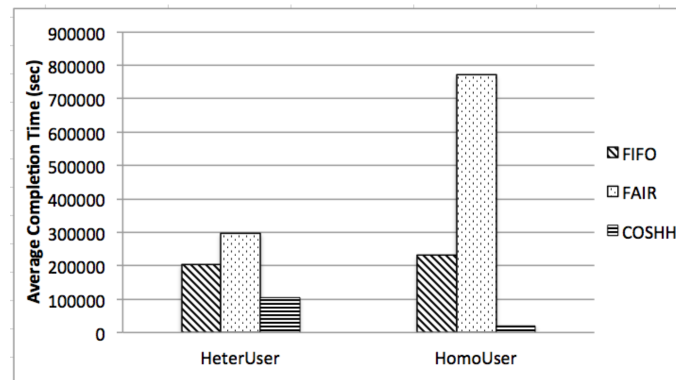


Figure 7.10. Average completion time - Heterogeneous-Equal

The COSHH algorithm achieves the best average completion time compared to the other schedulers. Because the arrival rates of all job classes are similar, the Sticky Slot problem in the Fair Sharing algorithm happens with higher frequency. Therefore, the Fair Sharing algorithm has larger average completion time than the FIFO algorithm. The COSHH algorithm reduces the average completion time of the FIFO and Fair Sharing algorithms by 49.28% and 65.24%, respectively. In the case of homogeneous users, where no minimum shares are assigned, the Fair Sharing algorithm has the highest average completion time, which is caused by the Sticky Slot problem. When the users are homogeneous, the COSHH algorithm reduces the average completion time of the FIFO and Fair Sharing algorithms by 90.28% and 97.29%, respectively. When the users do

not have minimum shares, the COSHH algorithm has just one class. Therefore, its overhead is reduced, which leads to a greater reduction in the average completion times.

The overheads in Figure 7.11 show that the improvement of average completion time in the COSHH scheduler is achieved at the cost of increasing the overhead of scheduling. The additional 10 second overhead for the COSHH algorithm, compared to the improvement for average completion time (which is around 200K seconds) is negligible. Further studies in Chapter 8 will show that even if the number of resources in the Hadoop cluster scales up, the COSHH algorithm can still provide a similar level of improvement in the average completion time. Because both the Fair Sharing and the COSHH algorithms search over the users and jobs to satisfy the minimum shares and fairness, they both have higher scheduling times than the FIFO algorithm.

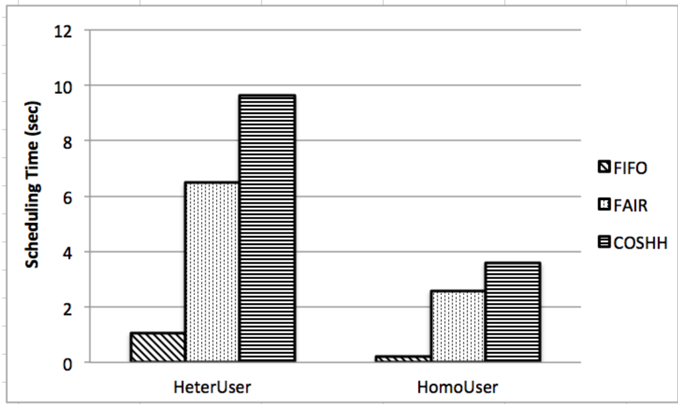


Figure 7.11. Scheduling time - Heterogeneous-Equal

Metrics	FIFO	Fair	COSHH
Dissatisfaction	8.287	0.0158	0.0247
Fairness	5.961	2.939	2.309
Locality (%)	93.7	95.2	87.6

Table 7.12. Dissatisfaction, fairness, and locality - heterogeneous users

Metrics	FIFO	Fair	COSHH
Fairness	1.195	1.127	1.085
Locality (%)	92.5	97.7	98.6

Table 7.13. Fairness and locality - homogeneous users

Tables 7.12 and 7.13 present the dissatisfaction, fairness, and locality when the users are heterogeneous and homogeneous, respectively. The COSHH algorithm has competitive performance with the Fair Sharing algorithm in improving the fairness and dissatisfaction.

7.6 Guidelines for Scheduler Selection

The provided experimental results and analysis show that a scheduler should be selected based on the heterogeneity levels of the Hadoop factors. Figure 7.12 presents guidelines suggested by the observations in this chapter. The main performance metric used for determining these guidelines is the average completion time. However, the selected algorithms are also either competitive with or better than the other schedulers with respect to the other performance metrics. To determine small size jobs, a threshold is defined based on the mean job execution times on resources and the time between heartbeats from the system. In these experiments, if the mean execution time of a job is less than the interval between two heartbeats, the job is considered to be small. However, the threshold can be customized for different Hadoop systems. How to do this (and which features are important to take into consideration) would be a useful topic for further work.

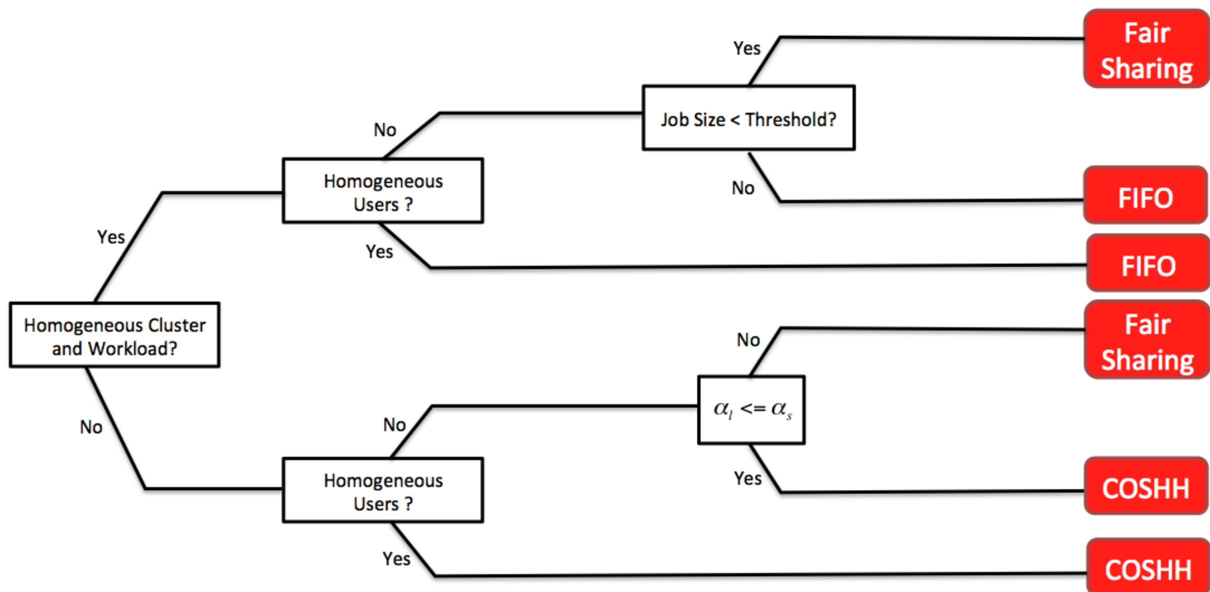


Figure 7.12. Suggested schedulers. Here α_l and α_s are the arrival rates for large and small jobs, respectively.

It should be noted that these guidelines are not tailored or defined for just the experiments in this chapter. The performance issues discussed for different algorithms happen in any case that the specified relations in terms of heterogeneity and job sizes hold. To evaluate the proposed guidelines, another real Hadoop workload is used, as presented in Table 7.14. The workload contains 100 jobs of a trace from a cluster at Facebook (FB), which is the same Facebook workload used for evaluations in Section 6.7. In the evaluation, there are 10 homogeneous users with zero minimum shares and equal priorities. Moreover, there are 10 heterogeneous users as presented in Table 7.15.

Each user submits jobs from one category in Table 7.14. The experimental environment here is identical to that in Section 7.3.

Job Categories	Duration (sec)	Job	Input	Shuffle	Output	Map Time	Reduce Time
Small jobs	32	126	$21KB$	0	$871KB$	20	0
Fast data load	1260	25	$381KB$	0	$1.9GB$	6079	0
Slow data load	6600	3	$10KB$	0	$4.2GB$	26321	0
Large data load	4200	10	$405KB$	0	$447GB$	66657	0
Huge data load	18300	3	$446KB$	0	$1.1TB$	125662	0
Fast aggregate	900	10	$230GB$	$8.8GB$	$491MB$	104338	66760
Aggregate and expand	1800	6	$1.9TB$	$502MB$	$2.6GB$	348942	76736
Expand and aggregate	5100	2	$418GB$	$2.5TB$	$45GB$	1076089	974395
Data transform	2100	14	$255GB$	$788GB$	$1.6GB$	384562	338050
Data summary	3300	1	$7.6TB$	$51GB$	$104KB$	4843452	853911

Table 7.14. Job categories in Facebook trace. Map time and Reduce time are in Task-seconds (Chen et al. [2011]).

User	$MinimumShare$	$Priority$
U_1	5	1
U_2	0	2
U_3	0	2
U_4	5	1
U_5	10	2
U_6	15	1
U_7	4	2
U_8	10	1
U_9	10	1
U_{10}	15	1

Table 7.15. Heterogeneous Users in FB workload

7.6.1 Homogeneous Hadoop

Figures 7.13-7.15 show the average completion time and the scheduling overhead in two case studies of homogeneous Hadoop systems. As the average completion time of all the algorithms in the Homogeneous-Small case are almost the same (around 98.8 seconds), its corresponding chart is not included in the figures. The results confirm the observations for the Yahoo! workloads. The guideline selects the FIFO algorithm when the system is homogeneous in all three factors. When the job size is small and the users are heterogeneous, the guideline suggests the Fair Sharing algorithm to improve the fairness.

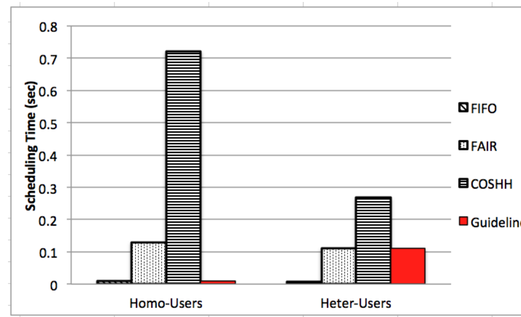


Figure 7.13. Scheduling time - Homogeneous-Small

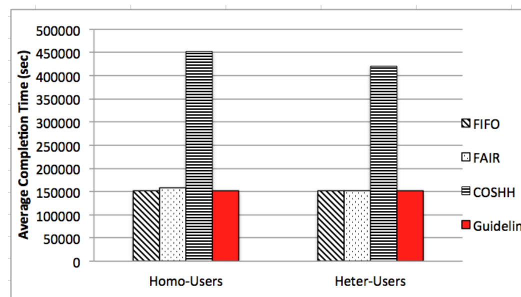


Figure 7.14. Average completion time - Homogeneous-Large

7.6.2 Heterogeneous Hadoop

Figures 7.16-7.21 show the average completion times and the scheduling times in the three case studies involving heterogeneous systems. In these experiments, the COSHH algorithm is the recommended scheduler in the majority of cases.

7.7 Related Work

Desired performance levels can be achieved by an appropriate submission of jobs to resources based on the system heterogeneity. The previous chapter provided an overview of Hadoop schedulers, and this section concentrates on the current Hadoop schedulers with respect to heterogeneity.

The main concern in most popular Hadoop schedulers is to quickly multiplex the incoming jobs on the available resources. Therefore, they use less system parameters and state information, which makes these algorithms an appropriate choice for homogeneous Hadoop systems. However, a scheduling decision based on a small amount of parameters and state information may cause some challenges such as less locality, and neglecting the system heterogeneity. Later proposed algorithms, such as (Aboulnaga et al. [2009]), improve scheduling decisions by providing the system parameters and state information as an input to the scheduler. However, the poor adaptability, and the large overhead of this algorithm (which is defined based on virtual machine scheduling),

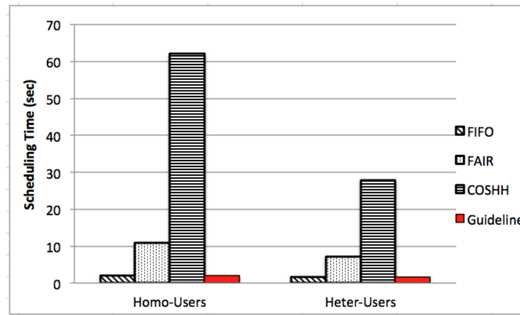


Figure 7.15. Scheduling time - Homogeneous-Large

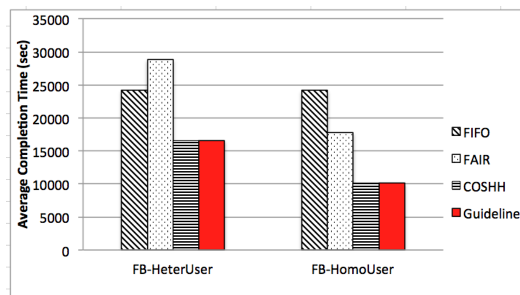


Figure 7.16. Average completion time - Heterogeneous-Small

make it an impractical choice for the systems considered in this research.

There are a number of Hadoop schedulers developed for restricted heterogeneous systems such as Dynamic Priority (DP) (Sandholm and Lai [2010]) and Dominant Resource Fairness (DRF) (Ghodsi et al. [2011]). The former is a parallel task scheduler which enables users to interactively control their allocated capacities by dynamically adjusting their budgets. The latter addresses the problem of fair allocation of multiple types of resources to users with heterogeneous demands. Finally COSHH (Rasooli and Down [2011]) is specifically proposed for heterogeneous environments.

This chapter evaluates Hadoop schedulers to propose heterogeneity-based guidelines. Although COSHH has shown promising results for systems with various types of jobs and resources, its scheduling overhead can be a barrier for small and homogeneous systems. DP was developed for user-interactive environments, differing from our target systems. Similarly, DRF was initially considered to be used instead of COSHH, but DRF only considers heterogeneity in the user demands while ignoring resource heterogeneity.

7.8 Conclusion

This chapter studies three key Hadoop factors, and the effect of heterogeneity in these factors on the performance of Hadoop schedulers. Performance issues for Hadoop schedulers are analyzed

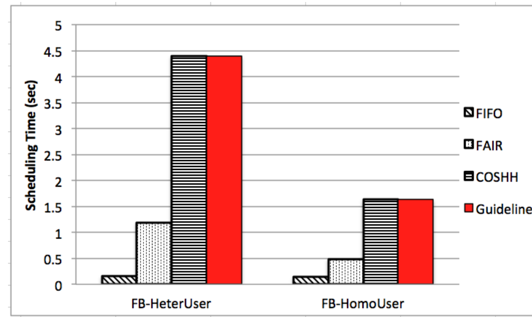


Figure 7.17. Scheduling time - Heterogeneous-Small

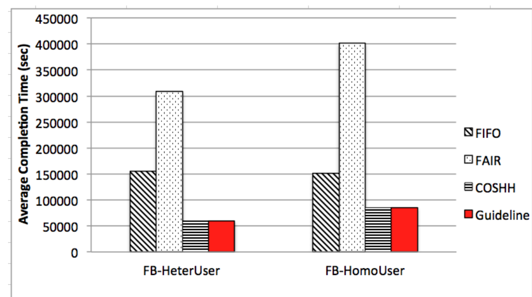


Figure 7.18. Average completion time - Heterogeneous-Equal

and evaluated in different heterogeneous and homogeneous settings. Five case studies are defined based on different levels of heterogeneity in the three Hadoop factors. Based on these observations, guidelines are suggested for choosing a Hadoop scheduler according to the level of heterogeneity in each of the factors considered.

So far all of the provided analysis and experiments for Hadoop have been performed in a fixed size Hadoop system, where the number of jobs and resources were not changed. However, one critical issue is scalability - Hadoop systems must perform well when the numbers of jobs and resources increase. The next chapter performs a scalability analysis of the Hadoop schedulers.

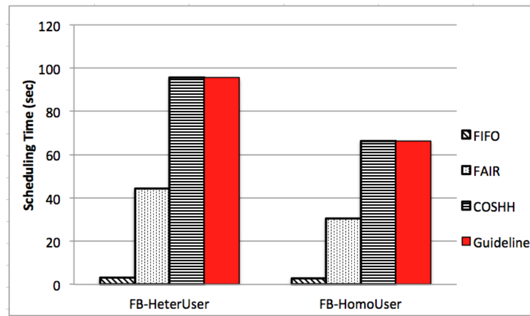


Figure 7.19. Scheduling time - Heterogeneous-Equal

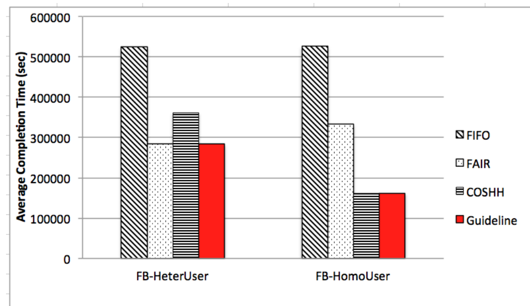


Figure 7.20. Average completion time - Heterogeneous-Large

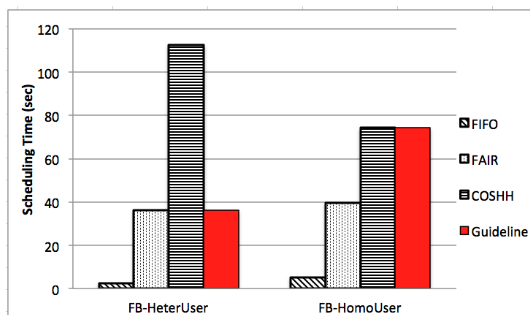


Figure 7.21. Scheduling time - Heterogeneous-Large

Chapter 8

A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems [1]

The scalability of Cloud infrastructures has significantly increased their applicability. Hadoop, which provides efficient processing of Big Data, is being used widely by most Cloud providers. Hadoop schedulers are critical elements for providing desired performance levels. There is a considerable challenge to schedule the growing number of tasks and resources in a scalable manner. Moreover, the potential heterogeneous nature of deployed Hadoop systems tends to increase this challenge. This chapter analyzes the scalability of widely used Hadoop schedulers including FIFO and Fair Sharing and compares them with the COSHH scheduler. Based on the introduced insights, a hybrid solution is proposed, which selects appropriate scheduling algorithms for scalable and heterogeneous Hadoop systems with respect to the number of incoming jobs and available resources.

