2017

Improving the Performance of Data-intensive Computing on Cloud Platforms

Wei Dai
University of Central Florida

IMPROVING THE PERFORMANCE OF DATA-INTENSIVE COMPUTING ON CLOUD PLATFORMS

by

WEI DAI
B.E. Zhejiang University, 1999
M.S. University of Central Florida, 2009

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2017

Major Professor: Mostafa Bassiouni

ABSTRACT

Big Data such as Terabyte and Petabyte datasets are rapidly becoming the new norm for various organizations across a wide range of industries. The widespread data-intensive computing needs have inspired innovations in parallel and distributed computing, which has been the effective way to tackle massive computing workload for decades. One significant example is MapReduce, which is a programming model for expressing distributed computations on huge datasets, and an execution framework for data-intensive computing on commodity clusters as well. Since it was originally proposed by Google, MapReduce has become the most popular technology for data-intensive computing. While Google owns its proprietary implementation of MapReduce, an open source implementation called Hadoop has gained wide adoption in the rest of the world. The combination of Hadoop and Cloud platforms has made data-intensive computing much more accessible and affordable than ever before.

This dissertation addresses the performance issue of data-intensive computing on Cloud platforms from three different aspects: task assignment, replica placement, and straggler identification. Both task assignment and replica placement are subjects closely related to load balancing, which is one of the key issues that can significantly affect the performance of parallel and distributed applications. While task assignment schemes strive to balance data processing load among cluster nodes to achieve minimum job completion time, replica placement policies aim to assign block replicas to cluster nodes according to their processing capabilities to exploit data locality to the maximum extent. Straggler identification is also one of the crucial issues data-intensive computing has to deal with, as the overall performance of parallel and distributed applications is often determined by the node with the lowest performance. The results of extensive

evaluation tests confirm that the schemes/policies proposed in this dissertation can improve the performance of data-intensive applications running on Cloud platforms.

ACKNOWLEDGMENTS

I have received assistance from many people, both directly and indirectly, in completing my doctoral study. I would like to take this opportunity to express my gratitude to all of them.

In particular, I owe a debt of gratitude to my advisor Dr. Mostafa Bassiouni for his understanding, patience, encouragement, assistance and guidance throughout the whole course of my doctoral study, without which I could not have possibly finished this work. It's been a great honor for me to conduct my doctoral research under his supervision. To him, I am and will always be deeply grateful.

I would also like to thank all my committee members, Dr. Cliff(Changchun) Zou, Dr. Jun Wang, Dr. Mingjie Lin, and Dr. Yuanli Bai for taking the time to serve in my doctoral committee, and for their valuable comments and suggestions regarding my research work as well.

I am also thankful to the university and the department for giving me the opportunity to pursue my PhD here at UCF, and to all the faculty and staff members that have ever provided assistance to me.

Finally, I would like to extend my gratitude to my family. The completion of my doctoral study has been a real challenge, and I couldn't have done it without the constant support of them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

We are in the era of Big Data, which is one of the most important technology trends that are revolutionizing the way enterprises conduct their businesses. Data has been generated in an explosive way that has never been witnessed before. It's estimated that the data accumulated by the World Wide Web during the time period of 2010 through 2014 is more than all of the prior data produced throughout the whole human history. Every day, there is about 3 Exabytes (1 Exabyte = 1 billion Gigabytes) data generated worldwide, and the total volume of data would reach 35 Zettabytes (1 Zettabyte = 1 trillion Gigabytes) by 2020.

1.1 Big Data

Big Data has four measurable characteristics, the so called four V's: Volume, Velocity, Variety and Veracity. The Volume and Velocity of data are the major technology concerns, and are responsible for most of the cost of data management.

- Volume of Data

  Big Data is measured in Petabytes and Exabytes in contrast to traditional data that is measured in Gigabytes and Terabytes. We now live in a digital world where we consistently and continuously generate digital data. A big part of this digital world is the World Wide Web, which covers more than 100 million domains and has well over one trillion web pages. And this huge volume of data is growing every single day with 20,000 new domains, 300 million daily Facebook posts, 250 million daily Tweeter tweets, and so on. The primary reason for the immense volume of data is

the cost of storing data has been decreasing dramatically, which makes it possible to record almost everything that can be digitalized.

- Velocity of Data

  Unlike traditional data, Big Data is a fast, continued and perpetual flow of data generated by billions of devices such as computers, mobile devices, and digitally connected machines and sensors, from various sources including business processes, online transactions, and social media posts. In most cases, the data users don't have any control over the speed at which data comes in. The velocity of data has been increasing rapidly because of the significant and continuous increase in Internet speed.

- Variety of Data

  Big Data is much more diversified than traditional data in terms of the form, function and source of data. The form of data includes number, text, figure, picture, audio, video, website logs, machine data and so on. Composite data consists of multiple forms, for example text documents can have figures and pictures embedded in them. The function of data has an even wider variety, such as transaction records for business operation, conversation data for human communication, performance data for machine monitoring, and songs or videos for recreation entertainment. As to the source of data, Big Data can be generated and communicated by various devices, such as mobile phones, tablet devices, computers, servers, sensors and machines, and by different entities, such as individuals, organizations and governments.

- Veracity of Data

  The quality of data can vary remarkably. The data absorbed by any Big Data system may contain errors, biases, noise and irrelevant information. There are two major reasons for the abnormality in data. First, the source of data may not be authoritative or reliable. Second, the data may not be generated, transmitted or received correctly due to human errors or machine failures. Data can be either structured or unstructured. Big Data is mostly unstructured data that usually contains a lot of uncertain or imprecise information. As a result, data has to be prepared, cleansed and filtered, before it can be analyzed to generate any useful output.

The combination of the above four characteristics in an interactive way poses a big challenge to data analytics. There have been many data management methodologies, solutions, platforms and tools developed to generate, capture, store, organize, analyze and visualize data. Big Data is defined as not only the enormous and ever growing data itself, but also all the technologies designed to deal with it.

1.2 Data-Intensive Computing

Data-Intensive Computing (DIC) is the enterprise counterpart of High Performance Computing (HPC). However, there are many differences between them. HPC mostly solves mathematical equations for static data sets, while DIC performs exploratory search, analysis and processing of huge, dynamic and complicated data sets. HPC employs high precision arithmetic and structured algorithms, while DIC uses variable precision or integer based arithmetic and

iterative refinement and interrogation. HPC computations usually have spatial and temporal locality, while DIC computations have only very limited (if any) locality.