8.1 Introduction

Cloud computing promises three distinct characteristics: elastic scalability, pay as you go, and manageability (Armbrust et al. [2010]). The advantages of Cloud computing have led to a significant increase in diversity and scale of cloud applications. One of the fastest growing applications is Big Data analysis (Agrawal et al. [2011]). The scalability and fault tolerance

[1]This chapter is mostly based on the following paper: A. Rasooli and D. G. Down, *A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems*, **In proceeding of the 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS), Co-located with Supercomputing 2012**, Salt Lake City, USA, November 12th, 2012.

in Cloud computing make it a great solution for these applications. However, the storage and processing requirements of Big Data applications make it extremely challenging to provide desired performance levels.

A common enterprise practice is to have a private Hadoop system installed on an intranet. In these Hadoop systems, the jobs and resources may change significantly during a day. As the experiments in this thesis show, a single scheduling algorithm may not provide the best performance in terms of average completion time. This chapter proposes a hybrid approach which uses alternative scheduling algorithms for specific situations. This approach considers average completion time for submitted jobs as the main performance metric. The proposed hybrid scheduler is based on the three Hadoop schedulers that we have considered throughout this thesis: FIFO, Fair Sharing, and COSHH. The hybrid scheduler chooses the best scheduling algorithm for different scales of jobs and resources to address average completion time and fairness.

The remainder of this chapter is organized as follows. Section 8.2 discusses the scalability of Hadoop systems. In Section 8.3 some of the motivations for proposing a scalable Hadoop scheduler are discussed. Section 8.4 presents important performance issues that Hadoop schedulers should consider for scalability in the face of heterogeneity. In Section 8.5 experimental results and analysis are provided. The proposed hybrid solution is introduced and analyzed in Section 8.6. Finally, conclusions are provided in the last section.

8.2 Scalability in Hadoop Systems

Most enterprise Hadoop systems have a higher load during the day, and lighter load during the evening. Similarly, resource numbers are subject to change at different times. Consequently, using a non-scalable Hadoop system could lead to resource underutilization during off-peak hours or resource overloading and poor performance during peak hours. Scalability of Cloud systems makes it possible for Hadoop systems to scale up and down based on the load to improve the utilization. A scalable Hadoop system considers both job numbers and complexity as well as the number of available resources to provide sufficient flexibility to adapt.

The heterogeneity of a Hadoop system can make scalability more complicated and challenging for the schedulers. The challenges of heterogeneity discussed in the previous chapter can be magnified as the numbers of jobs and resources in the system grow. Moreover, heterogeneity is increasing in current Hadoop systems as a result of new and emerging applications. Reported analysis on Hadoop systems found their workloads extremely heterogeneous with very different execution times (Chen et al. [2012]). Moreover, the number of small jobs (with short execution times) exceeds larger size jobs in typical Hadoop workloads such as the Facebook and Cloudera workloads discussed in (Chen et al. [2012]). This chapter analyzes the Hadoop schedulers in

scalable and heterogeneous environments. For this purpose, the average completion time reduction is addressed as the main goal, while considering the impact of fairness and scheduling overhead.

8.3 Motivation

A job scheduler is an essential component of every Hadoop system. Some of the main motivations of the proposed scheduling solution are listed in the following subsections.

8.3.1 Scalable Hadoop Schedulers

In a Hadoop system the numbers and features of jobs and resources may fluctuate at any time. Therefore, selecting an appropriate scheduling algorithm for a scalable heterogeneous Hadoop system is critical to achieve a desired performance level. However, scalability is for the most part a neglected issue in most currently used Hadoop schedulers. Facebook uses the two well-known schedulers, Fair Sharing and Delay Scheduler (Zaharia et al. [2009, 2010]). Delay Scheduler (Zaharia et al. [2010]) is a complementary algorithm for Fair Sharing to improve data locality. The Yahoo! large Hadoop cluster uses the Capacity scheduler (Apache Hadoop Capacity Scheduler [2010]). It provides capacity guarantees for queues while having elasticity in the sense that unused capacity of a queue can be harnessed by overloaded queues. Based on the analysis and experiments in (Sandholm and Lai [2010]), the Fair Sharing scheduler could not handle the experimental workload for two concurrent queues, whereas the Capacity scheduler was not able to handle the workload with ten queues. The results in (Sandholm and Lai [2010]) confirm the poor scalability of the Fair Sharing, Delay, and Capacity schedulers. Although these algorithms perform well in terms of fairness, they can provide poor performance when the system scales up in terms of the number of jobs.

8.3.2 Heterogeneity Aware Scheduling

As discussed before, heterogeneity is an increasing factor in a Hadoop system, which can be categorized at three levels: cluster, workload, and users. The scalability of Hadoop schedulers should be provided along with consideration of system heterogeneity. The default Hadoop scheduler, FIFO, can be considered as a scalable scheduler. However, as discussed in the previous chapter, both the FIFO and Fair Sharing schedulers neglect resource and job heterogeneity in their decisions. Moreover, there are a number of Hadoop schedulers developed for restricted scalable systems such as Dominant Resource Fairness (DRF) (Ghodsi et al. [2011]). This scheduler addresses the problem of fair allocation of multiple types of resources to users with heterogeneous demands. However, it only considers heterogeneity in the user demands while ignoring resource

heterogeneity. This can lead to poor average completion time of this scheduler in a heterogeneous Hadoop system.

In (Tang et al. [2012]), a task execution time model is proposed by evaluating task data processing unit times and data transfer unit times. This model is used to compute the number of map and reduce tasks which should be scheduled to satisfy deadline constraints. Based on the insights, a flexible, fine grained, dynamic and coordinated resource management framework, called MROrchestrator, is designed. Results from the implementation of MROrchestrator demonstrate the scheduler can lead to increases in resource utilization. However, this scheduler is only defined based on homogeneous Hadoop clusters.

8.3.3 Considering Critical Hadoop Performance Metrics

There are number of Hadoop schedulers which are designed for scalable systems. However, these schedulers neglect the critical Hadoop performance metrics such as fairness, locality, and average completion times. For instance, the Hadoop on Demand (HOD) (Apache [2007]) scheduler allows users to meet their changing demands over time. Increasing the number of users in this scheduling system is handled by scaling up the number of private clusters. However, this approach failed in practice due to violating the locality designed in the original MapReduce scheduler, and resulted in poor system utilization.

The experiments in (Sandholm and Lai [2010]) show better scaling of the Dynamic Priority (DP) (Sandholm and Lai [2010]) scheduler than the Capacity and Fair Sharing schedulers in the number of queues. As the Dynamic Priority scheduler permits more queues, it allows providers to assign more service levels. The scalability of this scheduler is due to its light weight design. However, this scheduler requires users to have accurate information about their job's resource requirements. As this is not true in most Hadoop systems, performance degradation may result.

8.3.4 Avoiding Large Scheduling Overhead

In (Sandholm and Lai [2009]) scheduling is considered in virtual machine hosted Hadoop clusters. They generally addressed the problem of how to scale up and down a set of virtual machines to complete jobs more cost effectively and faster, based on job and resource information. This approach performs well if each user works with a separate data set. However, when groups of users share large data sets, it leads to significant overhead to load the data into multiple virtual clusters. Moreover, if file system clusters are shared, the data locality reduction issue (similar to Hadoop On Demand) could add several challenges. Similarly, the proposed scheduler in (Qin et al. [2009]) uses kernel-level virtualization techniques to reduce resource contention among concurrent MapReduce jobs. As Hadoop is very I/O intensive both for file system access and MapReduce

scheduling, virtualization incurs a high overhead in this system. Although COSHH has shown promising results for systems with various types of jobs and resources, its scheduling overhead can be a barrier for small and under-loaded systems. This was one of the main motivations for composing different schedulers to provide a hybrid scheduler for scalable and heterogeneous systems.

8.3.5 Scheduling for the MapReduce model

In (Cardona et al. [2007]), the MapReduce scheduling model is extended to account for heterogeneity of the compute resources in terms of availability and CPU performance, common in large scale Grid systems. The Mars system (He et al. [2008]) implements MapReduce optimizations on GPU platforms mainly by aggressively taking advantage of the massive threading capacity. A large number of map and reduce tasks can thus be physically collocated but run in multiple threads. A similar extension is implemented in the Phoenix system (Ranger et al. [2007]). However, one issue of these approaches is that they can not enforce the different priorities of users.

Mesos (Hindman et al. [2011]) is a resource scheduling manager that provides fair shares of resources across diverse cluster computing frameworks like Hadoop and Message Passing Interface (MPI). Next Generation MapReduce (NGM) (Murthy [2011]) has been introduced as a new architecture of Hadoop MapReduce that includes a generic resource model for efficient scheduling of cluster resources. NGM replaces the default fixed size slot with another basic unit of resource allocation called a resource container. Condor (Litzkow et al. [1988]) is another resource scheduling manager that can potentially host Hadoop frameworks.

As the focus of this thesis is on providing scheduling solutions for the clusters designed based on the original Hadoop system, this chapter only considers the schedulers defined for Hadoop. Chapter 7 introduced the common problems of schedulers in the heterogeneous Hadoop systems. As the scalability issues are magnified in heterogeneous systems, this chapter uses the results of the previous chapter, and extends them by considering the scalability issues. To analyze the behaviour of Hadoop schedulers in different scalable and heterogeneous Hadoop configurations, we continue the use of the three Hadoop schedulers: FIFO, Fair Sharing, and COSHH. Each of these schedulers considers one or more of the motivations listed in Sections 8.3.1 - 8.3.5.

8.4 Performance Issues

The key performance issues for different schedulers in heterogeneous Hadoop systems were introduced in Section 7.1, and are summarized as follows:

- Problem I. Small Jobs Starvation: in a heterogeneous Hadoop workload, jobs have different execution times. For such workloads, the algorithms that do not take into account job sizes have the problem of small jobs potentially get stuck behind large ones.
- Problem II. Sticky Slots: this problem arises when the scheduler assigns a job to the same resource at each heartbeat. The problem can significantly increase average completion times, when an inefficient resource is selected for a job.
- Problem III. Resource and Job Mismatch: resources can have different features with respect to their computation or storage units. Moreover, jobs in a heterogeneous workload have different requirements. To reduce the average completion time, it is critical to assign the jobs to resources by considering resource features and job requirements.
- Problem IV: Scheduling Complexity: if the scheduling algorithm aims to improve its performance by using more complicated solutions, it may increase the overhead and complexity of scheduling. The increased overhead may even degrade the performance. This problem is more critical in homogeneous systems, however it can also cause challenges in heterogeneous systems.

These challenges are the main issues that each scheduler should consider in a scalable and heterogeneous Hadoop system, which are analyzed in more detail in the following section.

8.5 Analysis

This section analyzes the schedulers' performance for a real Hadoop workload in a scalable and heterogeneous environment. The result of this analysis is the justification of a hybrid approach for Hadoop scheduling. The provided experiments include two heterogeneous Hadoop systems: one with a varying number of jobs and one with a varying number of resources. For the purpose of extending the results in the previous chapters, the same experimental environment as Section 6.7 is used here with required modifications made to investigate scalability.

8.5.1 Case Study 1: Job Number Scalability

A heterogeneous cluster of six resources as defined in Section 6.7 is used in these experiments. First, to measure the performance of the schedulers in an under-loaded system, a Hadoop system with 5 jobs in its workload was evaluated. Then, multiple experiments were run by increasing the total number of jobs in the workload to investigate how performance scales with the number of jobs.

Figure 8.1(i) shows the average completion times for these experiments. Based on the results, when there is a small number of jobs in the workload, and the system is very lightly loaded, the COSHH algorithm has the highest average completion time. This trend continues until the number of submitted jobs reaches the total number of slots in the system (there are 31 map slots and 23 reduce slots on all six resources). After there are around 30 jobs in the workload, the system load reaches the point where all of the submitted jobs can not receive their required slots in the first round of scheduling. Therefore, they must wait until a slot becomes available. From this point on, the improvement in average completion time for the COSHH algorithm overcomes its scheduling overhead, and its average completion time is better than the other algorithms. Moreover, at around 30 jobs, the largest size jobs enter the system, which leads to a considerable increase in the average completion times for all of the schedulers.

The average completion time for the Fair Sharing algorithm is initially low. However, once the load in the system increases, and submitted jobs need to be assigned to resources at different heartbeats, its average completion time increases. This is because at each heartbeat, the Fair Sharing algorithm needs to perform sorting and searching over a large sort and search space. Moreover, when the system is underloaded there is almost no Sticky Slot problem in the Fair Sharing scheduler; however, by increasing the number of jobs, this problem is magnified. The Resource and Job Mismatch problem of the Fair Sharing scheduler is not dependent on the number of jobs, but the larger number of jobs leads to longer waiting times for small jobs. Both the magnified Resource and Job Mismatch and Sticky Slot problems of the Fair Sharing scheduler result in larger average completion times for this scheduler compared to the COSHH scheduler. In the case of the FIFO scheduler, there are two factors degrading its performance: Small Job Starvation and the larger ratio of small jobs in the workload. When the large jobs enter the system (between 20 and 30 total jobs), the average completion times significantly increase. However, when there are multiple small jobs in the workload, (between 40 and 50 total jobs), the average completion time decreases.

Figure 8.1(ii) shows the scheduling time for this experiment. The overheads of all algorithms increase as the number of submitted jobs increases. The growth rate for the COSHH algorithm is higher than the others as a result of its more complicated scheduling process. However, its growth rate decreases as the number of jobs increases. Generally the jobs in Hadoop workloads exhibit some periodic behaviour. The initial submitted jobs of a job class can cause a longer classification process. However, because subsequent jobs of the same job class do not need new classes to be defined, the overhead due to the classification process of the COSHH algorithm is reduced.

Figure 8.1(iii) shows the fairness for these experiments. Comparing the algorithms, the Fair Sharing algorithm has the best fairness, and the COSHH algorithm has competitive fairness with

the Fair Sharing algorithm. This order of fairness between the schedulers does not change by increasing the number of jobs; this is completely as expected.

8.5.2 Case Study 2: Resource Number Scalability

This case study varies the number of resources. The workload in these experiments consists of 100 jobs from the Yahoo! traces as presented in Section 6.10. To define different size clusters, six types of resources are used, as presented in Table 8.1. The initial experiment was started with six resources, one from each type. For each succeeding experiment, one resource was added to reach 102 resources for the final experiment (i.e. 17 resources of each type).

Resources	Slot		Memory	
	$slot\#$	$execRate$	$Capacity$	$RetrieveRate$
R_1	2	$5MHz$	$4TB$	$9Gbps$
R_2	2	$500GHz$	$400KB$	$40Kbps$
R_3	2	$500GHz$	$4TB$	$9Gbps$
R_4	2	$5MHz$	$4TB$	$9Gbps$
R_5	2	$500GHz$	$400KB$	$40Kbps$
R_6	2	$5MHz$	$400KB$	$40Kbps$

Table 8.1. Resource Types

Figure 8.1(iv) shows the average completion times for these experiments. Increasing the number of resources reduces the load in the system and improves the average completion time for all of the schedulers. However, this can reduce the chance of local execution for the jobs, which tends to increase the average completion time. Therefore, by increasing the number of resources, first the average completion time of the schedulers reduces until the number of resources reaches approximately 54. Beyond this point, the average completion times increase slightly, because of the locality issue. Moreover, increasing the number of resources can reduce the Sticky Slot and the Small Job Starvation problems in the Fair Sharing and FIFO schedulers, respectively. By increasing the number of resources, these problems are reduced, but still exist in the Fair Sharing and FIFO schedulers.

Figure 8.1(v) shows the scheduling times for the different schedulers. The overheads of the COSHH and the Fair Sharing algorithms get larger as the number of resources increases. The reason is the longer search and sort times in these algorithms. Moreover, the larger numbers of resources leads to an increase in the classification and LP solving times for the COSHH algorithm. The rate of increase in the COSHH algorithm is higher than the Fair Sharing algorithm. However, its growth rate decreases as the number of jobs increases.

Figure 8.1(vi) presents the fairness for these experiments. As the number of resources scales up, the fairness metric improves in all algorithms. However, the Fair Sharing algorithm achieves better performance in terms of fairness.

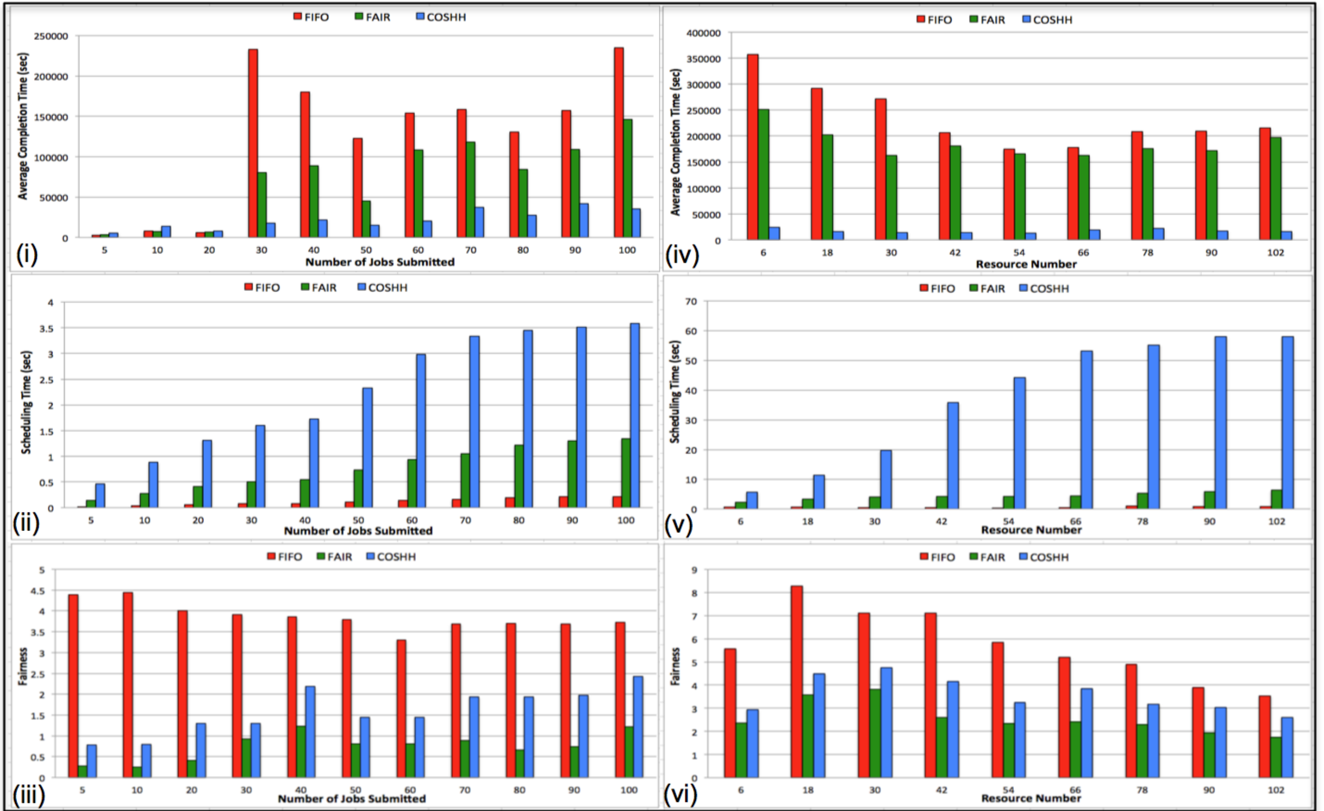


Figure 8.1. Performance Metrics for Yahoo! Workload. (i)-(iii): Scaling by Number of Jobs, (iv)-(vi): Scaling by Number of Resources

8.6 Hybrid Solution

Based on the experimental results and analysis, a hybrid scheduler is proposed (Figure 8.2). This scheduler is a combination of the three analyzed algorithms. The selector chooses an appropriate scheduler as the number of jobs and resources scales up or down. However, the overall solution is to use the COSHH algorithm when the system is overloaded (e.g., during peak hours), the FIFO algorithm for underloaded systems (e.g., after hours), and the Fair Sharing algorithm when the system load is balanced. We define the load to be balanced when the increased difference between the number of waiting tasks in two subsequent heartbeats is less than the total number of slots.

When the system is underloaded, and the number of free slots is greater than the number of waiting tasks, the scheduler switches to the FIFO algorithm. This can happen when the system has just started or during low load periods. Here, the simple FIFO algorithm can improve the average completion time with minimum scheduling overhead. However, as the system load increases such that the available number of slots is less than the number of waiting tasks, the hybrid scheduler

selects the Fair Sharing algorithm. A good example of this case is when the system has warmed up after starting, and the workload has not yet peaked. In this case, the FIFO algorithm may degrade the system performance with respect to both the average completion time and the fairness metrics. Moreover, as the system is not yet overloaded, using the complex COSHH algorithm can result in unacceptable overhead in terms of scheduling overhead and fairness. When the load increases such that the system is overloaded, and the number of waiting tasks is quickly increasing, the Fair Sharing algorithm can greatly increase the average completion time. Therefore, the scheduler switches to the COSHH algorithm, which improves the average completion time, while avoiding considerable degradation in the fairness metric. The selector needs to define two thresholds to determine the status of the system. Thresholds can be defined based on the system load. One possibility is to use the total number of slots, and the total number of tasks waiting in the scheduling queue, to define the threshold. In this chapter the thresholds are defined by comparing the waiting jobs in the queues and the total number of slots in the resources. When the number of waiting jobs is less than the number of slots, the system is considered to be underloaded. The threshold of detecting an overloaded system is defined to be the point that the increased difference between the number of waiting tasks in two subsequent heartbeats becomes greater than the total number of slots. Optimum thresholds can be obtained by performing some experiments and analysis on the target system. Other practical guidelines could be developed for this purpose, which are left for future work.

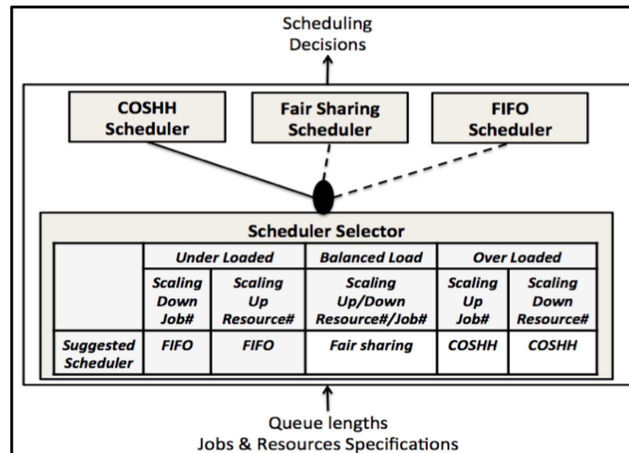


Figure 8.2. Suggested Hybrid Scheduler

To evaluate the proposed hybrid solution scheduler, another real Hadoop workload is used here. The workload contains 100 jobs of a trace from a cluster at Facebook, spanning six months from May to October 2009. It is the same Facebook workload used in Chapter 6, and it is provided in Table 8.2 again for ease of presentation. In the evaluation, there are 10 users with

zero minimum shares, and equal priorities. Each user submits jobs from one category in Table 8.2. The experimental environment is defined similar to that in Section 8.5.

Job Categories	Duration (sec)	Job	Input	Shuffle	Output	Map Time	Reduce Time
Small jobs	32	126	$21KB$	0	$871KB$	20	0
Fast data load	1260	25	$381KB$	0	$1.9GB$	6079	0
Slow data load	6600	3	$10KB$	0	$4.2GB$	26321	0
Large data load	4200	10	$405KB$	0	$447GB$	66657	0
Huge data load	18300	3	$446KB$	0	$1.1TB$	125662	0
Fast aggregate	900	10	$230GB$	$8.8GB$	$491MB$	104338	66760
Aggregate and expand	1800	6	$1.9TB$	$502MB$	$2.6GB$	348942	76736
Expand and aggregate	5100	2	$418GB$	$2.5TB$	$45GB$	1076089	974395
Data transform	2100	14	$255GB$	$788GB$	$1.6GB$	384562	338050
Data summary	3300	1	$7.6TB$	$51GB$	$104KB$	4843452	853911

Table 8.2. Job categories in Facebook trace. Map time and Reduce time are in Task-seconds (Chen et al. [2011]).

Figure 8.3(i)-(iii) shows the average completion time, the scheduling overhead, and fairness as the number of jobs scales. These results confirm our observations for the Yahoo! workloads. The hybrid scheduler uses the FIFO, the Fair Sharing, and the COSHH algorithms when the system is underloaded, balanced, and overloaded, respectively. When the number of jobs is scaling up, the first switch between schedulers happens at around 25 jobs in the workload. At this point the total number of waiting tasks becomes greater than the total number of slots (31 map slots and 23 reduce slots) on all six resources. The second switch between schedulers is when the number of jobs in the workload is at around 45. To determine this point, the scheduler monitors the increasing difference between the number of waiting tasks in the subsequent heartbeats, when this difference gets larger than the total number of slots, the scheduler switches to COSHH.

Figure 8.1(iv)-(vi) shows the average completion time, the scheduling time, and fairness, when the number of resources in the system varies. In these experiments, one switch between schedulers happens when the number of resources is at around 70. This is where the system moves from overloaded to balanced due to the increase in the number of resources.

Finally, it should be noted here that there is a transition period to observe the improved performance after each switch. The transition happens due to the tasks that have already been scheduled or are running as a result of the previous algorithm. The resources first need to complete their tasks assigned by the previous algorithm to be available for tasks scheduled by the newly selected algorithm. In these experiments, the transition period after switching from FIFO to Fair Sharing is around 250 sec, and after the switch from Fair Sharing to COSHH is around 847 sec. The transition time depends on the sizes of tasks which are currently being executed, but note that the duration here is negligible comparing to the average completion times.

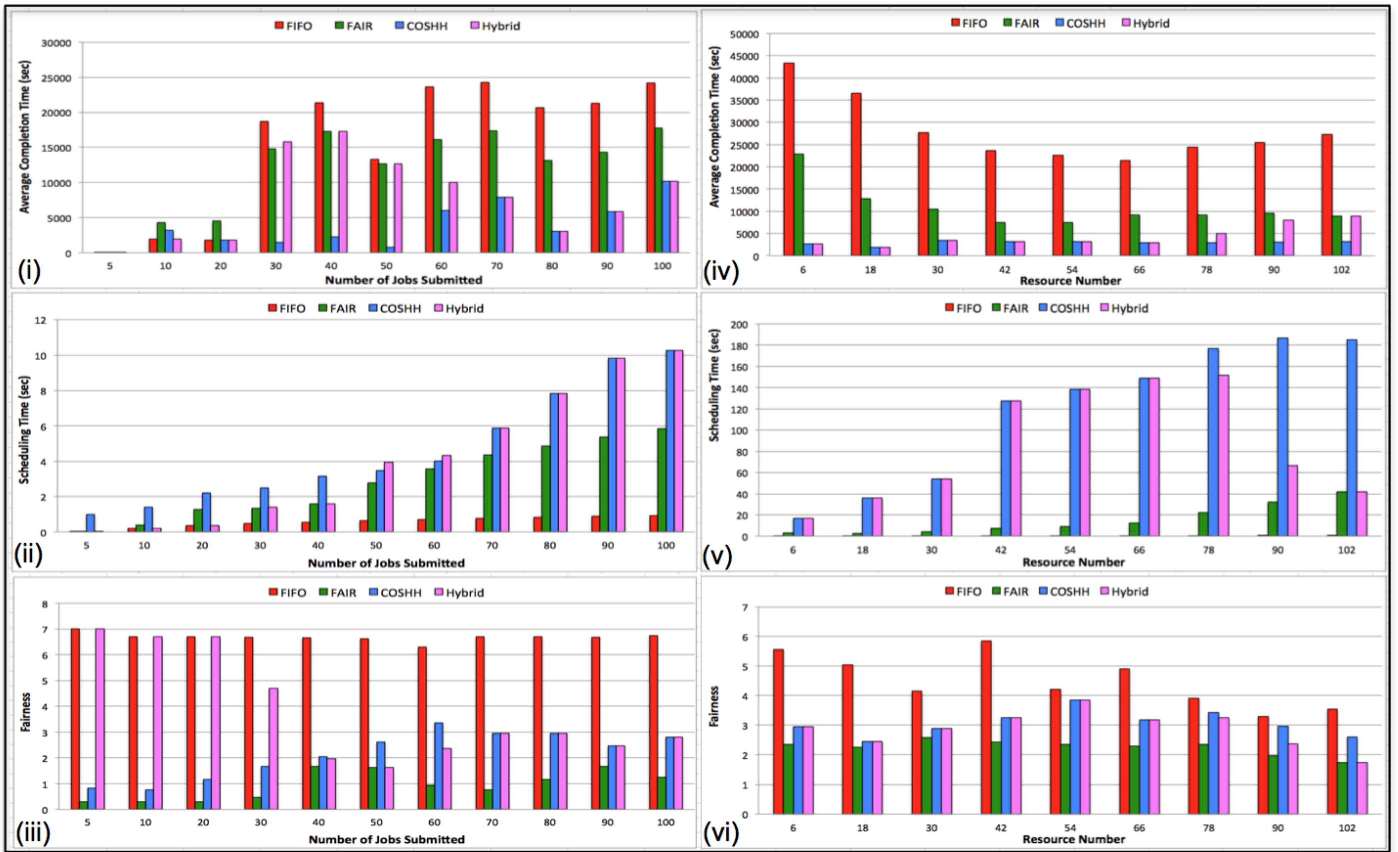


Figure 8.3. Performance Metrics for Facebook Workload. (i)-(iii): Scaling by Number of Jobs, (iv)-(vi): Scaling by Number of Resources

8.7 Conclusion

This chapter introduced a hybrid scheduler for scalable and heterogeneous Hadoop systems. The effect of scalability on the performance issues of Hadoop schedulers are analyzed and evaluated. These results suggested that a combination of the FIFO, Fair Sharing, and COSHH schedulers is effective, where the selection is based on the system load and available system resources.

One of the critical metrics in a Hadoop scheduler is satisfying the user minimum shares. As the minimum shares defined for each user may change over time, the Hadoop scheduler must handle these changes. The scheduler should satisfy the varying minimum share settings while preventing degradation of other key performance metrics, in particular average completion times. This issue along with the further evaluation of the COSHH scheduler on a real Hadoop cluster are discussed in the next chapter.

Chapter 9

The Effect of Minimum Shares on Performance of Hadoop Schedulers

There are a variety of users in a Hadoop system, differentiated based on features such as priority, the submitted job types, and required shares. Guaranteed minimum shares is a means to enforce priorities, without giving strict priorities to users. The scheduler is responsible for providing the defined minimum shares for the users. Moreover, the Hadoop scheduler can directly affect the system performance with respect to various metrics such as average completion times.

In a Hadoop system, the two critical processes of setting minimum shares, and selecting a scheduler are independent. The minimum shares are generally defined by the Hadoop providers and/or users to specify a guaranteed share for users at each point in time. The minimum share settings can be changed dynamically. On the other hand, the Hadoop scheduler is selected (generally once and permanently) by the system designer or administrator. The sensitivity of a Hadoop scheduler to changes in minimum shares can be addressed in two terms: speed of satisfying the new minimum shares, and the effect of new minimum share values on other performance metrics, such as the average completion time. While the scheduler should be concerned with minimum share satisfaction, its average completion time should not be significantly degraded by changing the minimum shares.

Hadoop providers may define the minimum shares based on different technical (such as job requirements) and non technical (such as business priorities) factors. For instance a Hadoop provider may determine the criticality of its advertisement jobs, and in turn define a corresponding minimum share. Different Hadoop schedulers may use various approaches when dealing with minimum shares. For instance, the FIFO scheduler make scheduling decisions independent of the minimum share values. On the other hand, the Fair Sharing algorithm is highly sensitive to the defined minimum shares. The COSHH scheduler simultaneously considers system heterogeneity

and minimum share satisfactions. This chapter analyzes the effects of minimum share modification on performance of these three Hadoop schedulers. These algorithms are selected as representatives of schedulers which consider minimum share satisfaction at different levels.

Based on the results, there are schedulers which despite their good performance in immediate minimum shares satisfaction, result in poor average completion times as a result of ignoring the system heterogeneity. Our analysis can be seen as counter to the intuition that defining a higher minimum share for users guarantees providing better performance for them.

This chapter introduces six case studies based on possible settings of the minimum shares (Section 9.1). The performance of Hadoop schedulers is analyzed in these case studies (Section 9.2). The results are evaluated and further discussed using a real Hadoop system with traces from a Facebook workload (Sections 9.3 and 9.4). Section 9.5 discusses related work, and Section 9.6 provides concluding remarks and a discussion of future work.

9.1 Minimum share effects on Hadoop schedulers

A minimum share is an input to a Hadoop system reflecting several factors such as pricing policies, job criticality, and system performance (Sandholm and Lai [2010]). Regardless of the factors that define the minimum shares, the ultimate goal is to provide better performance and completion time for jobs with high priority (jobs with minimum shares).

Hadoop schedulers consider the minimum shares in the job-resource assignments without having control over the defined minimum share settings. As the distribution of minimum shares is dynamic, and the Hadoop system is normally set-up with a specific (generally permanent) scheduler, the performance provided by the scheduler should be robust to changes in the minimum shares. The main focus of this chapter is to analyze how different minimum share distributions can affect the performance of Hadoop schedulers. For this purpose, six case studies are defined as follows. Two main technical factors for calculating the minimum shares are the job size and system heterogeneity. The case studies are defined base on these two factors.

1. Case 1 (Largest Job Priority): the user with the largest job size gets the largest minimum share, and most of the other users get little or no minimum share. A practical example of this case can be assigning the greatest minimum share to emergency backup jobs.

2. Case 2 (Large Jobs Priority): majority of the users are assigned a minimum share, where the distribution of the minimum shares is based on the job sizes. An example of this case is a Hadoop system with different large production jobs (jobs which generate revenue), and some small non-production jobs. In this system, the largest job gets the maximum minimum share, and the smallest jobs get no minimum share.