The challenge of data-intensive computing is now faced in many different application domains. Typical data-intensive applications include Internet search, video streaming, electronic commerce, social media, business intelligence, scientific analyses, experiments and simulations. Current data-intensive applications usually process Terabyte or Petabyte data sets that are mostly distributed and of heterogeneous forms. The process typically consists of multiple phases that employ various transformations and combinations of data. The process usually scales near-linearly while the size of the data set increases, and is amenable to straightforward parallelization in most cases. The fundamental issue that most data-intensive applications need to address is to manage and process exponentially growing data volumes streaming in rapidly often from various sources in a timely manner. Major complexity factors of data-intensive applications include the size of data, the complexity of processing, the number of data sources, the heterogeneity of data, the distribution of data, and the required timeliness of processing.

Although the requirements of data-intensive applications vary greatly in characteristics across different application domains, three major solution paradigms have been widely adopted to solve various data-intensive problems.

- Data Processing Pipeline

  The paradigm of data processing pipeline is mostly used for applications in the scientific domain, especially the processing of data from scientific experiments or simulations. Under this paradigm, the processing usually consists of three major phases. The first phase typically remove data noise, and also perform certain

preliminary processing, such as index, summarize, or markup, to facilitate the manipulation of data in the second phase. In the second phase, complex processing algorithms are applied to the output of the first phase, which generate analytical results that can be utilized either by users or further processing. The processing in the second phase usually requires large scale parallel and distributed computing platforms. Finally, the analytical results produced are presented to users in the third phase, which often employs advanced visualization tools. Moreover, users usually can change the value of analytical parameters in this phase and re-execute the previous processing steps in the pipeline to either validate or examine the results.

- Data Warehouse

  The paradigm of data warehouse is commonly used for applications of archival storage. Many commercial enterprises need to store various transaction information for business analytics and informatics purpose. Data is first generated in operational systems, such as marketing or sales, and then goes through data cleansing to ensure data quality, before it is finally stored in the data warehouse. Scientific applications may also need the archival storage functionality to store large volume of data for subsequent processing.

- Data Center

  The paradigm of data center is widely adopted for various data-intensive processing needs. Under this paradigm, data is stored and processed in data centers that are geographically distributed. Nowadays, one single data center can contain hundreds of thousands of cluster nodes constructed from low-cost commodity hardware.

Parallel programming models are necessary to provide abstractions for the simplification of application development. Also, data management frameworks are used to provide run-time support and transparent fault tolerance as well.

1.3 MapReduce

MapReduce [1] is a parallel and distributed programming model and also an associated implementation for processing huge volumes of data on a large cluster of commodity machines. Since it was proposed by Google in 2004, MapReduce has become the most popular technology that makes data-intensive computing possible for ordinary users, especially those that don't have any prior experience with parallel and distributed data processing.

In the programming model of MapReduce, the input of the computation is a set of key/value pairs, and the output is also a set of key/value pairs usually in a different domain from the input. Users define a map function which converts one input key/value pair to an arbitrary number of intermediate key/value pairs, and a reduce function which merges all intermediate values of the same intermediate key into a smaller set of values, typically one value for each intermediate key. An example of the application of the programming model is counting the number of occurrences of each word in a large collection of documents. In this example, the input <key/value> pair to the map function is <the name of certain document in the collection / contents of that document>. The map function emits an intermediate key/value pair of <word/1> for each word in the document. Then the reduce function sums all counts emitted for a particular word to obtain the total number of occurrences of that word.

1.4 Hadoop

While Google owns its proprietary implementation of MapReduce, an open source implementation called Hadoop [2] has gained great popularity in the rest of the world. Hadoop is now being used at many organizations in various industries, including Amazon, Adobe, Facebook, IBM, Powerset/Microsoft, Twitter, and Yahoo! [3]. Many well-known IT companies have been either offering commercial Hadoop-related products or providing support for Hadoop, including Cloudera, IBM, Yahoo!, Google, Oracle, and Dell [4].

The access to computer clusters of sufficient size is necessary for the parallel processing of large volumes of data. However, not every organization with data-intensive computing needs can afford or has the interest to purchase and maintain such computer clusters. The innovative concept of utility computing proposed a perfect solution to this problem, which eliminates both upfront hardware investment and periodical maintenance costs for cloud users. The combination of Hadoop and cloud computing has become an attractive and promising solution to parallel processing of Terabytes and even Petabytes datasets. A well-known feat of running Hadoop in clouds for data-intensive computing was the New York Times used 100 Virtual Machines (VM's) on Amazon Elastic Compute Cloud (EC2) [5] to convert 4 Terabytes of scanned archives from the paper to 11 million articles in PDF format in less than 24 hours.

Hadoop is currently the most mature, accessible, and popular implementation of the MapReduce programming model. A Hadoop cluster adopts the master-slave architecture, where the master node is called the JobTracker, and the multiple slave nodes TaskTrackers. Hadoop is usually supported by the Hadoop Distributed File System (HDFS), an open-source implementation of the Google File System (GFS). HDFS also adopts the master-slave architecture, where the

NameNode (master) maintains the file namespace and directs client applications to the DataNodes (slaves) that actually store the data blocks. HDFS stores separate copies (three copies by default) of each data block for both fault tolerance and performance improvement. In a large Hadoop cluster, each slave node serves as both the TaskTracker and the DataNode, and there would usually be two dedicated master nodes serving as the JobTracker and the NameNode respectively. In the case of small clusters, there may be only one dedicated master node that serves as both the JobTracker and the NameNode.

When launching a MapReduce job, Hadoop first splits the input file into fixed-sized data blocks (64 MB by default) that are then stored in HDFS. The MapReduce job is divided into certain number of map and reduce tasks that can be run on slave nodes in parallel. Each map task processes one data block of the input file, and outputs intermediate key/value pairs generated by the user defined map function. The output of a map task is first written to a memory buffer, and then written to a spill file on local disk when the data in the buffer reaches certain threshold. All the spill files generated by one map task are eventually merged into one single partitioned and sorted intermediate file on the local disk of the map task. Each partition in this intermediate file is to be processed by one different reduce task, and is copied by the reduce task as soon as the partition becomes available. Running in parallel, reduce tasks then apply the user defined reduce function to the intermediate key/value pairs associated with each intermediate key, and generate the final output of the MapReduce job.

In a Hadoop cluster, the JobTracker is the job submission node where a client application submits the MapReduce job to be executed. The JobTracker organizes the whole execution process of the MapReduce job, and coordinates the running of all map and reduce tasks. TaskTrakers are

the worker nodes which actually perform all the map and reduce tasks. Each TaskTraker has a configurable number of task slots for task assignment (two slots for map tasks and two for reduce tasks by default), so that the resources of a TaskTraker node can be fully utilized. The JobTracker is responsible for both job scheduling, i.e. how to schedule concurrent jobs from multiple users, and task assignment, i.e. how to assign tasks to all TaskTrackers. In this chapter, I only address the problem of map task assignment. The map task assignment scheme of Hadoop adopts a heartbeat protocol. Each TaskTraker sends a heartbeat message to the JobTracker every few minutes to inform the latter that it's functioning properly and also whether it has an empty task slot. If a TaskTracker has an empty slot, the acknowledgment message from the JobTracker would contain information on the assignment of a new input data block.

To reduce the overhead of data transfer across network, the JobTracker attempts to enforce data locality when it performs task assignment. When a TaskTraker is available for task assignment, the JobTracker would first attempt to find an unprocessed data block that is located on the local disk of the TaskTracker. If it cannot find a local data block, the JobTracker would then attempt to find a data block that is located on certain node that is on the same rack as the TaskTracker. If it still cannot find a rack-local block, the JobTracker would finally find an unprocessed block that is as close to the TaskTracker as possible based on the topology information on the cluster.

While map tasks have only one stage, reduce tasks consist of three: copy, sort and reduce stages. In the copy stage, reduce tasks copy the intermediate data produced by map tasks. Each reduce task is usually responsible for processing the intermediate data associated with many intermediate keys. Therefore, in the sort stage, reduce tasks need to sort all the intermediate data copied by the intermediate keys. In the reduce stage, reduce tasks apply the user defined reduce

function to the intermediate data associated with each intermediate key, and store the output in final output files. The output files are kept in the HDFS, and each reduce task generates exactly one output file.

Failures are mostly inevitable when Hadoop runs at large scales. Consequently, Hadoop is designed as a fault-tolerant framework that can handle various failures with minimum impact on the quality of service. There are three different failure modes, task failure, TaskTracker failure, and JobTracker failure. When the TaskTracker detects a task failure, it will mark the task attempt as failed, free the task slot on which the task is running, and notify the JobTracker of the failure in its heartbeat message. The JobTracker will then try to reschedule execution of that task on a different TaskTracker. The whole job will fail, if any task fails a configurable number of times (four times by default), which usually means the user code is buggy. TaskTracker failure occurs, when the JobTracker hasn't received any heartbeat message from certain TaskTracker for a configurable period of time (10 minutes by default). TaskTracker failure is a much more serious failure mode than task failure, because the intermediate output of all map tasks that previously ran and finished on the failed TaskTracker becomes inaccessible. In this case, the JobTracker will rerun all those completed map tasks, and reschedule any tasks in progress on other TaskTrackers. JobTracker failure is the most serious failure mode, but it is not likely to happen as the chance that a particular machine fails is low. In the case of JobTracker failure, Hadoop provides a configuration option that can attempt to recover all jobs that were running at the time the failure occurred.

1.5 Contributions

This dissertation includes five contributions that address the performance issue of data-intensive computing on Cloud platforms from three different aspects: task assignment, replica placement, and straggler identification. Most of the research work presented in this dissertation is conducted in the context of Hadoop running on Cloud platforms.

The first contribution presents an improved task assignment scheme based on an optimal minimum makespan algorithm. The scheme projects and compares the completion times of all task slots' next data block, and explicitly strives to shorten the map phase completion time of MapReduce jobs. The results of extensive evaluation tests indicate that, compared with the Hadoop task assignment scheme, the proposed scheme can remarkably reduce the map phase completion time, and it can reduce the amount of remote processing to a much more significant extent, which makes the data processing much less vulnerable to both network congestion and disk contention.

The replica placement policy of Hadoop Distributed File System (HDFS) has a drawback that it cannot generate balanced replica assignment, and hence has to rely on a load balancing utility to balance replica assignment across the cluster nodes at the cost of extra system resources and running time. The second contribution presents an innovative replica placement policy that can assign replicas to nodes in homogeneous clusters as evenly as possible, and also meet all replica placement requirements of HDFS. As a result, there is no need to run any load balancing utility to balance the replica assignment. The third contribution presents an improved replica placement policy that can work in heterogeneous clusters where the nodes on the same rack have the same processing capability. A more advanced and general solution is presented in the fourth contribution, which can work in any homogeneous or heterogeneous environment.

The Standard Deviation (SD) method is a commonly used straggler identification scheme in parallel processing. In spite of its wide adoption, the SD method has certain inherent limitations. The fifth contribution presents an improved straggler identification scheme based on Tukey's method for outlier detection. Tukey's method has two unique features that make it more suitable for straggler identification than the SD method. The results of extensive evaluation tests confirm that the proposed scheme can identify stragglers and, more importantly, start speculative execution earlier than the SD method.

1.6 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter two briefly reviews the literature that is closely related to my research work. Chapter three discusses the issues with the Hadoop task assignment scheme, and presents an improved scheme, the earliest completion time task assignment scheme. Chapter four presents and discusses the evaluation results of the improved scheme. Chapter five presents the partition replica placement policy for Hadoop Distributed File System, which works in homogeneous clusters. Chapter six presents the slot replica placement policy, which is an improved scheme that can work in heterogeneous clusters where nodes on the same rack have the same processing capability. Chapter seven presents improved slot replica placement policy, which is a more advanced and general scheme that can work in any homogeneous or heterogeneous cluster. Chapter eight presents an improved straggler identification scheme for data-intensive computing on cloud platforms. Chapter nine concludes the dissertation and proposes possible future work.

CHAPTER TWO: LITERATURE REVIEW

In this chapter, I briefly review the literature that is closely related to my research work. The reviewed literature is either in the area of data-intensive computing, or on the related mathematical and statistical models.

2.1 Task Assignment

Dean et al. introduced the MapReduce programming model, implementation details, and various refinements in [1]. Their work served as the fundamental basis for the development of Hadoop, as well as all the following research on both MapReduce and Hadoop. Jiang et al. presented a comprehensive performance study of Hadoop on Amazon EC2 in [6]. They identified certain design factors of Hadoop and discussed alternative methods for these factors. Their study indicated that the performance of Hadoop could be remarkably improved by tuning the design factors in a correct way. Lee et al. presented a comprehensive survey on MapReduce in [7]. They discussed the merits and drawbacks of MapReduce, various improvement contributions in literature, and remaining open issues regarding parallel processing with MapReduce. Vijayalakshmi et al. introduced various implementations of MapReduce in [8] including Hadoop. They evaluated and compared the performance of different implementations. Zaharia et al. [9] focused on the speculative execution mechanism of Hadoop to reduce the job completion time. They discussed all the Hadoop assumptions related to speculative execution, and explained why these assumptions broke down in the clouds. They suggested a new strategy of speculative execution, which always speculatively executed the task that was predicted to finish the farthest

into the future. The strategy launched a speculative copy of any potential straggler tasks, before they could actually prolong the completion of the whole MapReduce job at the expense of extra system resources. The insights and experimental results provided in the above papers are very helpful to the in-depth understanding of Hadoop as well as the further improvement on the framework.