3. Case 3 (Equal Priority): the users are homogeneous with equal minimum shares and priorities. An example of this case is a Hadoop cluster specified for production jobs with the same size.
4. Case 4 (Short Jobs Priority): majority of the users receive a minimum share, where the distribution of minimum shares is the opposite of job sizes. As an example of this case consider a system which assigns large minimum shares to different small queries like interactive jobs and small shares to the (few) large jobs, such as long term analysis jobs.
5. Case 5 (Shortest Job Priority): only the user with the smallest job size gets the largest minimum share, and the rest of the users get almost no minimum share. An example of this case is a system with the largest minimum shares assigned to real time ad-hoc queries.
6. Case 6 (Homogeneous System): the system cluster, workload, and users (including the assigned minimum shares) are homogeneous. This case study is defined to analyze the scheduling algorithms in a homogeneous environment, where all the users have similar minimum shares.

There are a number of performance issues in the Hadoop scheduling algorithms which were introduced in detail in Chapter 7.1. These issues, presented as follows, can be magnified by different settings of minimum shares.

- **Problem I. Small Jobs Starvation.** This problem arises in a heterogeneous Hadoop workload, where the jobs have different execution times.
- **Problem II. Sticky Slots.** This problem arises when the scheduler assigns a job to the same resource at each heartbeat.
- **Problem III. Resource and Job Mismatch.** To reduce the average completion time, it is critical to assign the jobs to resources by considering resource features and job requirements.
- **Problem IV: Scheduling Complexity.** This problem can result from different features, such as gathering more system parameters and state information, and considering various factors in making scheduling decisions. In a fully homogeneous system, collecting state information and using a complex scheduler can add considerable overhead, which may not be compensated for by the more precise job and resource assignment.

9.2 Analysis

This section analyzes the performance of scheduling algorithms using the first five defined case studies (in Section 9.1). The sixth case study is analyzed in more detail in Section 9.3. For this purpose, an example system is used, which includes four heterogeneous resources and three users with five different settings for the minimum shares. The characteristics of the system are presented in the following (the choice of system size is only for ease of presentation, the same issues arise in larger systems):

- Task1, Task2, and Task3 represent three heterogeneous task types with the following mean execution times. Here, $m_t(T_i, R_j)$ is the mean execution time of task T_i on resource R_j.

$$m_t = \begin{bmatrix} 2.5 & 2.5 & 10 & 10 \\ 2.5 & 2.5 & 5 & 5 \\ 10 & 10 & 2.5 & 2.5 \end{bmatrix}$$

- Three users submit three jobs to the system, where each job consists of a number of similar tasks. Jobs arrive to the system in the following order: *Job1*, *Job2*, and *Job3*. Each user submits one job to the system as follows:

User1: Job1 (consists of 10 Task1)
User2: Job3 (consists of 20 Task3)
User3: Job2 (consists of 5 Task2)

This example includes five cases, each is a representative for the five minimum share case studies (Table 9.1).

Table 9.1. The minimum shares in each case study of the example

Cases	Minimum Shares		
	User1	User2	User3
Case1	0	3	0
Case2	1	2	0
Case3	0	0	0
Case4	1	0	2
Case5	0	0	3

Figure 9.1 presents the scheduling decisions of the FIFO algorithm. Job completion times are highlighted in the time line. As this algorithm does not take into account the minimum shares and system heterogeneity, the same scheduling decisions are made for all of the cases. The FIFO algorithm has the Resource and Job Mismatch problem, which leads to increasing the completion

time for some jobs. The main drawback of the FIFO algorithm is ignoring the minimum share satisfaction, which is a critical metric for Hadoop providers.

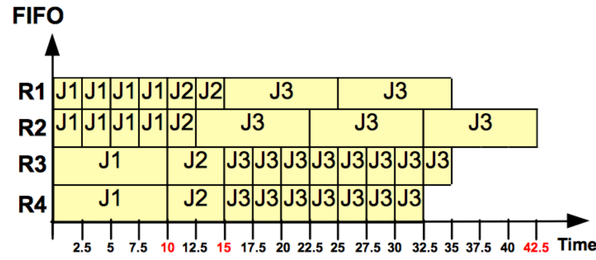


Figure 9.1. Job assignment by FIFO algorithm.

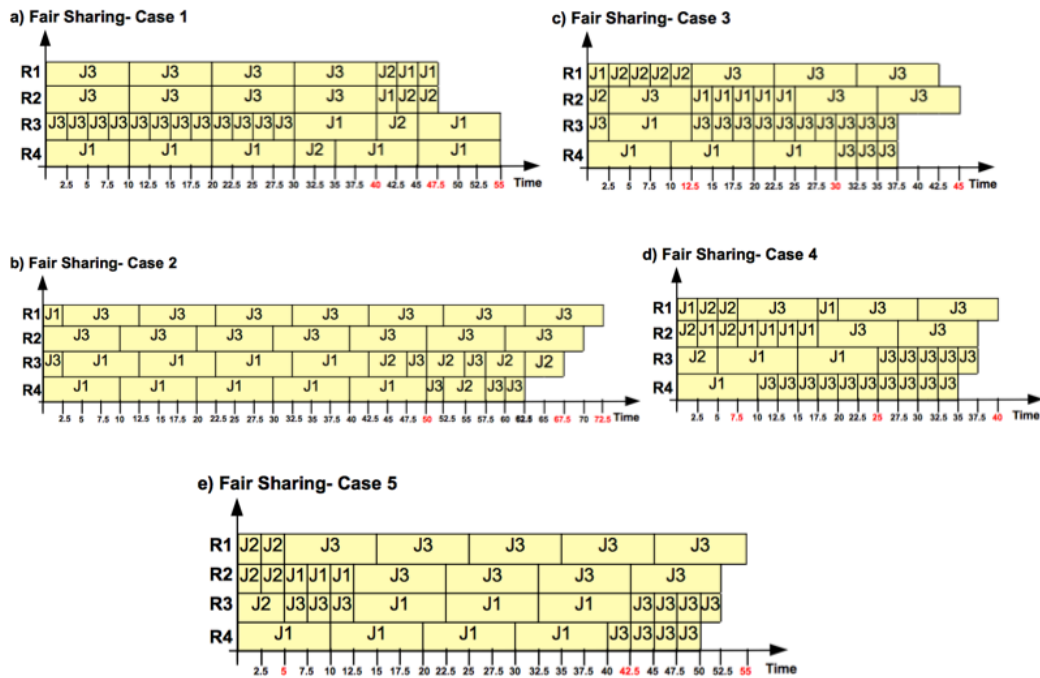


Figure 9.2. Job assignment by Fair Sharing algorithm in a) Largest Job Priority, b) Large Jobs Priority, c) Equal Priority, d) Short Jobs Priority, and e) Shortest Job Priority case studies in the example Hadoop system.

The scheduling decisions for the Fair Sharing algorithm in each of these case studies are presented in Figure 9.2. This algorithm assigns the guaranteed minimum shares to the users. However, two of the performance problems, the Resource and Job Mismatch problem and the Sticky Slots problem arise. The former problem appears in multiple places of the presented case studies, such as in assigning $J3$ to $R1$, and $J1$ to $R3$ in Figure 9.2. Where this problem arises, the assignment of jobs to resources without considering resource features and job requirements

leads to increasing the completion times. The Sticky Slots problem happens in all the presented case studies (such as repeated assignment of $J3$ to $R1$ and $R2$ in Figure 9.2). This problem can significantly increase the average completion times, when an inefficient resource is selected for a job.

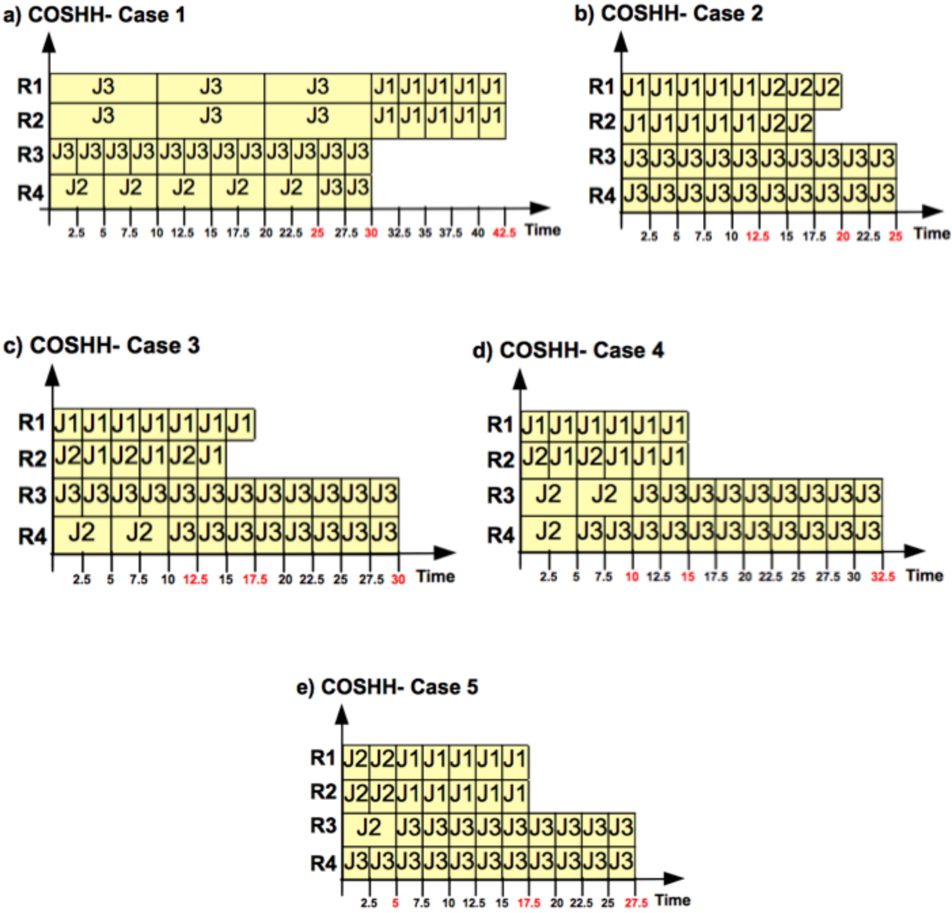


Figure 9.3. Job assignment by COSHH algorithm in a) Largest Job Priority, b) Large Jobs Priority, c) Equal Priority, d) Short Jobs Priority, and e) Shortest Job Priority case studies in the example Hadoop system.

We see that the Sticky Slots problem in the Fair Sharing algorithm can be magnified in some case studies. For example, in Case2, this problem has led to very poor performance. When a job is finished, the minimum share of its user drops. The immediate requirement of minimum share satisfaction in the Fair Sharing algorithm, forces the instant assignment of the same resource to the same job. This leads to increasing the number of Sticky Slots occurrences. When this problem is mixed with the Resource and Job Mismatch problem, it can significantly degrade the performance.

Figure 9.3 presents the scheduling decisions of the COSHH algorithm. The results show less fluctuation in the average completion time compared to the Fair Sharing algorithm. COSHH may be less successful in immediate minimum share satisfaction. However, it leads to better job completion times. Recall that the COSHH algorithm has two levels of classifications: minimum share satisfaction and fairness, respectively. Changing the minimum shares only affects the first level of classification.

The first level classification in COSHH varies for different case studies. For instance, in Case2, the COSHH algorithm classifies the jobs into two classes: First-Class1 and First-Class2, which contain Job1 and Job3, respectively. Then, it solves the following LP, which is defined using the execution rates of these classes on all the resources, and their arrival rates:

$$
\begin{aligned}
&\max \lambda \\
&\quad \text{s.t.} \\
&\qquad 0.04 \times \delta_{1,1} + 0.04 \times \delta_{1,2} + 0.01 \times \delta_{1,3} + 0.01 \times \delta_{1,4} \geq 1 \times \lambda, \\
&\qquad 0.02 \times \delta_{2,1} + 0.02 \times \delta_{2,2} + 0.08 \times \delta_{2,3} + 0.08 \times \delta_{2,4} \geq 1 \times \lambda, \\
&\qquad \sum_{i=1}^{3} \delta_{i,j} \leq 1, \ \textit{for all } j = 1, \ldots, 4, \\
&\qquad \delta_{i,j} \geq 0, \ \textit{for all } i = 1, \ldots, 3, \ \textit{and } j = 1, \ldots, 4.
\end{aligned}
$$

The optimization results suggest the following first classes for each resource. Resource3 and Resource4 are assigned to just First-Class2 to avoid the Resource and Job Mismatch problem.

Resource1:	{First-Class1, First-Class2}
Resource2:	{First-Class1, First-Class2}
Resource3:	{First-Class2}
Resource4:	{First-Class2}

In the second level of classification, the COSHH algorithm classifies the jobs into three classes: Class1, Class2, and Class3, which contain Job1, Job2, and Job3, respectively. This level of classification considers all of the jobs in the system, and it is independent of the minimum share settings. This scheduler solves an LP to find the best set of suggested job classes for each resource, as follows.

After computing the suggested sets, the COSHH algorithm considers fairness and minimum share satisfaction to assign a job to a resource. Although the COSHH algorithm assigns Job1 exclusively to Resource1 (Sticky Slots Problem), it does not increase the completion time of

Resource1: {Class1, Class2}
Resource2: {Class1, Class2}
Resource3: {Class2, Class3}
Resource4: {Class2, Class3}

Job1. The reason is that COSHH considers the execution times of jobs on resources in selecting Resource1 for Job1. This is one of the main advantages of the COSHH algorithm over Fair Sharing in a heterogeneous system. As in both levels of classification, the resources are assigned to the jobs by considering the heterogeneity, the completion time of the jobs are robust to different settings of the minimum share. This makes the COSHH algorithm a more reliable choice in heterogeneous Hadoop systems with dynamic minimum share settings. In the following, the effects of different minimum shares on schedulers are analyzed from two points of view:

- System Performance: Figure 9.4 compares the schedulers based on average completion time. As evidence to our initial insight, the average completion times depend (very significantly in some of the cases) on the minimum share settings. The overall average completion time for the COSHH algorithm is better than for the Fair Sharing algorithm. As the FIFO algorithm does not have the Sticky Slots problem, in this example it provides better average completion time than the Fair Sharing algorithm.

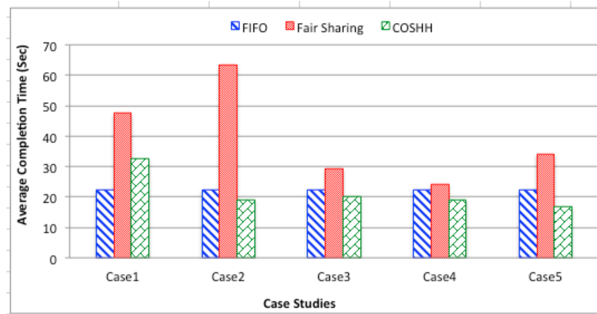


Figure 9.4. Average completion times of the schedulers in example case studies.

- High Priority Job Performance: as the main goal of the minimum shares is to improve the performance for the high priority users, Figure 9.5 compares the completion times of users with highest minimum shares. The results show that the COSHH algorithm is more successful in improving the performance for the jobs of users with highest minimum shares. Moreover, Figure 9.6 presents the average completion time for jobs with minimum shares assigned to them. The critical drawback of the FIFO algorithm in neglecting the minimum shares is exposed in Figures 9.5 and 9.6.

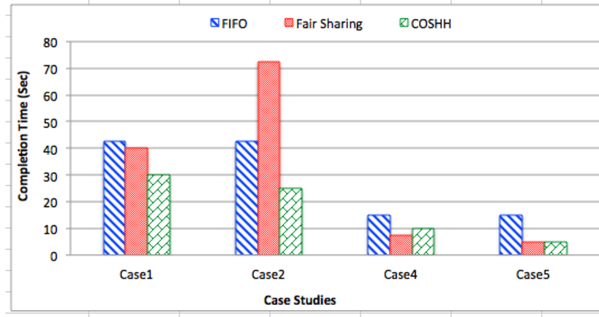


Figure 9.5. Completion time of the highest priority job in example case studies.

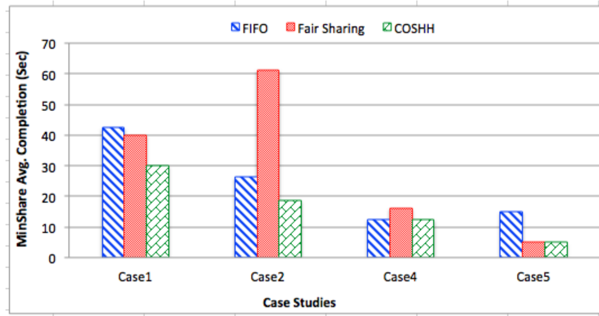


Figure 9.6. Average completion times of the jobs with minimum shares in example case studies.

9.3 Experimental Results

The performance of Hadoop schedulers in all six case studies is evaluated by running experiments using a real Hadoop workload on a Hadoop cluster.

9.3.1 Experimental Environment

In experiments with heterogeneous resources (Case1 to Case5), a cluster of four quad core nodes is used (Table 9.2). The bandwidth between the resources is 2Gbps. For Case6, a cluster of four homogeneous nodes is used, where all the nodes are the same as R_4 in Table 9.2. Hadoop 0.20 is installed on the cluster, and the Hadoop block size is set to 128MB. Also, the data replication number is set to the default value of three in all algorithms.