The mathematical model related to the map task assignment problem is scheduling identical jobs on uniform parallel machines [10-21]. Pinedo discussed various deterministic scheduling models in the first part of [22], including the model of uniform parallel machines. Lawler et al. [23] pointed out that the problem of scheduling identical jobs on uniform parallel machines could be formulated as linear assignment problem and solved accordingly in polynomial time. Dessouky et al. [24] proposed a priority queue procedure for solving the same problem, which was an optimal and more efficient algorithm. The algorithm served as the basis of my improved task assignment scheme.

2.2 Replica Placement

HDFS replica placement policy has been a popular research subject because of the wide adoption of Hadoop as a general purpose platform for data-intensive computing. Shabeera et al. propose a bandwidth-aware replica placement policy for Hadoop in [25], which measures and compares the bandwidth between the HDFS client and cluster nodes periodically, and places the replica on the node that has the maximum bandwidth to reduce the data transfer time. Zhang et al. address the dynamic block replication issue in [26] from the perspective of replica placement. They propose several constant-factor local search approximation algorithms, and present a dynamic

replica distribution mechanism that implements the algorithms in HDFS. Xie et al. propose a replica placement mechanism for HDFS running on heterogeneous clusters in [27], which distributes replicas to nodes based on their different computing capacities to balance the data processing load across all cluster nodes. Eltabakh et al. present CoHadoop in [28], which is an HDFS implementation of a flexible, dynamic, and light-weight approach for collocating related data files. Compared with the HDFS replica placement policy, CoHadoop can remarkably improve the performance of Hadoop applications that process data from multiple files.

The mathematical model related to my research work on replica placement is a classical NP-hard problem, the Number Partitioning Problem [29-41]. Various solutions to the problem have been proposed in literature. The greedy heuristic is to first sort all the numbers in descending order, and then always assign the largest number remaining to the subset with the smaller sum. Although the greedy heuristic can find the optimal partition in most cases, the set differencing (or KK) heuristic [33] performs much better. At each step, the heuristic replaces the two largest numbers remaining by the absolute value of their difference, which is equivalent to assigning the two numbers to different subsets. This process continues until there is only one number left in the list, which is the discrepancy of the partition. The actual partition can be constructed by working backward through the replacing process. The Complete KK algorithm [34] is extended from the KK heuristic. The KK heuristic always assigns the two largest numbers remaining to different subsets, while the CKK algorithm adds the other option of assigning them to the same subset, which is done by replacing the two largest numbers remaining by their sum. The CKK searches a binary tree where the left branch of each node replaces the two largest numbers by their difference, and the right branch replaces the numbers by their sum.

2.3 Straggler Identification

The statistical model behind straggler identification in parallel processing is outlier detection [42-56]. In statistics, outliers are unusually large or small values in a data set. Informal test is one major category of outlier detection methods, which generates certain criterion interval for outlier detection. The observed values beyond this interval are identified as possible or probable outliers.

The subject of straggler identification and mitigation [57-63] has received considerable amount of research attention due to its importance to the performance of data-intensive computing. The research idea of using task replication to improve job response time is proposed in [64]. The mechanism of speculative execution is used in MapReduce [1] to address the straggler problem, which performs backup execution of the remaining running tasks when the parallel processing is close to completion. Mantri is presented in [65], which is a system that can monitor the performance of processing nodes in MapReduce clusters and remove stragglers based on their causes. Mantri employs three major techniques: restarting tasks running on stragglers, network-aware task placement, and protecting the output of valuable tasks. Maximum Cost Performance (MCP), a new speculative execution strategy, is presented in [66]. MCP uses both progress rate and process bandwidth within a phase to identify stragglers, uses exponentially weighted moving average to predict the processing speed of cluster nodes and calculate the remaining running time of tasks, and performs speculative execution based on a cost-benefit model. The experimental results indicate that MCP can outperform Hadoop 0.21 with respect to both job completion time and cluster throughput. Dolly is presented in [67], which uses task cloning to address the straggler problem for small parallel processing jobs generated by interactive data analyses. Dolly launches

multiple clones of every task, and only uses the output of the clone that completes first. Dolly employs a technique called “delay assignment” to avoid contention for intermediate data, which is the main challenge faced by task cloning. The Longest Approximate Time to End (LATE), an improved task scheduling algorithm for Hadoop, is proposed in [68]. LATE is developed to address Hadoop’s problem of performance degradation in heterogeneous environments by speculatively executing tasks that hurt the job completion time the most based on projected task finish times. The evaluation results show that LATE can improve the job completion time of Hadoop by a factor of two.

CHAPTER THREE: EARLIEST COMPLETION TIME TASK ASSIGNMENT SCHEME FOR HADOOP (PART ONE)

Nowadays, data-intensive problems are so prevalent that numerous organizations in various industries have to face them in their business operation. It is often crucial for enterprises to have the capability of analyzing large volumes of data in an effective and timely manner. MapReduce and its open-source implementation Hadoop dramatically simplified the development of parallel data-intensive computing applications for ordinary users, and the combination of Hadoop and cloud computing made large-scale parallel data-intensive computing much more accessible to all potential users than ever before. Although Hadoop has become the most popular data management framework for parallel data-intensive computing in the clouds, the Hadoop scheduler is not a perfect match for the cloud environments. In this chapter, I discuss the problems of Hadoop task assignment scheme, and present an improved scheme [69], which is based on an optimal algorithm for minimum makespan scheduling and explicitly strives to shorten the duration of the map phase of MapReduce jobs.

3.1 Introduction

We have entered the era of Big Data. It was estimated that the total volume of digital data produced worldwide in 2011 was already around 1.8 zettabytes (one zettabyte equal to one billion Terabytes) compared to 0.18 zettabytes in 2006 [70]. Data has been generating in an explosive way. Back in 2009, Facebook already hosted 2.5 Petabytes of user data growing at about 15 Terabytes per day. And the trading system in the New York Stock Exchange generates around one Terabyte of data every day. For many organizations, Petabyte datasets have already become the

norm, and the capability of data-intensive computing is a necessity instead of a luxury. Data-intensive computing lies in the core of a wide range of applications used across various industries, such as web indexing, data mining, scientific simulations, bioinformatics research, text/image processing, and business intelligence. In addition to large volume, Big Data also features high complexity, which makes the processing of data sets even more challenging. As a result, it is difficult to work with Big Data using most relational database management systems. And the solution is parallel and distributed processing on large number of machines.

Although Hadoop has become the most prevalent data management framework for the processing of large volumes of data in clouds, there exist issues with the Hadoop scheduler that can seriously degrade the performance of Hadoop running in clouds. In this chapter, I discuss the issues with the Hadoop task assignment scheme, and present an improved scheme for heterogeneous computing environments, such as the public clouds. The proposed scheme is based on an optimal minimum makespan algorithm. It projects and compares the completion times of all task slots' next data block, and explicitly strives to shorten the completion time of the map phase of MapReduce jobs. I conducted extensive simulation to evaluate the performance of the proposed scheme compared with the Hadoop scheme in two types of heterogeneous computing environments that are typical on the public cloud platforms. The simulation results showed that the proposed scheme could remarkably reduce the map phase completion time, and it could reduce the amount of remote processing employed to a more significant extent which makes the data processing less vulnerable to both network congestion and disk contention.

The rest of the chapter is organized as follows. Section 2 discusses the issues with the Hadoop task assignment scheme in the context of cloud environments. Section 3 introduces the

related mathematical model on which my new scheme is based. Section 4 provides the details of the new scheme.

3.2 Issues with the Hadoop Task Assignment Scheme

Both MapReduce and Hadoop were originally designed for computer clusters instead of computer clouds. Clusters are mostly a homogeneous computing environment, where homogeneous nodes run in similar load conditions, and tasks of the same type tend to start and finish at roughly close times. However, the situation is completely different in the clouds. Cloud service providers employ virtualization technology to abstract physical resources, simplify their usage, improve hardware utilization, and provide user isolation for security purposes. Although current virtualization technology can isolate CPU and memory usage effectively, co-located VM's still have to compete for both network and disk bandwidth, especially in the case of I/O intensive workload, such as the MapReduce jobs. Even in homogeneous environments, network bandwidth is often a bottleneck, and it is more precious in the clouds due to the employment of virtualization technology. Consequently, a cluster of VM's in the clouds are mostly heterogeneous instead of homogeneous, and there exist two different sources of heterogeneity. First, when the cloud service user only uses small to medium numbers of VM's, most of these VM's would probably reside on distinct physical hosts, and hence would not compete with each other for the I/O resources. However, these VM's still have to compete with varying numbers of VM's belonging to other users running different applications, and hence face resource contention of different intensities that could change during the whole time period of data processing. Secondly, the even worse scenario is when the user runs Hadoop at large scales, and hence large numbers of VM's are allocated. In

this case, most of the VM's belonging to the same user would not be isolated to each other anymore, and would have to compete with each other for I/O resources, which results in much higher heterogeneity in the VM cluster than in the first scenario. As an example, Zaharia et al. [9] conducted large scale experiments on Amazon EC2 to measure its heterogeneity. Their experimental results indicated that the performance difference can be of a factor of about 1.4 in the first scenario, and up to 2.7 in the second scenario due to the heterogeneity of the computing platform. To evaluate the performance of my proposed task assignment scheme, the above scenarios were resembled in the simulation as the slightly and highly heterogeneous environments.

The Hadoop task assignment scheme is simple and intuitive. Whenever a task slot becomes available, the scheme assigns a data block to it. Initially all slots only consume local blocks. After a while, certain slots, most likely the faster ones, would run out of local blocks, and as the data processing approaches the end of the map phase, more and more slots would so. The challenging question of how to utilize these task slots arises. The Hadoop scheme simply assigns a remote block to the slot to prevent it from becoming idle, which may not be appropriate for two reasons. First, task slots can process local blocks much faster than they can remote ones. Although the local slots have to start the processing later than the remote ones that are immediately available, the local slots may still be able to finish earlier than the remote ones. Therefore, it may not be necessary to assign data blocks to remote slots, even if the overhead of fetching remote data blocks is acceptable. Secondly, the utilization of remote slots comes at a price that may be high enough to offset or even outweigh its benefits. This is because reduce tasks start copying intermediate data produced by map tasks as soon as the data becomes available. In most cases, the copy stage accounts for the majority of the execution time of reduce tasks, and requires large amount of data

transfer across network. Therefore, after the first batch of map tasks finish, all the reduce tasks would be running and fetching remote data across network. At this point, the network bandwidth would become the most precious resource within the cluster environment, especially if it is a VM cluster running in a cloud. Any map tasks processing remote blocks would increase the contention for network bandwidth, and slow down the copy stage of all running reduce tasks. Moreover, a remote map task also competes for the local disk bandwidth with all running local tasks, as the remote task needs to read the data block located on the local disk. The competition would slow down both local and remote tasks.

The second issue with the Hadoop scheme is about data locality. One rule works well in the context of data-intensive computing is moving processing to data, which is adopted by both MapReduce and Hadoop. The Hadoop scheduler always attempts to schedule a map task on a node that has a copy of the data block to be processed first. If it could not make it, the scheduler would instead schedule the task as close to a copy of the data block as possible based on the network topology information on the computer cluster. Nodes in a computer cluster are typically connected by high performance network, which means the network topology of the cluster is known and remain unchanged throughout the whole processing period. However, in the case of clouds, the VM's in a cluster are linked together by certain network topology that is completely unknown or at least obscure. And, cloud operators employ the technology of VM migration to balance load among physical servers or bring down certain servers for maintenance purposes, which means the network topology of a cluster of VM's could change during the whole period of data processing. Therefore, when running in the clouds, the Hadoop scheduler would not have sufficient information on the network topology of the VM cluster. As a result, the scheduler may schedule a

map task on a VM to process certain data block located on another VM that could be many network hops away from the first one. Since the data block must be fetched across the network to be processed, that single map task could seriously slow down the data processing if any part of the network between those two VM's is congested.

The completion time of MapReduce jobs is an important performance metric of Hadoop, especially for the use cases of ad-hoc queries where users need to obtain the results as quickly as possible, and for the use cases of public clouds where users are charged according to the amount of time the provisioned resources are used. The Hadoop scheme is reactive instead of proactive, it doesn't make any explicit effort to shorten the Map Phase Completion Time (MPCT). Whereas, the MPCT is crucial to the job completion time of MapReduce jobs, because reduce tasks need to copy intermediate data produced by map tasks before they can start processing, and typically all reduce tasks need the intermediate data produced by each and every map task including the one that finishes the last, which stops at the MPCT.

3.3 Related Mathematical Model

Minimum makespan scheduling is one classical combinatorial optimization problem, where given a set of jobs and a cluster of machines, the scheduling is to assign jobs to machines so that the makespan (maximum completion time of all jobs) is minimized. There exist many variants of this problem, but the one that is specifically related to the map task assignment problem is scheduling identical jobs on uniform parallel machines. The scheduling problem can be defined as follows. A set of identical and independent jobs $\boldsymbol{J_i}$ ($\boldsymbol{i}$ = 1, 2, …, $\boldsymbol{n}$) need to be assigned to a set of uniform machines $\boldsymbol{M_j}$ ($\boldsymbol{j}$ = 1, 2, …, $\boldsymbol{m}$) running in parallel. Machines are uniform if they can

process at most one job at a time, and do so at known processing speeds, which can be either the same or different for different machines. The scheduling objective is to minimize the makespan.