Resources	Slot		Mem	
	slot#	*execRate*	*Capacity*	*RetrieveRate*
R_1	4	$100MHz$	$500MB$	$3.2GB/s$
R_2	4	$800MHz$	$16GB$	$3.2GB/s$
R_3	4	$400MHz$	$4GB$	$3.2GB/s$
R_4	4	$3200MHz$	$32GB$	$3.2GB/s$

Table 9.2. Resources in the heterogeneous cluster

The jobs in the experiments are selected from Facebook production Hadoop MapReduce traces, presented in (Chen et al. [2011]). The workload is the same one used in Section 6.7, which is from a cluster at Facebook, spanning six months from May to October 2009. In the Case6 experiments, the workload contains 100 homogeneous jobs, where jobs are set to be *Small Jobs* in the Facebook workload (Table 6.10). Table 9.3 presents the minimum shares assigned to the users in these experiments. It contains the different sets of users defined for the five heterogeneous case studies (Case1 to Case5) in Section 9.2. In all case studies, each user submits jobs from one of the job classes for the Facebook workload in Table 6.10. The minimum share of each user is defined to be proportional to its submitted job size and the available slots in the system.

Users	Case1	Case2	Case3	Case4	Case5
U_1	0	5	0	0	0
U_2	0	3	0	1	0
U_3	0	1	0	3	0
U_4	0	2	0	2	0
U_5	0	1	0	3	0
U_6	0	5	0	0	0
U_7	0	0	0	4	0
U_8	0	0	0	6	0
U_9	0	0	0	8	8
U_{10}	8	6	0	0	0

Table 9.3. User minimum shares

The experiments for Case6 are set to have 10 users with the same submitted jobs. This case study includes a homogeneous Hadoop system with equal job sizes, which is similar to Case3 (Equal Priority) in the defined minimum shares in Table 9.3.

9.3.2 Results

In this section, the results for the case studies are presented. Figure 9.7 presents the dissatisfaction performance metric for the schedulers. As the Fair Sharing algorithm has minimum share satisfaction as its main goal, it is successful in reducing the dissatisfaction rate compared to the other schedulers. The COSHH algorithm also considers the minimum shares as the first critical issue in making scheduling decisions. It assigns the minimum shares while it takes the system heterogeneity into account. This results in competitive dissatisfaction with the Fair Sharing algorithm. However, the FIFO algorithm significantly increases the dissatisfaction rate by ignoring the minimum shares.

Generally, the minimum shares are defined for some users to improve their jobs' completion times. Therefore, to analyze the success rate of schedulers in achieving this goal, the average completion times of the jobs with minimum shares are presented in Figure 9.8. The results show

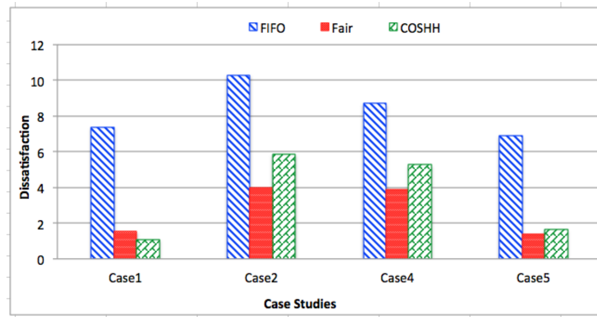


Figure 9.7. Dissatisfaction of schedulers.

that the Fair Sharing algorithm is successful in immediate assignment of minimum shares to the jobs. However, as it does not consider job and resource heterogeneity (Resource and Job Mismatch problem) it increases the average completion time.

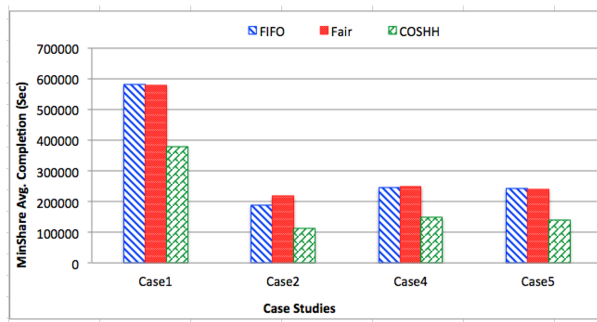


Figure 9.8. Average completion time of jobs with minimum shares.

In Figure 9.9, the average completion times for the three schedulers are presented. The FIFO algorithm achieves better average completion time than the Fair Sharing algorithm in most of the case studies. Both the FIFO and the Fair Sharing algorithm have the Resource and Job Mismatch problem. However, due to the increased Sticky Slots problem in the Fair Sharing algorithm, it repeats the same inefficient decision for a long time. Moreover, the minimum shares assigned by the Fair Sharing algorithm magnify the Sticky Slots problem. Therefore, the average completion time of the Fair Sharing algorithm is larger than the other schedulers. When there are no minimum shares (Case3), the Sticky Slots problem of the Fair Sharing algorithm is reduced. Due to the Small Jobs Starvation problem, the average completion time of small jobs is increased in the case of FIFO.

Figures 9.10 and 9.11 present the fairness and locality of the schedulers, respectively. The Fair Sharing algorithm provides the best fairness, while the COSHH scheduler has competitive locality and fairness. The scheduling overheads in the case studies are presented in Figure 9.12. The Scheduling Complexity problem in the COSHH algorithm leads to higher scheduling time and

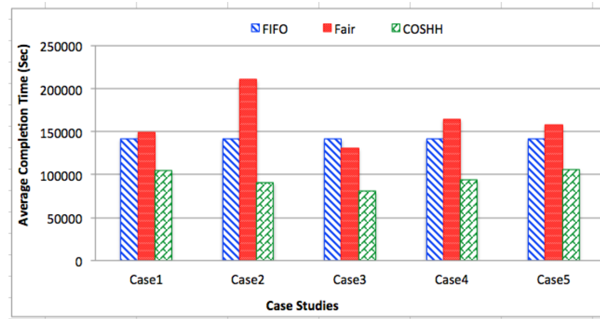


Figure 9.9. Average completion time of schedulers.

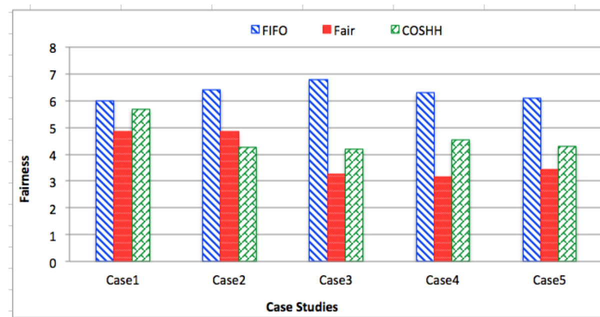


Figure 9.10. Fairness of schedulers.

overhead. However, the total scheduling overhead of COSHH is less than 35 seconds, which is negligible compared to the processing times.

The discussed advantages of the COSHH algorithm are achieved in the heterogeneous Hadoop systems (Case1 to Case5). To provide a broader view, this chapter also evaluates the schedulers in a homogeneous Hadoop system (Case6). For this purpose, experiments are performed on a cluster of homogeneous resources with homogeneous workload.

In this fully homogeneous system, the FIFO algorithm achieves lower overhead and better average completion time. The Scheduling Complexity problem in the COSHH and the Fair Sharing algorithms leads to increases in their average completion times. Based on these results, due to the extra overhead introduced by the COSHH algorithm, it is not a good option for homogeneous Hadoop systems.

9.4 Discussion

The analysis and experimental results in the previous two sections yields the following insights for selecting a scheduler for a Hadoop system:

- Both Fair Sharing and COSHH are sensitive to the changes in the minimum shares. They adapt their scheduling decisions based on any modification in the defined minimum shares.

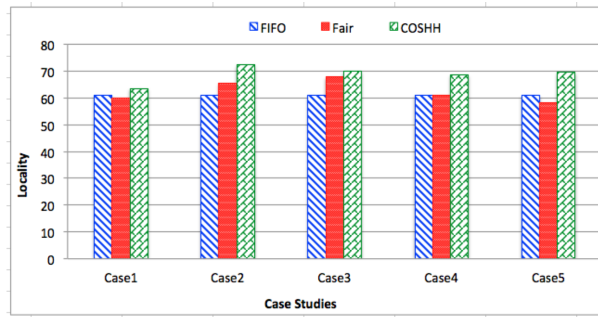


Figure 9.11. Locality of schedulers.

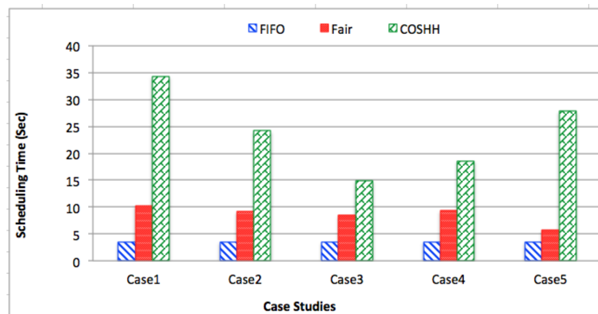


Figure 9.12. Scheduling time of schedulers.

The Fair Sharing algorithm provides better minimum share satisfaction than both the COSHH and the FIFO algorithms. However, as COSHH considers minimum shares in its scheduling decisions, it provides competitive dissatisfaction compared to the Fair Sharing algorithm.

- The sensitivity of some Hadoop schedulers to the minimum shares can significantly degrade their average completion times. For instance, the experiments in this chapter show that the average completion time of the Fair Sharing algorithm can highly fluctuate as the minimum shares change. This problem leads to poor performance of the Fair Sharing algorithm in Case2. On the other hand, COSHH has less fluctuations in its performance. Although the average completion time of COSHH does respond to the minimum shares, its performance is more robust.

- The experimental results confirm the promising performance of the COSHH algorithm in terms of the average completion time for both high priority and the remaining submitted jobs. The results suggest that even though the Fair Sharing algorithm provides immediate shares for the high priority users, it can lead to poor completion times for them in heterogeneous environments.

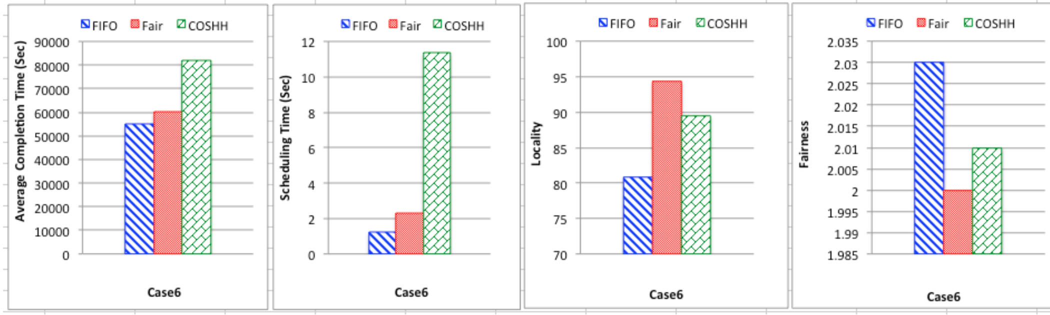


Figure 9.13. Comparison of schedulers in homogeneous Hadoop system (Case6).

- The COSHH algorithm has competitive performance with the Fair Sharing algorithm in terms of the locality and the fairness metrics.

- To find the appropriate scheduler for each Hadoop system, it is first important to consider cluster and workload heterogeneity. If the system is heterogeneous, then the COSHH algorithm can be a better option, even if the assigned minimum shares by the Hadoop provider change during operation.

- The analysis in this research highlights a question about the correlation between a minimum share and the resulting average completion time. As discussed in this chapter, the main intent of defining minimum shares in Hadoop is to provide better performance for some high priority users. Although in a homogeneous Hadoop system, a higher minimum share indeed provides better performance for a user, this may not be the case for heterogeneous Hadoop systems. Based on the experiments in this chapter, in some cases even a higher minimum share of a user may degrade the performance provided for her. This is a critical issue, which suggests further studies as to whether it is still feasible to define minimum shares or whether it is preferable to eliminate the whole minimum share concept in Hadoop (at least for systems that display cluster heterogeneity). This is left as future work.

9.5 Related Work

Minimum shares are defined to improve the performance of jobs submitted by high priority users. The initial idea of defining minimum shares in Hadoop was introduced in Hadoop On Demand (HOD) (Apache [2007]). The HOD approach uses the Torque resource manager for node allocation based on the needs of the virtual cluster. With allocated nodes, the HOD system automatically prepares configuration files, and then initializes the system based on the nodes within the virtual cluster. Once initialized, the HOD virtual cluster can be used in a relatively independent way. Therefore, HOD defines the minimum shares in terms of the actual number

of physical nodes that it allocates to the jobs of each user. This approach guarantees that these users can receive their required shares at each point in time. However, it has serious drawbacks such as removing the shared memory advantages of Hadoop, and can lead to poor utilization of resources.

The Fair Sharing algorithm and its improved version, the Delay Scheduler (Zaharia et al. [2010]) have a similar approach in dealing with the minimum shares. However, the Delay scheduler relaxes the minimum share satisfaction in order to improve the data locality in the system.

The Capacity scheduler (Apache Hadoop Capacity Scheduler [2010]) introduced by Yahoo! (Bodkin [2010]), uses a different approach in considering the guaranteed shares for high priority jobs. It was defined for large clusters, which may have multiple, independent consumers and target applications. For this reason, the Capacity scheduler provides greater control as well as the ability to provide a minimum capacity guarantee, and share excess capacity among users. The queues defined in this scheduler are assigned a guaranteed capacity (where these capacities are similar to the minimum shares defined in other schedulers). Queues are monitored; if a queue is not consuming its allocated capacity, this excess capacity can be temporarily allocated to other queues. The Capacity scheduler provides the ability to prioritize jobs within a queue. Therefore, jobs with a higher priority have access to resources sooner than lower-priority jobs. The Capacity scheduler does not take into account heterogeneity in either jobs or resources, which can lead to poor performance in such settings.

There are a number of Hadoop schedulers developed to improve other performance metrics, while allowing guaranteed shares for users. These include Dynamic Priority (DP) (Sandholm and Lai [2010]) and FLEX (Wolf et al. [2010]). The DP scheduler allows users to bid for map and reduce slots by adjusting their spending over time. The FLEX scheduler extends the Fair Sharing algorithm by proposing a special slot allocation schema that aims to optimize explicitly a given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce phases) that produces the job execution time as a function of the allocated slots.

This chapter evaluates Hadoop schedulers based on the performance levels that they provide for different minimum share settings. Although COSHH has shown promising results for heterogeneous systems with different settings of the minimum shares, its scheduling overhead can be a barrier for small and/or homogeneous systems. DP was developed for user-interactive environments, which are different from our target systems. DP allows dynamically controlled resource allocation. However, it is driven by economic mechanisms rather than a performance model and/or application profiling. Similarly, FLEX relies on a performance function which aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does not provide a technique for considering dif-

ferent system settings or a detailed MapReduce performance model, but instead it uses a set of simplifying assumptions about job execution, task durations, and job progress over time.

9.6 Conclusion

Generally, Hadoop schedulers do not have any control over defining minimum shares. However, the minimum shares can vary over time, which can significantly affect performance. This chapter studies the effects of different minimum share settings on the performance of Hadoop schedulers (FIFO, Fair Sharing, and COSHH). Six case studies are defined based on different settings of minimum shares and heterogeneity. Based on the analysis and experimental results, the COSHH algorithm is a promising solution for heterogeneous Hadoop systems with different minimum share settings. However, in a homogeneous system, the FIFO algorithm is preferred.

Chapter 10

A Prototype System

This section provides an overview of the implementation process of the COSHH scheduler prototype, and the challenges and lessons learned from this process.

10.1 Design Diagram

As introduced in Section 6.3, there are two main message flows in the COSHH scheduler: job arrival flow and heartbeat flow. Figure 10.1 presents the design diagram of COSHH for the job arrival flow. In this diagram, a user submits a job using the Hadoop terminal. The submitted jobs are first received by the components in the Hadoop *core* and *mapreduce* packages. In Figure 10.2, the design diagram of the heartbeat flow is presented. This flow is triggered when there is a free resource in the Hadoop cluster. Three main Hadoop components (presented in blue) are used in the design diagrams of the COSHH scheduler. Some of the main components in these design diagrams are as follows:

- *JobClient*: is a Java class in the *mapreduce* package of Hadoop. It is the primary interface for the user job to interact with the Hadoop cluster. *JobClient* provides facilities to submit jobs, track their progress, access component and tasks' reports/logs, and get the MapReduce cluster status information. The job submission process involves:
  - Checking the input and output specifications of the job.
  - Computing the input data slices for the job.
  - Copying the job's jar and configuration files to the MapReduce system directory on the distributed file system.
  - Submitting the job to the cluster and optionally monitoring its status.