In 1990, Dessouky et al. [24] proposed an algorithm for solving the above scheduling problem, which is based on the Earliest Completion Time (ECT) rule. The algorithm maintains a priority queue of the completion times of the *m* machines' next job assignment. It selects jobs in sequential order and schedules each job on the machine that can complete it the earliest among all machines. And the earliest completion time in the priority queue is replaced by the updated completion time of the machine that is assigned the job. The procedure continues until all *n* jobs are assigned, and returns a series of job completion times of t_1 , t_2 , …, t_n , where $t_1 \leq t_2 \leq \ldots \leq t_n$. It is obvious that no job would be assigned to a machine in such a manner that its completion time can be reduced by the assignment of the job to another machine. Therefore, the Minimality Property can be directly reasoned out from the algorithm procedure, which asserts that there does not exist any other schedule with job completion times $t_1' \leq t_2' \leq \ldots \leq t_n'$, such that $t_k' < t_k$ for any $k = 1, 2, \ldots, n$. In other words, the completion time of each job is the earliest possible time.

In spite of its simplicity, the ECT algorithm is optimal, which can be proved by the Minimality Property. Suppose we have another schedule with job completion times t_1', t_2', …, t_n', which can yield a smaller makespan. Whether the series of t_1', t_2', …, t_n' is in certain order or no order at all, we can always sort it to make it in the ascending order. The last completion time in this sorted series is the makespan of the schedule, which cannot be possibly smaller than t_n (the makespan of the schedule produced by the ECT algorithm) according to the Minimality Property.

3.4 The ECT Task Assignment Scheme

Although the map task assignment problem is related to scheduling identical jobs on uniform parallel machines, the former is not identical to the latter for two reasons. First, each machine in the latter can process all jobs at the same speed. Whereas, in the former, machines can process local blocks faster than they can remote ones. Secondly, each machine in the latter has known and constant processing speed, nevertheless the processing speeds of machines in the former are unknown and fluctuate over the whole processing period. However, the ECT rule proposed by Dessouky et al. is still applicable in the map task assignment problem. Since all data blocks are of the same size, I assume all of them require the same amount of processing work to be processed. Therefore, in the context of map task assignment, the identical jobs are the input data blocks to be processed, and the machines are the task slots on VM's. The scheduler assigns a data block (job) to a task slot (machine) by scheduling a map task on that slot. The whole map phase of a MapReduce job can be considered as a multi-step process. The number of the steps is equal to the number of input data blocks to be processed. In each step, one single data block is processed by certain slot. If we can minimize the completion time of each step, we would be able to minimize the MPCT. Suppose we have a set of task slots S_i ($i \in \{1, 2, \ldots, m\}$) for the processing of input data blocks. Each task slot S_i has an available time T_i at which it would complete its current data block and be available to process its next data block. And if we know that it will take a task slot processing time P_i ($i \in \{1, 2, \ldots, m\}$) to process its next data block, then the completion time of its next data block will be $C_i = T_i + P_i$ ($i \in \{1, 2, \ldots, m\}$). The minimum value $C_j = \min\{C_i\}$ ($i, j \in \{1, 2, \ldots, m\}$) is the earliest possible completion time of the next step of the whole map phase, which means we can minimize the completion time of the next step by assigning a data

block to the task slot S_j ($j \in \{1, 2, \ldots, m\}$). Therefore, the basic task assignment strategy of my new scheme is the ECT rule, i.e. the slot that can complete a data block (either a local or a *remote* one) the earliest among all slots is assigned one. Note that this task slot doesn't have to be the earliest available slot, to which the Hadoop scheme would always assign the next data block. I call the new scheme ECT as it's based on the ECT rule. Figure 3.1 is a flowchart of the ECT scheme. The two concurrent processes: {When a slot becomes available, update its PTE} and {Initialize the priority queue of completion times} are modeled in the flowchart by a concurrent construct similar to the Fork symbol used in UML activity diagrams.