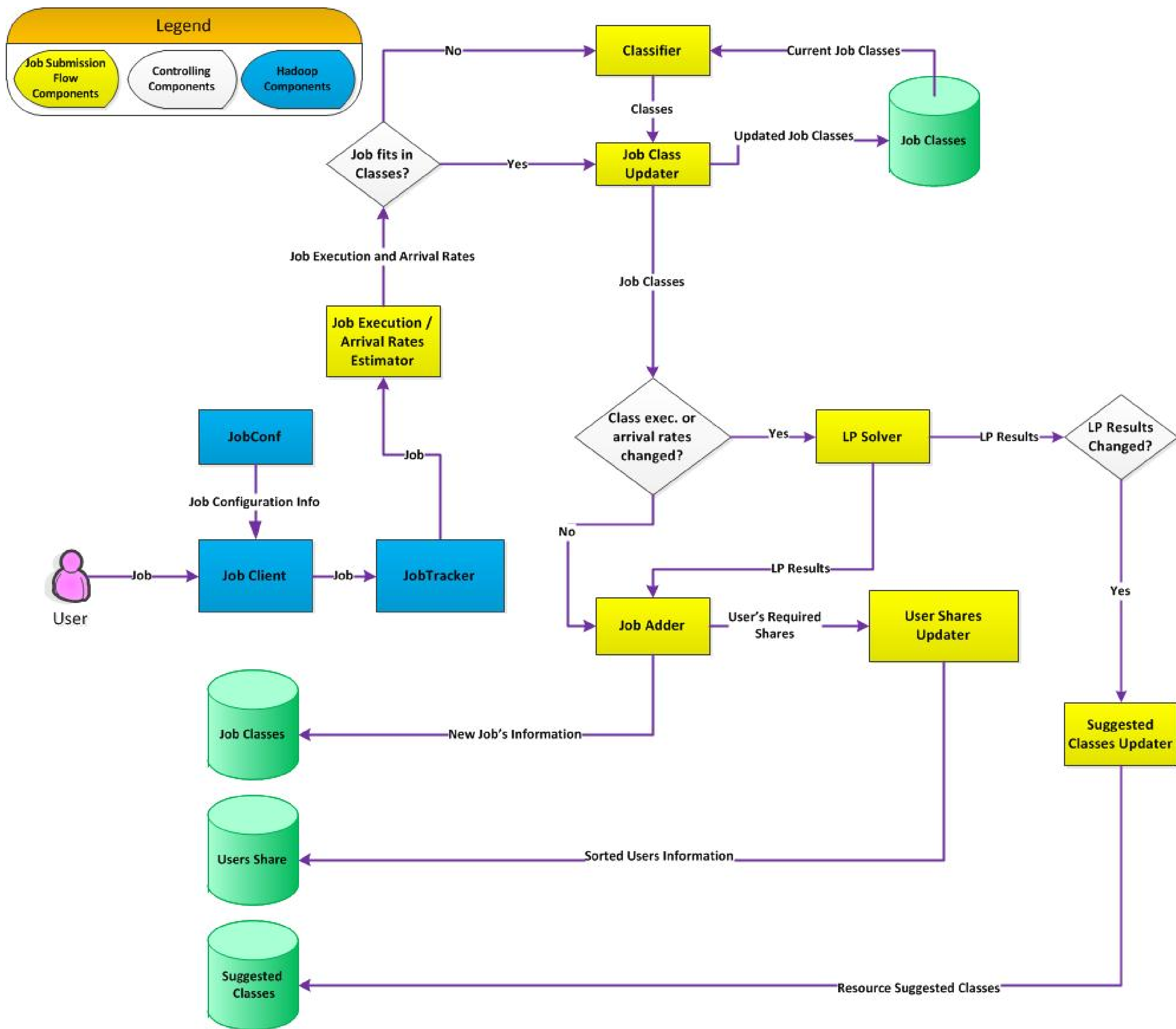


Figure 10.1. Design Diagram - Job Arrival Flow.

Normally a user creates the job, describes its various facets via the *JobConf* class, and then uses the *JobClient* to submit the job and monitor its progress.

- *JobConf*: is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution. The framework executes the job as is described by *JobConf*. However, some configuration parameters might have been marked as final by administrators, and hence cannot be altered. While some job parameters are easy to set (e.g., setting number

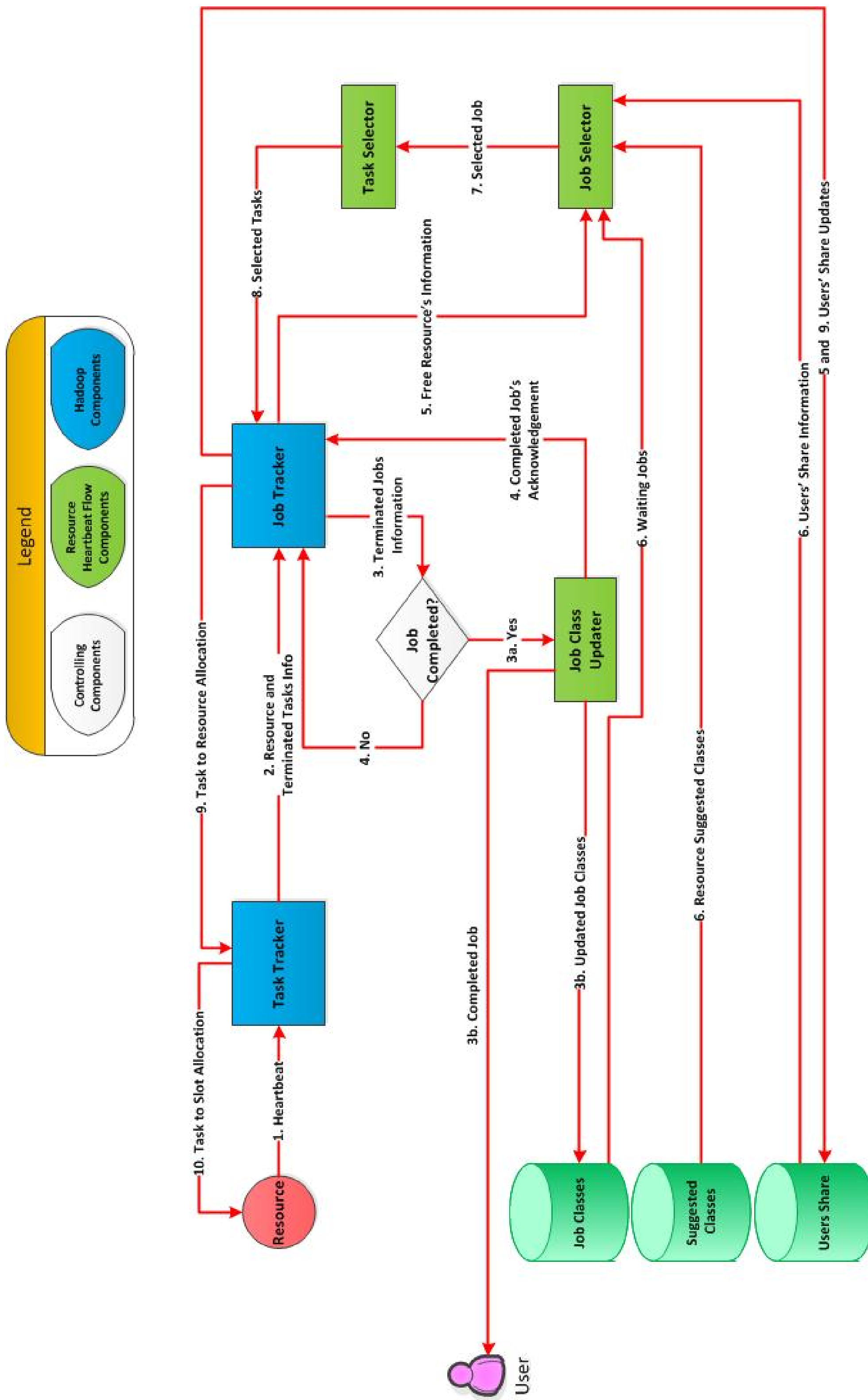


Figure 10.2. Design Diagram- Heartbeat Flow.

of reduce tasks), some parameters (e.g. setting the number of map tasks) interact implicitly with the rest of the framework and job configuration, and are relatively more complex for the user to control.

- *JobTracker*: is the central location for submitting and tracking MapReduce jobs in the network environment of Hadoop. All schedulers in Hadoop, including the COSHH scheduler, inherit from the *TaskScheduler* abstract class. This class provides access to a *TaskTrackerManager*, which is an interface to the *JobTracker* as well as a Configuration instance. It also asks the scheduler to implement three abstract methods: the lifecycle methods start and terminate, and a method called *assignTasks* to launch tasks on a given *TaskTracker*. The task assignment in Hadoop is reactive. *TaskTracker*s periodically send heartbeats to the *JobTracker* with their *TaskTrackerStatus*, which contains a list of running tasks, the number of slots on the resource, and other information. The *JobTracker* then calls *assignTasks* on the scheduler to obtain tasks to launch. These are returned with the heartbeat response.

- *Job Execution and Arrival Rates Estimator*: using the algorithm provided in Section 6.3, these estimates are provided for each incoming job. The estimated values are used to classify the incoming jobs. This component is based on work in the AMP lab at UC Berkeley (Agarwal and Ananthanarayanan [2010]), which is tailored for our scheduler.

- *Classifier*: if the incoming job can not fit in any of the available job classes, this component is triggered to define new job classes. Algorithm 4 presents the pseudocode of the *classifier* component in COSHH.

Algorithm 4 COSHH Scheduler's Classifier Algorithm

for all newly created job class i **do**
  Calculate the mean arrival rates (σ_i) and mean execution rates ($exec_i$) of all jobs in the class ($\sigma_i, exec_i$)
end for
repeat
  Use calculated means, and estimated arrival rates and execution rates of jobs to classify them into classes
  for all class $\in$ JobClasses **do**
    Replace ($\sigma_i, exec_i$) with the means of all jobs' arrival rates and execution rates in each class i
  end for
until there is no change in any mean

- *JobClass Updater*: modifies current job classes. It receives different change requests from other components including an updated list of current JobClasses, modified mean arrival rates and/or mean execution rates of job classes, or an updated list of jobs in the class. The modifications are received either from the *classifier* component, or from assigning a job to a class. The former may change all of the job classes, while the latter only leads to adding

a job to a class. If a job fits in a class it does not modify its mean execution rate or mean arrival rate, and it only updates the job queues.

- *LP Solver*: this component solves the proposed LP in Section 6.3 to find the suggested resources for each job class. It is triggered when the *JobClass Updater* changes the JobClasses rates or adds a new class to the JobClasses list. As discussed in Chapter 6, the IBM ILOG CPLEX Optimizer is used for solving the LP in the COSHH scheduler. The full IBM ILOG CPLEX Optimization Studio consists of the CPLEX Optimizer for mathematical programming, the IBM ILOG CPLEX CP Optimizer for constraint programming, the Optimization Programming Language (OPL), and a tightly integrated IDE. The CPLEX Optimizer has three forms to consider different ranges of users' needs: 1) The CPLEX Interactive Optimizer, 2) Concert Technology, and 3) The CPLEX Callable Library. The COSHH scheduler uses the Concert Technology of CPLEX, to embed the *LP solver* in the code for the scheduler. As introduced in Section 6.3, the LP to be solved is:

$$\max \gamma$$

$$s.t. \sum_{j=1}^{M} \psi_{i,j} \times \theta_{i,j} \geq \gamma \times \sigma_i, \ for\ all\ i = 1, \ldots, B, \tag{10.1}$$

$$\sum_{i=1}^{B} \theta_{i,j} \leq 1, \ for\ all\ j = 1, \ldots, M, \tag{10.2}$$

$$\theta_{i,j} \geq 0, \ for\ all\ i = 1, \ldots, B, \ and\ j = 1, \ldots, M. \tag{10.3}$$

CPLEX is used in the *LP Solver* component of COSHH, based on the following steps:

1. **Create the model.** IloCplex object functionality is used to create our optimization model to be solved by CPLEX. The interface functions for doing so are defined by the ILOG Concert Technology interface IloModeler and its extension IloMPModeler. These interfaces define the constructor functions for modeling objects of the following types, which can be used with IloCplex:
   - IloNumVar: modeling variables
   - IloRange: ranged constraints of the type lb <= expr <= ub
   - IloObjective: optimization objective
   - IloNumExpr: expression using variables

Modeling variables are represented by objects implementing the IloNumVar interface. The continuous variables of our LP are:

$$\gamma,\ \psi_{i,j},\ \theta_{i,j},\ \text{where } i = 1, \ldots, B, \text{ and } j = 1, \ldots, M.$$

These variables are modelled to be used for building expressions (of type IloLinearNumExpr in our LP). The defined expressions are used to create constraints or an objective function for a model.

2. **Solve the Model.** The following method is called to solve the defined LP model:

$$IloCplex.solve()$$

The returned value indicates whether ILOG CPLEX could find an optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been determined at this point. Even more detailed information about the termination of the solver call is available through the method IloCplex.getCplexStatus.

3. **Query the Results.** After the solve method succeeds in finding a solution, its objective value is queried using the following statement:

$$double \quad objval = cplex.getObjValue();$$

Similarly, solution values for all the variables in the array x can be queried by calling:

$$double[\,] \quad xval = cplex.getValues(x);$$

The solve method returns a Boolean value reporting whether (true) or not (false) a solution (not necessarily the optimal one) has been found. The most important solution information computed by IloCplex is usually the solution vector and the objective function value. The method IloCplex.getValue queries the solution vector and IloCplex.getObjValue queries the value of the objective function. Most optimizers also compute additional solution information (for example, dual values, reduced costs, simplex bases, none of which are required in our application). The results are extracted for each resource and are stored to be used in the Job Selector component.

- *Job Adder*: this component adds the incoming job to the waiting list of jobs in its defined class. It sets the information of the incoming job, and prepares it to be selected for execution in the heartbeat flow.

- *User Shares Updater*: once a job is added to the waiting list, the required share of its corresponding user is updated.

- *Suggested Classes Updater*: based on the LP results, the list of suggested classes for each resource is updated and stored to be used in the heartbeat flow.

- *TaskTracker*: receives the heartbeat messages from its corresponding resource, and sends out the number of free slots on the resource, and their information to the *JobTracker*. Moreover, the heartbeat message may include task termination information. The COSHH scheduler first checks the completion of a job to update the users shares, and the job classes.

- *JobSelector*: the *JobTracker* calls the *JobSelector* to select a job for the available resource. The *JobSelector* is implemented based on the algorithm introduced in Section 6.4.

10.2 Implementation Challenges

Hadoop is a large, complicated system. Modifying its internal components along with implementing an entirely new scheduling system for it led to different challenges. Some of the main challenges are listed as follows:

- Overhead Challenge. The biggest implementation challenge of COSHH was reducing the overhead of the LP solving and classification phases. If implemented naively, (i.e. solving the LP and performing classification for all job arrivals) then the completion time of COSHH would be inferior to that of the other Hadoop schedulers, especially for small systems. The implementation aims to make COSHH lightweight so that even for light workloads (small number of jobs), it would not add considerable additional overhead and its overhead would be negligible compared to the processing times involved. The potential overhead of our COSHH scheduler is due to two factors: (1) performing numerous classification and LP solving phases, and (2) using complicated LP solvers and classifiers. The analysis provided in this thesis shows that the number of times the *classifier* and *LP solver* are executed is typically reduced over time, and the classes become more stable as they are calculated for a large number of jobs. Moreover, using a fast *classifier* and *LP solver* helped reduce the overhead in the scheduling process.

- Lack of performance monitoring for developers of Hadoop schedulers. As the Hadoop package was developed for users to receive the computational results of their submitted jobs, it is not well organized for Hadoop scheduler developers. The Hadoop package provides some analysis of the submitted jobs and Hadoop cluster, which is generally useful for users and

administrators of the Hadoop system. However, to compare the proposed Hadoop schedulers, more performance metrics were required to be implemented and defined. For this purpose, the implemented COSHH scheduler modified some of the main Hadoop classes to extend them for measuring more performance metrics.

- Compatibility. Different versions of the Hadoop system have compatibility issues with some hardware or software, which should be considered in implementing a new scheduler. For instance, the cluster used in this research is an IBM blade server, with Linux nodes, which can only support non-SUN JREs. However, the versions of Hadoop released after Hadoop0.20.3 are not compatible with this version of JDK. The compatibility issue leads to some implementation limitations in using Hadoop on different clusters.

- Lack of benchmarks for Hadoop users. Hadoop is generally used in industry. As a result, companies are conservative about releasing their Hadoop workloads, and information related to them. Most of these companies do not release their workloads at all, and those who have done so have made them available in a limited manner. One neglected issue is the lack of comprehensive user benchmarks for Hadoop. User features such as priorities and minimum shares can highly affect the performance of developed Hadoop schedulers. However, to the best of our knowledge there is no accepted benchmark for evaluating Hadoop schedulers. This research handles these limitations by testing schedulers under similar simulated workloads, as well as two real workloads that have been released.

10.3 Lessons Learned

The following are some of the lessons learned while developing the COSHH scheduler and extending the Hadoop system:

- While Big Data is growing rapidly, the applications and systems which can benefit from the analyzed results of Big Data are increasing as well. Hadoop has been developed as a tool for analyzing Big Data for clients; however, as this PhD research demonstrated, Hadoop itself can be a client for Big Data analysis, in the following manner. The COSHH scheduler uses information about previous jobs to calculate estimates for future incoming jobs. These estimates are obtained from analyzing log and history files from previous job submissions. Big Data analysis can be applied to improve the performance of the COSSH scheduler. For instance, the generated job logs can be further used to improve the estimates for future incoming jobs. These more accurate estimates could improve the performance of the COSHH scheduler, and the Hadoop system.

- Hadoop APIs specify complex infrastructure and backend systems which are not easy to use or modify. Moreover, there are various difficulties for programming in Hadoop, such as the lack of easy debugging tools and graphical user interfaces. The researcher has to scroll through lines of logs to figure out a minor problem. Therefore, it is suggested to debug and evaluate the scheduler first on a simulator, and then implement it in a real Hadoop system. The other motivation toward using a simulator as the first evaluation tool is the limitations of real clusters in terms of available resources and cost. Using a simulator, the size of a cluster can be easily increased or decreased, different levels of heterogeneity can be evaluated, and the behaviour of schedulers can be evaluated for different system loads. For example, in our experiments, we could evaluate the scheduler on a real Hadoop cluster, but accessing a large number of resources was not possible without using simulation.

  A good practice employed in this research is evaluating the simulation results by a developed scheduler on a real Hadoop system. There are several factors which are not easily tested on a simulator such as: the computation and memory cost that the schedulers may add to *JobTracker*'s resource, the set up complexity of schedulers, and the effect of actual network delay on schedulers' performance.