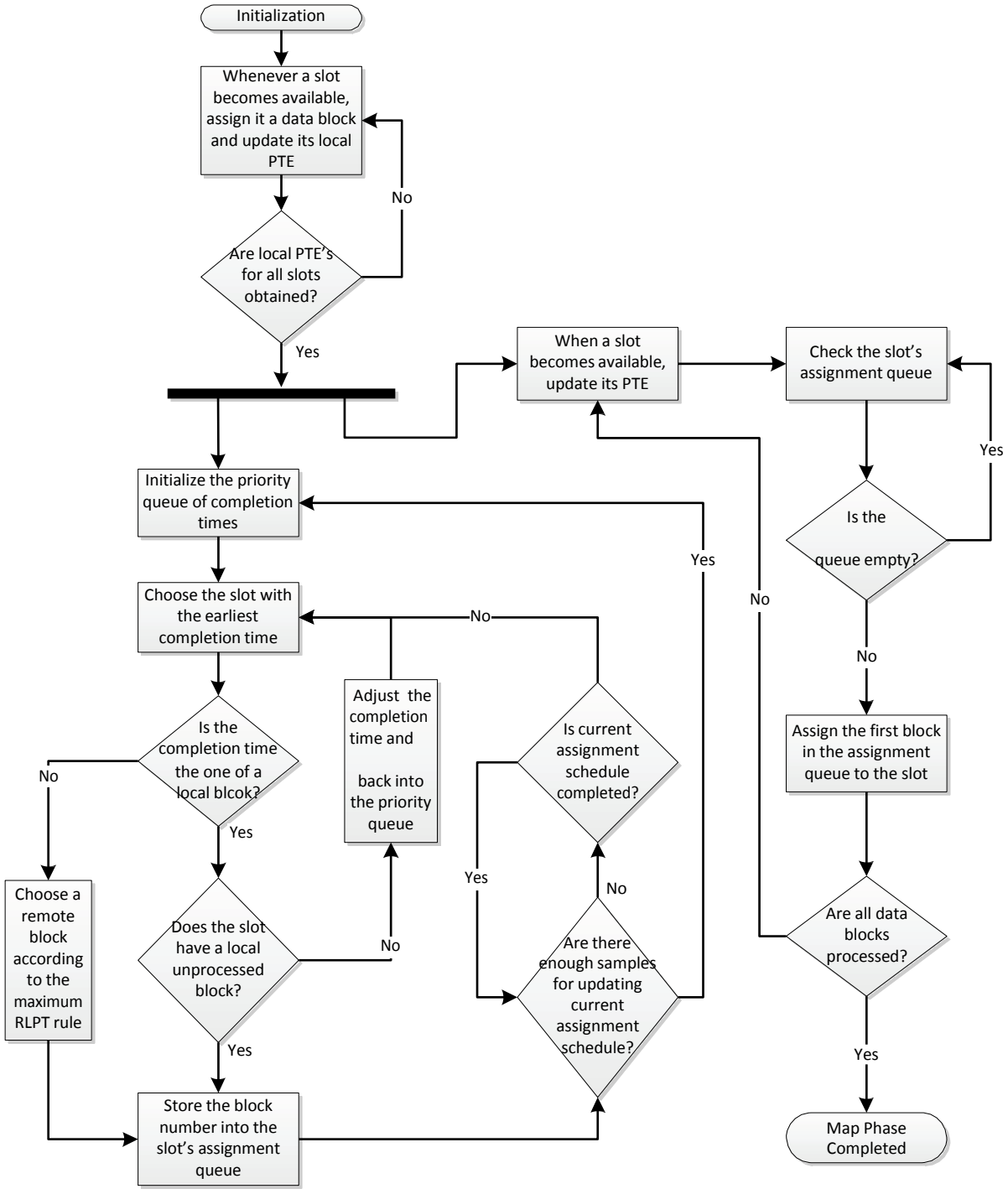

Figure 3.1: Flowchart of ECT

Since the processing speeds of slots fluctuate, ECT predicts the amount of time it takes a slot to process a data block by sampling the processing behaviors of that slot and averaging those samples into a Processing Time Estimate (PTE). For each slot, there are two types of PTE's, local and remote PTE's for the processing of local and remote blocks, respectively. In the simulation, the processing times of a slot were randomly generated within certain range, and hence the PTE was simply computed as the average of all processing time samples. Nevertheless, in practice, the PTE can also be calculated in a way that is similar to the way the round-trip time is estimated in network transport protocols to allow the PTE adapt to sample variance more quickly. Whenever a new sample is obtained, ECT can update the task slot's PTE according to the following formulas:

$$PTE_1 = S_1 \ , \ PTE_{j+1} = \alpha \times PTE_j + (1 - \alpha)\, S_{j+1} \ \ (j = 1, 2, 3, \ldots) \tag{3.1}$$

, where $\boldsymbol{PTE_j}$ is the current PTE, $\boldsymbol{PTE_{j+1}}$ is the new estimate based on the new sample $\boldsymbol{S_{j+1}}$, and $\boldsymbol{\alpha}$ is a constant between 0 and 1 that controls how fast the PTE adapts to sample variance. The value of $\boldsymbol{\alpha}$ can be set to higher ones for more variable computing environments, such as virtualized clouds, and lower ones for more stable computing environments, such as real clusters.

Before it can assign data blocks according to the ECT rule, ECT needs to obtain the local PTE of all task slots. Therefore, ECT assigns data blocks to slots whenever they become available in its first stage. After the local PTE is available for each slot, ECT starts assigning data blocks according to the ECT rule by working out an assignment schedule. ECT maintains a priority queue of the completion times of all slots' next data block based on their PTE's. At each step of the schedule calculation, ECT chooses the slot that can complete its next data block the earliest among

all slots. The next data block can be either local or remote to the slot. Initially, all completion times in the priority queue are the ones of local blocks, as all slots would process local blocks first. At certain point, each slot would run out of its local blocks. The situation that needs special treatment is when there are no remaining local blocks to be assigned to certain slot that has the earliest completion time in the priority queue. Instead of assigning a remote block to the slot immediately, ECT needs to replace the slot's completion time in the priority queue, which is the completion time of its next *local* block, with the completion time of its next *remote* block. Only when this updated completion time shows up as the earliest time in the priority queue, ECT will then assign a remote block to the corresponding slot. And, this remote assignment still minimizes the completion time of the corresponding processing step.

When it does need to assign a data block to a remote slot, ECT chooses a block on the local disk of the slot that has the maximum Remaining Local Processing Time (RLPT) to further reduce the amount of remote processing. The RLPT of a slot is calculated as the number of unprocessed local blocks times the current local PTE of the slot. For the slot having maximum RLPT, the assignment of its local blocks to remote slots is less likely (compared with other slots) to make it run out of local blocks before the map phase ends, and hence it is less likely this slot would engage in remote processing at a later time.

ECT maintains an assignment queue for each slot to store the block numbers of all data blocks assigned to that slot. Task slots don't wait until the whole assignment scheme is calculated to receive their assignments. Instead, as soon as the first data block is assigned to a slot, the slot can start processing the data block. And a slot's data processing would not be interrupted, as long as its assignment queue is not empty. The processing speed of any task slot always fluctuates, and

ECT updates each slot's PTE whenever a new sample becomes available for that slot. ECT also updates the assignment schedule of data blocks to make it better reflect the changing computing environment. After a configurable number of processing time samples are obtained, ECT will work out a new assignment schedule to replace the current one. The update of assignment schedule also helps reduce the effect of estimation error, which is the difference between the PTE and the actual slot processing time. Estimation error will accumulate in the calculation of an assignment schedule, therefore the accuracy of the schedule decreases from the beginning to the end. If the schedule is frequently updated, only the beginning part of it, which is more accurate, will actually be executed.

CHAPTER FOUR: EARLIEST COMPLETION TIME TASK ASSIGNMENT SCHEME FOR HADOOP (PART TWO)

In this chapter, the evaluation results of ECT is presented in section 1, the limitations of ECT is discussed in section 2, and I conclude the research work in section 3.

4.1 Evaluation

I evaluated the performance of ECT compared with the Hadoop scheme by extensive simulation. I used Discrete-Event Simulation to model the operation of the map phase under both the Hadoop scheme and ECT. The discrete sequence of events is the completion of individual data blocks, each of which is completed at a particular instant in time and causes the change of state in the simulated map phase operation. After the state of the map phase operation has been updated, the current simulation time skips to the completion time of the data block that is to be completed next. The simulation procedure continues until all data blocks are processed. Various statistics are recorded during the simulation, and the ones of special interest are the MPCT, and the total Number of data Blocks Remotely Processed (NBRP). The simulation program also records the completion time of each data block for the generation of processing time traces. The common settings of all simulation scenarios are shown in Table 4.1.