- Hadoop uses parallel processing approaches to reduce the computation time. However, the Hadoop resource management process is an integrated and central process. The two main components of Hadoop, *JobTracker* and *NameNode*, are the central points for allocating the computation and storage. The Hadoop scheduling process is designed such that the *JobTracker* component is the central place which makes the final scheduling decisions. This gives rise to a single point of failure and potential network congestion caused by the central scheduling approach. This is one of the drawbacks of Hadoop which should be considered in the next generations of architectures developed for Big Data processing.

10.4 Installing the COSHH scheduler

The COSHH scheduler is implemented in the *contrib* package of Hadoop, and is plugged into the Hadoop system. The following are the steps to set up the COSHH scheduler:

1. Download, and copy the COSHH.jar file in the lib folder in Hadoop. The source code can be found in: *http://www.cas.mcmaster.ca/∼rasooa/*.

2. Download CPLEX, set it up on your system. The installation guide is available in: *http://www-01.ibm.com/support/docview.wss?uid=swg21437813*.

3. Modify HADOOP_CLASSPATH to include the COSHH and CPLEX jar files.

4. Set the following property in the Hadoop config file HADOOP_CONF/mapred-site.xml:

$$< name > mapred.JobTracker.TaskScheduler < /name >$$

$$< value > org.apache.hadoop.mapred.COSHH < /value >$$

5. Configure the XML file, located in HADOOP_CONF/COSHH.xml, which includes minimum shares of users, class numbers, running job limits and preemption timeouts. This file can be modified without restarting the Hadoop cluster as it is reloaded periodically at runtime.

Once the Hadoop cluster is restarted the system starts to use the COSHH scheduler. The scheduler's process can be monitored at http://<JobTracker URL>/scheduler on the *JobTracker*'s web user interface.

Chapter 11

Discussion and Future Work

This thesis presents solutions for improving performance in two critical distributed computing environments: Grid and Cloud. First, a scheduler is introduced for Computational Grid systems, addressing computational requirements of jobs and resources. Then, including data requirements, a scheduler is proposed for Data Grid systems. The proposed schedulers in both Grid systems aim to reduce the required state information, while improving the performance. However, the major focus of this thesis is on proposing scheduling solutions for the widely used distributed computing environment, Hadoop, which includes both computing and data challenges. The proposed Hadoop schedulers are analyzed from the viewpoints of scalability, sensitivity to the estimated parameters, heterogeneity, and minimum share sensitivity.

The proposed scheduler for Computational Grid, called the Grid Shadow Routing algorithm, defines virtual queues to reduce the required state information (Chapter 3). Its performance is evaluated using the GridSim simulator, where the results show its promising performance for aggregate measures such as flowtime. This scheduler is recommended for Grid environments where the system elements are not tightly coupled, and the communication cost is considerable.

The scheduler introduced for Data Grids (called DATALPAS) reduces the required state information and search space for scheduling decisions (Chapter 4). Its main objective is to improve average completion times by considering the system heterogeneity. The proposed scheduler consists of two parts for addressing the data and computational issues of Data Grids. The scheduler is evaluated using real Grid workloads and the Data GridSim simulator. The results show the promising performance of the DATALPAS scheduler in terms of metrics such as flowtime and data availability.

The major part of this research concentrates on providing scheduling solutions for Hadoop systems (Chapter 6). Heterogeneity is for the most part neglected in designing Hadoop schedulers. Growing interest in applying the MapReduce programming model in various applications gives rise

to greater heterogeneity, and thus must be considered in its impact on performance. The COSHH scheduler is introduced for Hadoop, consisting of a classifier and an optimizer. The proposed scheduler classifies the jobs based on their requirements and finds an appropriate matching of resources and jobs. The COSHH scheduler is evaluated using various artificial and real Hadoop workloads in terms of different performance metrics. The COSHH scheduler improves the average completion time, and has promising performance in terms of fairness, minimum share satisfaction, and locality. Moreover, the experiments show that compared to the improvement in average completion time, the additional overhead of the COSHH scheduler is in most cases negligible.

Chapter 7 studies three key Hadoop factors, and the effect of heterogeneity in these factors on the performance of Hadoop schedulers. Performance issues for Hadoop schedulers are analyzed and evaluated in different heterogeneous and homogeneous settings. Five case studies are defined based on different levels of heterogeneity in the three Hadoop factors. Based on these observations, guidelines are suggested for choosing a Hadoop scheduler according to the level of heterogeneity in each of the factors considered.

There is a considerable challenge in Hadoop systems to schedule the growing number of tasks and resources in a scalable manner (Chapter 8). Moreover, the potential heterogeneous nature of deployed Hadoop systems tends to increase this challenge. A hybrid scheduler is introduced for scalable and heterogeneous Hadoop systems. This research analyzes the performance of widely used Hadoop schedulers including FIFO and Fair Sharing and compares them with the COSHH scheduler. Performance issues for Hadoop schedulers are analyzed and evaluated for heterogeneous and scalable Hadoop systems. These results suggested a combination of the FIFO, Fair Sharing, and COSHH schedulers can be effective, where the selection is based on the load on the system and available system resources.

Generally, Hadoop schedulers do not have any control on defining minimum shares (Chapter 9). However, the minimum shares can vary over time, which can significantly affect performance. This thesis studied the effects of different minimum share settings on the performance of Hadoop schedulers (FIFO, Fair Sharing, and COSHH). Six case studies are defined based on different settings of minimum shares and heterogeneity. Based on the analysis and experimental results, the COSHH scheduler is a promising solution in a heterogeneous Hadoop system with different minimum share amounts. However, in a homogeneous system, the FIFO algorithm is preferred.

11.1 Applications

The proposed schedulers and guidelines in this thesis have various applications. The following list provides several of them.

1. In a Computational Grid system with widely and geographically distributed resources, gath-

ering state information can be very costly (in terms of time and network traffic). Moreover, analyzing the collected information to make a scheduling decision can add considerable complexity and overhead. The Grid Shadow Routing algorithm can be a promising candidate for such Grid systems as it provides good average completion time performance while requiring minimal state information. Furthermore, as it considers system heterogeneity, this scheduler is suggested for Computational Grids with heterogeneous jobs and resources. A good example of such a system is in the Enabling Grids for E-sciencE (EGEE) project (Erwin and Jones [2009]), where a Grid platform is provided as a service to the broad e-science community.

2. The Data Grid systems used for the CERN project utilize huge data sets on large numbers of geographically distributed resources (Gagliardi et al. [2002]). In these large heterogeneous Data Grid systems, gathering full state information and performing searches over large search spaces requires significant time. Moreover, the data in these systems is analyzed by different scientists all around the world, where the submitted jobs generally have similar requirements (Gagliardi et al. [2002]). The DATALPAS scheduler can be a good candidate for such Data Grid systems. This scheduler can simultaneously improve both the average completion time and the communication time.

3. Due to significant advantages of Hadoop, different companies have built their own Hadoop clusters using their current heterogeneous resources (Zhang et al. [2010]). Generally, these heterogeneous Hadoop clusters are highly loaded. The COSHH scheduler can significantly increase the benefits by reducing the average completion times and satisfying minimum share requirements. Moreover, our sensitivity analysis shows that this scheduler can provide desired performance levels even in a system where accurate estimates of job lengths are not available.

4. Selecting an appropriate Hadoop scheduler should be based on detailed information of the system. If the scheduler is not appropriately selected for the corresponding Hadoop system, it can lead to significant performance degradation. The guidelines provided in Chapter 7 should be used when a company wants to build its Hadoop cluster. Whether the resources are in-house or obtained from Cloud providers, these guidelines should help set up an appropriate scheduler for the system.

5. Reducing the resource rental cost can greatly increase the benefits for small companies building their own Hadoop clusters using Cloud resources. These systems can result in significant savings by adjusting the number of resources based on the system load. The

scalability in these Hadoop systems can be provided by using the proposed hybrid scheduler in Chapter 8.

6. The minimum share satisfaction analysis provided in Chapter 9 can be used by Hadoop administrators. They can select an appropriate scheduler based on the minimum share settings of the corresponding Hadoop system.

11.2 Challenges

The main challenges of the proposed schedulers are as follows:

1. The Grid Shadow Routing algorithm requires estimates of two parameters: task length and resource execution rate. If in a Computational Grid system these parameters have very large and frequent fluctuations, providing the estimates of new parameters can increase the overhead. Therefore, this scheduler may not be a good candidate for such systems. It should be noted that based on the sensitivity analysis in Chapter 3, the Grid Shadow Routing algorithm can tolerate up to around 40% fluctuations in estimation errors for these parameters.

2. The proposed DATALPAS scheduler is designed to reduce the completion time of highly loaded systems. Therefore, if the Grid system is generally under loaded, using this scheduler can increase the overhead without adding considerable improvements. Moreover, in a Data Grid system with very large and frequently varying job requirements, the performance of the DATALPAS scheduler may degrade.

3. The significant improvements of the COSHH scheduler are provided with the cost of increasing the complexity and overhead of scheduling system. Therefore, as discussed in Chapter 7, this scheduler may perform poorly in homogeneous or underloaded Hadoop systems. Moreover, if in a Hadoop system the parameters are frequently and highly varying, the overhead of the COSHH scheduler can increase. However, based on the sensitivity analysis in Chapter 6, small variations do not degrade the scheduler's performance.

4. The proposed guidelines for selecting schedulers are defined based on heterogeneity levels of the system to improve the average completion time. Although it improves the current situation, where there are no specific guidelines provided for selecting a Hadoop scheduler, it needs to be extended to consider other factors such as combinations of performance metrics.

5. The proposed hybrid scheduler is designed for heterogeneous Hadoop systems. Therefore it does not target homogeneous Hadoop systems. In a dynamic case where a system varies

between homogeneous and heterogeneous status (due to adding different types of resources), a solution can be provided by combining the hybrid scheduler with the results of Chapter 7.

11.3 Future Work

The following items present potential future extensions to the research presented in this thesis, that could be of benefit to future developments in Grid computing and Hadoop systems.

1. *Grid Shadow Routing algorithm (Chapter 3):* refining the virtual queues and their comparison process to minimize the data transfer and storage costs. The modified algorithm could find appropriate matchings of resources and tasks while it reduces both the costs of data transfer and storage, in addition to what it currently considers.

2. *DATALPAS scheduler (Chapter 4):* extending the low bandwidth replication method to define thresholds and time intervals for other Data Grid systems. Moreover, further Data Grid models and workload models can be considered in an extended version of this scheduler. The scheduler can also include more system constraints such as memory limitations, and economic objectives for users and resources.

3. *COSHH scheduler (Chapter 6):* further analysis of this scheduler on large scale Hadoop clusters to determine its power in increasing the locality. Moreover, its performance can be improved by using other light weight classification and optimization methods such as GLPK (GNU Linear Programming Kit) (Makhorin [2012]). The scheduler can also be extended to provide separate classifications for data-intensive and computation-intensive jobs leading to more precise matchings of resources and jobs.

4. *Guidelines for selecting schedulers (Chapter 7):* evaluation in larger systems by scaling up the number of jobs, resources, and users. The required threshold specifying small versus large jobs can be further investigated. The outcome will be a selection function that considers system parameters including type, number, and complexity of jobs as well as specification of available resources. Moreover, other performance metrics can be considered in the guidelines. The end result will be a guideline suggesting an appropriate scheduler based on desired performance levels for different metrics.

5. *Hybrid scheduler (Chapter 8):* extending the scheduler to also consider homogeneous environments. The future hybrid scheduler will be smart enough to recognize the degree of heterogeneity in a system and then select the best scheduler for a heterogeneous or homogeneous environment. The hybrid scheduler has the potential to also consider other perfor-

mance metrics. The scheduler can be extended to receive a desired performance metric as an input, and select an appropriate algorithm with respect to the corresponding metric.

6. *Minimum share suggestions (Chapter 9):* further analysis in this direction may question the minimum share setting process in Hadoop. As can be seen in this thesis, in some cases defining minimum shares for users may even degrade performance for high priority users. Further analysis in various real Hadoop systems can propose a general critical conclusion as to whether or not it is recommended to define minimum shares. Moreover, this work could be extended in a way that the schedulers can provide suggestions for setting the minimum shares. Therefore, a Hadoop provider could factor these suggestions into their decision making process.

11.4 Relevant Publications and Submissions by the Author

- **Refereed Journal Articles**

  1. A. Rasooli and D. G. Down, *COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop systems.* **Journal of Future Generation Computer Systems (FGCS)**, 2012, (Submitted).
  2. A. Rasooli and D. G. Down, *Guidelines for Selecting Hadoop Schedulers based on System Heterogeneity.* **Journal of Grid Computing**, 2012, (Submitted).

- **Referred Conference and Workshop Proceedings**

  1. A. Rasooli and D. G. Down, *A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems*, **In proceeding of the 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS), Co-located with Supercomputing 2012**, Salt Lake City, USA, November 12th, 2012.
  2. A. Rasooli and D. G. Down, *An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems*, **In Proceeding of the 21st Annual International Conference hosted by the Centre for Advanced Studies Research, IBM Canada (CASCON 2011)**, November 7-10, 2011, Toronto, Canada.
  3. A. Rasooli and D. G. Down, *State Independent Resource Management for Distributed Grids*, **In Proceeding of the 6th International Conference on Software and Data Technologies (ICSOFT 2011)**, July 18-21, 2011, Seville, Spain.

- **Posters**

  1. A. Rasooli and D. G. Down, *Heterogeneous Hadoop Scheduling Algorithms*, **Poster presented in IBM CASCON 2011Technology Showcase**, November 1- 4, 2011.
  2. A. Rasooli and D. G. Down, *Improving Overhead of Scheduling in Computational and Data Distributed Computing Systems*, **Poster presented in IBM CASCON 2010 Technology Showcase**, November 1 4, 2010.

Bibliography

Ashraf Aboulnaga, Ziyu Wang, and Zi Y. Zhang. Packing the Most onto Your Cloud. In *Proceedings of the First International Workshop on Cloud Data Management*, pages 25–28, 2009. ISBN 978-1-60558-802-5. doi: 10.1145/1651263.1651268.

Ajith Abraham, Rajkumar Buyya, and Baikunth Nath. Nature's Heuristics for Scheduling Jobs on Computational Grids. In *Proceedings of the 8th IEEE International Conference on Advanced Computing and Communications (ADCOM)*, pages 45–52, Tata McGraw -Hill, India, 2000.

David Abramson, Jon Giddy, and Lew Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 520, Los Alamitos, CA, USA, 2000. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.846030.

Sameer Agarwal and Ganesh Ananthanarayanan. Think Global, Act Local: Analyzing the Trade-Off between Queue Delays and Locality in MapReduce Jobs. Technical report, EECS Department, University of California, Berkeley, 2010.

Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big Data and Cloud Computing: Current State and Future Opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 530–533. ACM, 2011. ISBN 978-1-4503-0528-0. doi: 10.1145/1951365.1951432.

Sayaka Akioka and Yoichi Muraoka. Extended Forecast of CPU and Network Load on Computational Grid. In *Proceedings of the 4th IEEE International Symposium on Cluster Computing and the Grid(CCGrid'04)*, pages 765–772, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7803-8430-X.

Issam Al-Azzoni and Douglas G. Down. Dynamic Scheduling for Heterogeneous Desktop Grids. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (GRID '08)*, pages 136–143, Washington, DC, USA, 2008a. IEEE Computer Society. ISBN 978-1-4244-2578-5. doi: 10.1109/GRID.2008.4662792.

Issam Al-Azzoni and Douglas G. Down. Linear Programming-Based Affinity Scheduling of Independent Tasks on Heterogeneous Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1671–1682, 2008b. doi: 10.1109/TPDS.2008.59.

Ammar H. Alhusaini, Viktor K. Prasanna, and Cauligi S. Raghavendra. A Unified Resource Scheduling Framework for Heterogeneous Computing Environments. In *Proceedings of the 8th Heterogeneous Computing Workshop*, page 156, Los Alamitos, CA, USA, 1999. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/HCW.1999.765123.

Bill Allcock, Ann Chervenak, Ian Foster, Carl Kesselman, and Miron Livny. Data Grid Tools: Enabling Science on Big Distributed Data. *Journal of Physics: Conference Series*, 16(1):571, 2005. URL `http://stacks.iop.org/1742-6596/16/i=1/a=079`.

Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman. Basic Local Alignment Search Tool (BLAST). *Journal of Molecular Biology*, 215:403–410, 1990. doi: doi:10.1016/S0022-2836(05)80360-2.

Apache. Hadoop On Demand Documentation, 2007. URL `http://hadoop.apache.org/common/docs/r0.17.2/hod.html`. [Online; accessed 30-November-2010].

Apache Hadoop Capacity Scheduler, 2010. URL `http://hadoop.apache.org/docs/r0.20.2/capacity_scheduler.html`. [Online; accessed 30-November-2011].

Apache Hadoop Fair Scheduler, 2010. URL `http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html`. [Online; accessed April-2010].

Apache Hadoop Foundation. Hadoop Wiki. `http://wiki.apache.org/hadoop/PoweredBy`, June 2012.

Apache Hadoop Foundation. The GridMix Hadoop Benchmark. `http://hadoop.apache.org/docs/stable/gridmix.html`, May 2010a.

Apache Hadoop Foundation. Hadoop. `http://hadoop.apache.org/hadoop`, May 2010b.

Apache Hadoop Foundation. The Hbase Project. `http://hadoop.apache.org/hbase`, May 2010c.

Apache Hadoop Foundation. HDFS Architecture Guide. `http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html`, May 2010d.

Apache Hadoop Foundation. The Hive Project. `http://hadoop.apache.org/hive`, May 2010e.

Apache Hadoop Foundation. The Pig Project. `http://hadoop.apache.org/pig`, May 2010f.

Apache Hadoop Foundation. The Zookeeper Project. http://hadoop.apache.org/zookeeper, May 2010g.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

Robert Armstrong, Debra Hensgen, and Taylor Kidd. The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-Time Predictions. In *Proceedings of the 7th Heterogeneous Computing Workshop*, page 79, Los Alamitos, CA, USA, 1998. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/HCW.1998.666547.

Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and Grid Technologies for Wide-Area Distributed Computing. *SoftwarePractice & Experience*, 32(15):1437–1466, 2002. ISSN 0038-0644. doi: 10.1002/spe.488.

Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson Correlation Coefficient. *Springer Topics in Signal Processing*, 2:1–4, 2009. doi: 10.1007/978-3-642-00296-0_5.

Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003. ISSN 1045-9219. doi: 10.1109/TPDS.2003.1195409.

Ron Bodkin. Yahoo! Updates from Hadoop Summit 2010. http://www.infoq.com/news/2010/07/yahoo-hadoop-summit, July 2010.

Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L Bölöni, Muthucumara Maheswaran, and Albert I Reuther. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001. ISSN 0743-7315. doi: 10.1006/jpdc.2000.1714.

Rajkumar Buyya. The World-Wide Grid, June 2001. URL http://www.buyya.com/ecogrid/wwg/.

Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience(CCPE)*, 14(13–15):1175–1220, 2002. doi: 10.1002/cpe.710.

Rajkumar Buyya, Kim Branson, Jonathan Giddy, and David Abramson. The Virtual Laboratory: A Toolset to Enable Distributed Molecular Modelling for Drug Design on the World-Wide Grid. *Concurrency and Computation: Practice and Experience*, 15(1):1–25, 2003. doi: 10.1002/cpe.704.

Rajkumar Buyya, Manzur Murshed, David Abramson, and Srikumar Venugopal. Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimization Algorithm. *Journal of Software: Practice and Experience (SPE)*, 35(5): 491–512, 2005. ISSN 0038-0644. doi: 10.1002/spe.646.

David G. Cameron, Rubén Carvajal-Schiaffino, Paul Millar, Caitriana Nicholson, Kurt Stockinger, and Floriano Zini. Evaluating Scheduling and Replica Optimisation Strategies in OptorSim. In *Proceedings of the 4th International Workshop on Grid Computing (GRID '03)*, page 52, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2026-X.

Kelvin Cardona, Jimmy Secretan, Michael Georgiopoulos, and Georgios Anagnostopoulos. A Grid Based System for Data Mining Using MapReduce. Technical Report TR-2007-02, AMALTHEA, 2007.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816.

Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of the 19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 390–399, Washington, DC, USA, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/MASCOTS.2011.12.

Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the International Conference on Very Large Data Bases (VLDB) Endowment*, 5(12):1802–1813, 2012. URL `http://dl.acm.org/citation.cfm?id=2367502.2367519`.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM - 50th Anniversary Issue: 1958 - 2008*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

Alexandre Denis, Christian Pérez, and Thierry Priol. Towards High Performance CORBA and MPI Middlewares for Grid Computing. In *Proceedings of the 2nd International Workshop on Grid Computing*, GRID '01, pages 14–25. Springer-Verlag, 2001. ISBN 3-540-42949-2. URL `http://dl.acm.org/citation.cfm?id=645441.652842`.

Frederic Desprez and Antoine Vernois. Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid. *Journal of Grid Computing*, 4(1):19–31, March 2006. ISSN 1572-9184. doi: 10.1007/s10723-005-9016-2.

Fangpeng Dong. *Workflow Scheduling Algorithms in the Grid*. PhD thesis, Queen's University, Kingston, Ontario, Canada, April 2009.

Fangpeng Dong and Selim G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. Technical Report 504, School of Computing, Queens University, Kingston, Ontario, Canada, 2006.

Fangpeng Dong and Selim G. Akl. A Joint Data and Computation Scheduling Algorithm for the Grid. In *Proceedings of the 13th International Euro-Par Conference*, Lecture Notes in Computer Science, pages 587–597. Springer, August 2007. ISBN 978-3-540-74465-8. doi: 10.1007/978-3-540-74466-5_62.

Laure Erwin and Bob Jones. Enabling Grids for e-Science: The EGEE Project, EGEE-PUB-2009-001, 2009. URL `http://www.eu-egee.org/`.

Alpaydin Ethem. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

Stefka Fidanova and Mariya K. Durchova. Ant Algorithm for Grid Scheduling Problem. In *Proceedings of the 5th International Conference on Large-Scale Scientific Computing (LSSC)*, pages 405–412, Sozopol , Bulgaria, 2005. doi: 10.1007/11666806_46.

Fabrizio Gagliardi, Bob Jones, Mario Reale, and Stephen Burke. European DataGrid Project: Experiences of Deploying a Large Scale Testbed for e-Science Applications. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 480–500, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44252-9. URL `http://dl.acm.org/citation.cfm?id=647414.760331`.

Archana Ganapathi, Yanpei Chen, Armando Fox, Randy H. Katz, and David A. Patterson. Statistics-Driven Workload Modeling for the Cloud. In *Proceedings of the 26th In-*

ternational Conference on Data Engineering (ICDE), pages 87–92. IEEE, March 2010. doi: 10.1109/ICDEW.2010.5452742.

Dennis Gannon, Peter Beckman, and Elizabeth Johnson. HPC++. `http://www.extreme.indiana.edu/ sage`, December 2009.

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 24–37. USENIX Association, 2011. URL `http://dl.acm.org/citation.cfm?id=1972457.1972490`.

David Goddeau. Profile Driven Scheduling for a Heterogeneous Server Cluster. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops(ICPPW '05)*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2381-1. doi: 10.1109/ICPPW.2005.73.

Derek Gottfrid. The New York Times Archives + Amazon Web Services = TimesMachine. `http://open.blogs.nytimes.com/2008/05/21/the-new-york-times-archives-amazon-web-services-timesmachine/`, May 2013.

Derek Gottfrid. Self-Service, Prorated Super Computing Fun. `http://tinyurl.com/2pjh5n`, March 2009.

Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah K. Alham, and Zelong Liu. MRSim: A Discrete Event based MapReduce Simulator. In *Proceedings of the 7th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2010)*, pages 2993–2997, Yantai, Shandong, China, 2010. IEEE.

Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269. ACM, 2008. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454152. URL `http://doi.acm.org/10.1145/1454115.1454152`.

Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22. USENIX Association, 2011. URL `http://dl.acm.org/citation.cfm?id=1972457.1972488`.

Wolfgang Hoschek, Francisco Javier Jaén-Martínez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID '00)*, pages 77–90, London, UK, 2000. Springer-Verlag. ISBN 3-540-41403-7. doi: 10.1.1.16.3897.

IBM ILOG CPLEX Optimizer, 2010. URL `http://www-01.ibm.com/software/integration/optimization/cplex/`. [Online; accessed 30-November-2010].

Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, and Catalin Dumitrescu. The Grid Workloads Archive, Nov 2006. URL `http://gwa.st.ewi.tudelft.nl/`.

Alexandru Iosup, Ozan Sonmez, Shanny Anoep, and Dick Epema. The Performance of Bags of Tasks in Large-Scale Distributed Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 97–108, 2008.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72. ACM, 2007. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005.

Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629601.

William E. Johnston, Dennis Gannon, and Bill Nitzberg. Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 34, Los Alamitos, CA, USA, 1999. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/HPDC.1999.805298.

Jorge Jovicich, Carey Priebe, Michael M. Miller, Randy Buckner, and Bruce Rosen. Biomedical Informatics Research Network: Integrating Multi-Site Neuro Imaging Data Acquisition, Data sharing and Brain Morphometric Processing. In *Proceedings of the 18th IEEE Symposium on Computer-Based Medical Systems (CBMS '05)*, pages 288–293, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2355-2. doi: 10.1109/CBMS.2005.38.

George Karypis, Eui-Hong (Sam) Han, and Vipin Kumar. Chameleon: Hierarchical Clustering Using Dynamic Modeling. *Journal of Computer*, 32:68–75, 1999. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.781637.

Kamal Kc and Kemafor Anyanwu. Scheduling Hadoop Jobs to Meet Deadlines. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 388–392, Washington, DC, USA, 2010. ISBN 978-0-7695-4302-4. doi: 10.1109/CloudCom.2010.97.

Houda Lamehamedi, Boleslaw Szymanski, Zujun Shentu, and Ewa Deelman. Data Replication Strategies in Grid Environments. In *Proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing*, page 378, Los Alamitos, CA, USA, 2002. IEEE Computer Society. ISBN 0-7695-1512-6. doi: http://doi.ieeecomputersociety.org/10.1109/ICAPP.2002.1173605.

Ralf Lämmel. Google's MapReduce Programming Model – Revisited. *Science of Computer Programming*, 70(1):1–30, January 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.07.001.

Keqin Li. Job Scheduling and Processor Allocation for Grid Computing on Meta Computers. *Journal of Parallel and Distributed Computing*, 65(11):1406–1418, 2005. ISSN 0743-7315. doi: 10.1016/j.jpdc.2005.05.015.

Michael Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstation. In *Proceeding of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Springer-Verlag, June 1988. ISBN 0-8186-0865-X. doi: 10.1109/DCS.1988.12507.

Miron Livny and Rajesh Raman. *The Grid: Blueprint for A New Computing Infrastructure*, chapter High-Throughput Resource Management, pages 311–337. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-475-8.

Dong Lu, Huanyuan Sheng, and Peter Dinda. Size-Based Scheduling Policies with Inaccurate Scheduling Information. In *Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 31–38, Los Alamitos, CA, USA, 2004. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/MASCOT.2004.1348179.

Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-Based Enterprise Grid Computing System. In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, Las Vegas, USA, June 2005.

Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. *High-Performance Computing: Paradigm and Infrastructure*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Hoboken, NJ, USA, October 2006. ISBN 978-0-471-65471-1.

Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW' 99)*, page 30, Los Alamitos, CA, USA, 1999. IEEE Computer Society. ISBN 0-7695-0107-9. doi: http://doi.ieeecomputersociety.org/10.1109/HCW.1999.765094.

Carolyn Mair, Gada F. Kadoda, Martin Lefley, Keith Phalp, Chris Schofield, Martin J. Shepperd, and Steve Webster. An Investigation of Machine Learning based Prediction Systems. *Journal of Systems and Software*, 53(1):23–29, 2000.

Andrew Makhorin. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk`, 2012.

Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-13360-1.

Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceeding of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518. ACM, 2010. doi: 10.1145/1807167.1807223.

Benjamin Moseley, Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. On Scheduling in MapReduce and Flow-Shops. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 289–298. ACM, 2011. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989540.

Arun C. Murthy. Hadoop. Next Generation MapReduce Scheduler. `http://goo.gl/GACMM`, March 2011.

Sang-Min Park, Jai-Hoon Kim, Young-Bae Ko, and Won-Sik Yoon. Dynamic Data Grid Replication Strategy Based on Internet Hierarchy. In *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing (GCC '03)*, volume 3033 of *Lecture Notes in Computer Science*, pages 838–846. Springer, 2004. ISBN 3-540-21993-5. doi: 10.1007/b97163.

Linh T.X. Phan, Zhuoyao Zhang, Boon T. Loo, and Insup Lee. Real-Time MapReduce Scheduling. Technical Report MS-CIS-10-32, Department of Computer and Information Science, University of Pennsylvania, January 2010. URL `http://repository.upenn.edu/cis_reports/942/`.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005. URL `http://dblp.uni-trier.de/db/journals/sp/sp13.html#PikeDGQ05`.

Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-Driven Task Co-Scheduling for MapReduce Environments. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*, pages 373 –380, april 2010. doi: 10.1109/NOMS.2010.5488494.

Lincoln Pratson and Wences Gouveia. Seismic Simulations of Experimental Strata. In *American Association of Petroleum Geologists (AAPG) Bulletin*, pages 129–144, 2002.

An Qin, Dandan Tu, Chengchun Shu, and Chang Gao. XConveryer: Guarantee Hadoop Throughput via Lightweight OS-Level Virtualization. In *Proceedings of the International Conference on Grid and Cloud Computing*, volume 0, pages 299–304. IEEE Computer Society, 2009. ISBN 978-0-7695-3766-5. doi: http://doi.ieeecomputersociety.org/10.1109/GCC.2009.62.

Arcot Rajasekar, Michael Wan, Reagan Moore, and Wayne Schroeder. Data Grid Federation. In Hamid R. Arabnia and Jun Ni, editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '04)*, volume 2, pages 541–546, Las Vegas, Nevada, USA, June 2004. CSREA Press. ISBN 1-892512-24-6.

Kavitha Ranganathan and Ian Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC '02)*, page 352, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1686-6. doi: http://doi.ieeecomputersociety.org/10.1109/HPDC.2002.1029935.

Kavitha Ranganathan and Ian T. Foster. Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids. *Journal of Grid Computing*, 1(1):53–62, 2003. doi: 10.1023/A:1024035627870.

Kavitha Ranganathan and Ian T. Foster. Identifying Dynamic Replication Strategies for a High-Performance Data Grid. In *Proceedings of the 2nd International Workshop on Grid Computing (GRID '01)*, pages 75–86, London, UK, 2001. Springer-Verlag. ISBN 3-540-42949-2.

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-Core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24. IEEE Computer Society, 2007. ISBN 1-4244-0804-0. doi: 10.1109/HPCA.2007.346181. URL `10.1109/HPCA.2007.346181`.

Aysan Rasooli and Douglas G. Down. An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems. In *Proceedings of the 2011 Conference of the Center for Advanced*

Studies on Collaborative Research, CASCON '11, pages 30–44, Toronto, Ontario, Canada, 2011. IBM Corp. URL http://dl.acm.org/citation.cfm?id=2093889.2093893.

Thomas Sandholm and Kevin Lai. MapReduce Optimization Using Regulated Dynamic Prioritization. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 299–310. ACM, 2009. doi: 10.1145/1555349.1555384.

Thomas Sandholm and Kevin Lai. Dynamic Proportional Share Scheduling in Hadoop. In *Proceedings of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 110–131. Heidelberg, 2010.

Hongzhang Shan, Leonid Oliker, and Rupak Biswas. Job Super Scheduler Architecture and Performance in Computational Grid Environments. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, page 44, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 1-58113-695-1.

Alexander L. Stolyar and Tolga Tezcan. Control of Systems with Flexible Multi-Server Pools: A Shadow Routing Approach . Bell Labs Technical Memo, Revised, 2009.

Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. A Toolkit for Modelling and Simulating Data Grids: An Extension to GridSim. *Concurrency and Computation: Practice & Experience*, 20(13):1591–1609, September 2008. ISSN 1532-0626. doi: 10.1002/cpe.v20:13.

Martin Swany and Rich Wolski. Representing Dynamic Performance Information in Grid Environments with the Network Weather Service. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, page 48, Los Alamitos, CA, USA, 2002. IEEE Computer Society. ISBN 0-7695-1582-7. doi: http://doi.ieeecomputersociety.org/10.1109/CCGRID.2002.1017111.

Zhuo Tang, Junqing Zhou, Kenli Li, and Ruixuan Li. MTSD: A Task Scheduling Algorithm for MapReduce Base on Deadline Constraints. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, volume 0, pages 2012–2018, Los Alamitos, CA, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-0974-5. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPSW.2012.250.

The European DataGrid Project, October 2007. URL http://eu-datagrid.web.cern.ch/eu-datagrid.

The Large Hadron Collider, CERN, January 2004. URL http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/.

The NEESGrid System Integration Team, August 2004. URL http://it.nees.org/documentation/pdf/TR_2004_13.pdf.

Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In *Proceedings of the 18th International Conference on Grid and Cooperative Computing*, GCC '09, pages 218–224, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3766-5. doi: 10.1109/GCC.2009.19.

Jason Venner. *Pro Hadoop*. Apress, Berkely, CA, USA, 1st edition, 2009. ISBN 1430219424, 9781430219422.

Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids. In *Proceedings of the 2nd Workshop on Middleware for Grid Computing (MGC '04)*, pages 75–80, New York, NY, USA, 2004. ACM. ISBN 1-58113-950-0. doi: http://doi.acm.org/10.1145/1028493.1028506.

Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys (CSUR)*, 38(1):3, 2006. ISSN 0360-0300. doi: http://doi.acm.org/http://doi.acm.org/10.1145/1132952.1132955.

Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, Karlsruhe, Germany, June 2011. IEEE/ACM.

Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20. Springer-Verlag, 2010. ISBN 978-3-642-16954-0.

Worldwide LCG Computing Grid, November 2012. URL http://wlcg.web.cern.ch.

Lingyun Yang, Jennifer M. Schopf, and Ian Foster. Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 31, Los Alamitos, CA, USA, 2003. IEEE Computer Society. ISBN 1-58113-695-1. doi: http://doi.ieeecomputersociety.org/10.1109/SC.2003.10015.

Matei Zaharia, Dhruba Borthakur, Joydeep S. Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, April 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html`.

Matei Zaharia, Dhruba Borthakur, Joydeep S. Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, pages 265–278, Paris, France, 2010. doi: http://doi.acm.org/10.1145/1755913.1755940.

Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010. doi: 10.1007/s13174-010-0007-6.

Xuehai Zhang, Jeffrey L. Freschl, and Jennifer M. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC '03)*, page 270, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1965-2. doi: http://doi.ieeecomputersociety.org/10.1109/HPDC.2003.1210036.

Yuanyuan Zhang, Wei Sun, and Yasushi Inoguchi. Predicting Running Time of Grid Tasks based on CPU Load Predictions. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (Grid06)*, pages 286–292, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 1-4244-0343-X. doi: http://doi.ieeecomputersociety.org/10.1109/ICGRID.2006.311027.

Fred Zlotnick. *The POSIX.1 Standard: A Programmer's Guide*. Benjamin Cummings, Redwood City, USA, 1991. ISBN 0-8053-9605-5.