Table 4.1: Common settings of all simulation scenarios

Total Processing Workload of Input Data	100,000 task slot × time units
Number of VM's	500
Number of Map Task Slots on Each VM	2 (Hadoop Default)
Duplication Factor of Data Blocks	3 (Hadoop Default)
Speculation	No

I tested ECT in two different computing environments that were typical in the public clouds, the slightly and highly heterogeneous environments. To resemble the heterogeneous environment in the clouds, I assumed the base processing speeds of all 1000 task slots were evenly distributed within a fixed range, while the two slots on the same VM had identical base speed. In the simulation, I actually used slot processing times to represent slot processing speeds. There were two types of slot processing times, Local Processing Time (LPT) and Remote Processing Time (RPT), which indicated the amount of time it took a slot to process a local and remote block, respectively. LPT was randomly generated within the range $[(1 - \boldsymbol{VPPT})\, \boldsymbol{t}, (1 + \boldsymbol{VPPT})\, \boldsymbol{t}]$, where $\boldsymbol{t}$ was the base processing time of the task slot, and VPPT was the Variation Percentage of Processing Time, which was used to reflect the fact that the actual slot processing times fluctuated during the whole processing period. Most MapReduce jobs belong to the relatively short and interactive category, so their job completion times are usually measured in minutes instead of hours. As a result, it is unlikely that the actual slot processing times would fluctuate significantly during the whole processing period. Therefore, I ran the simulation at three different VPPT values: 2.5%, 5% and 10%. The base processing time t's of slots on different VM's were assumed to be evenly distributed within the range $[(1 - \boldsymbol{P})\, \boldsymbol{T}, (1 + \boldsymbol{P})\, \boldsymbol{T}]$, where $\boldsymbol{T}$ = 100,000 / total number of data blocks. In the simulation, the value of P was set to 0.2 and 0.5 to resemble the slightly and highly heterogeneous environments respectively, which was based on the experimental results obtained on Amazon EC2 by Zaharia et al. [9]. The RPT consisted of two parts, the LPT and the amount of time it took the processing slot to fetch the data bock across the network. In the simulation, the RPT was calculated as ($\boldsymbol{RPC} \times \boldsymbol{LPT}$), where RPC was the Remote Processing Coefficient used to reflect the overhead of fetching remote data blocks and hence was greater than one. Since LPT

was randomly generated within the range [(1 – ***VPPT***) ***t***, (1 + ***VPPT***) ***t***], RPT was randomly generated within the range [***RPC*** (1 – ***VPPT***) ***t***, ***RPC*** (1 + ***VPPT***) ***t***].

For both slightly and highly heterogeneous environments, I ran the simulation at four typical combinations of RPC and VPPT values as shown in Table 4.2. Scenario one resembles the computing environment where the remote fetching overhead is low and background load on VM's is fairly stable. Scenario two resembles the environment where the remote fetching overhead is medium and background load is relatively stable. Whereas, in Scenario three, task slots experience network congestion (disk contention), and slot processing speeds fluctuate. Scenario four is to evaluate the performance of ECT in the circumstance where there is no overhead of fetching remote data blocks. For the performance comparison, I examined two metrics: the MPCT and the NBRP. Although minimum MPCT is one of the ultimate goals of all task assignment schemes, the NBRP is also an important metric in the sense that smaller NBRP values mean less remote processing employed, and thus it is less likely the data processing would be delayed by network congestion and/or disk contention. Furthermore, remote processing involves more factors than local processing, thus it is more likely to fail due to its complexity. Therefore, a task assignment scheme that employs less remote processing would be more favorable to one that employs more, if the MPCT's achieved by both schemes are close.

Table 4.2: RPC and VPPT settings in the simulation

	Remote Processing Coefficient (RPC)	**Variation Percentage of Processing Time (VPPT)**
Scenario One	1.5	± 2.5%
Scenario Two	2.5	± 5%
Scenario Three	4.0	± 10%
Scenario Four	1.0	± 2.5%

4.1.1 Slightly Heterogeneous Environment

For the slightly heterogeneous environment, it can be reasoned out from the simulation settings that the shortest possible processing time of one data block is 0.9 x 0.8$\boldsymbol{T}$ = 0.72$\boldsymbol{T}$ time units, and the longest possible processing time 1.1 x 1.2$\boldsymbol{T}$ = 1.32$\boldsymbol{T}$ time units, which is less than two times the shortest time. This means at the time the slowest slot finishes its first block and is assigned its second block, all other slots would be processing their second blocks. Consequently, ECT needs to assign the first 2000 data blocks to obtain the processing time estimates of all task slots, before it can assign the remaining data blocks according to the ECT rule. Therefore, for each of the four scenarios, I ran the simulation with the total number of data blocks taking on values of 2500, 3000, 3500, …, 8000. The amount of processing work of one data block was adjusted according to the total number of data blocks (i.e. $\boldsymbol{T}$ = 100,000 / total number of data blocks), so that the results of different total numbers of data blocks are comparable to each other.

Figures 4.1, 4.2 and 4.3 present the MPCT's achieved by both schemes at different total numbers of data blocks in scenarios one, two and three, respectively. It can be observed that ECT always achieves less MPCT than the Hadoop scheme, and the reduction is most significant in scenario three, where the remote fetching overhead is high. Simulation results of different total numbers of data blocks in scenarios one, two and three are shown in Tables 4.3, 4.4 and 4.5, respectively. (Only partial results are included due to length limitation. All results are the average of ten simulation runs.)

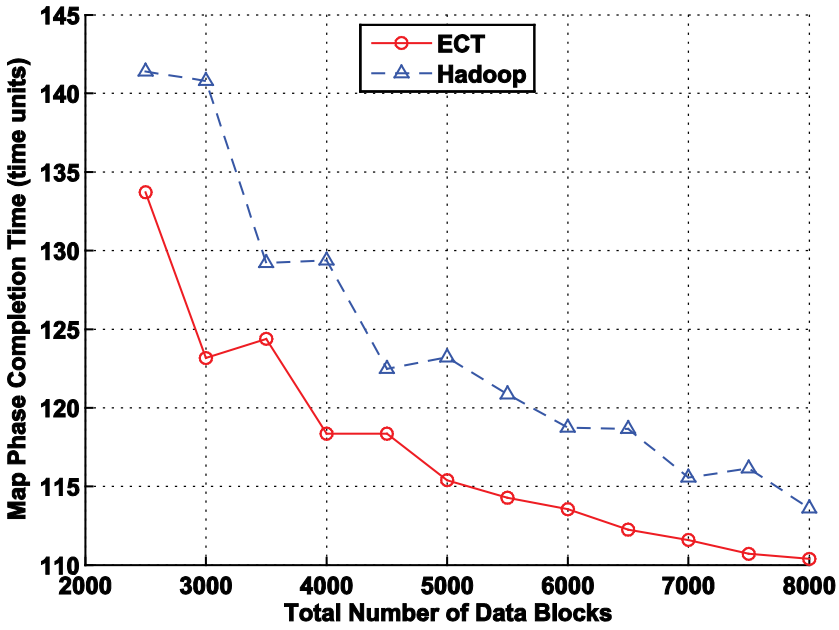

Figure 4.1: Map phase completion times of scenario one (slightly heterogeneous)

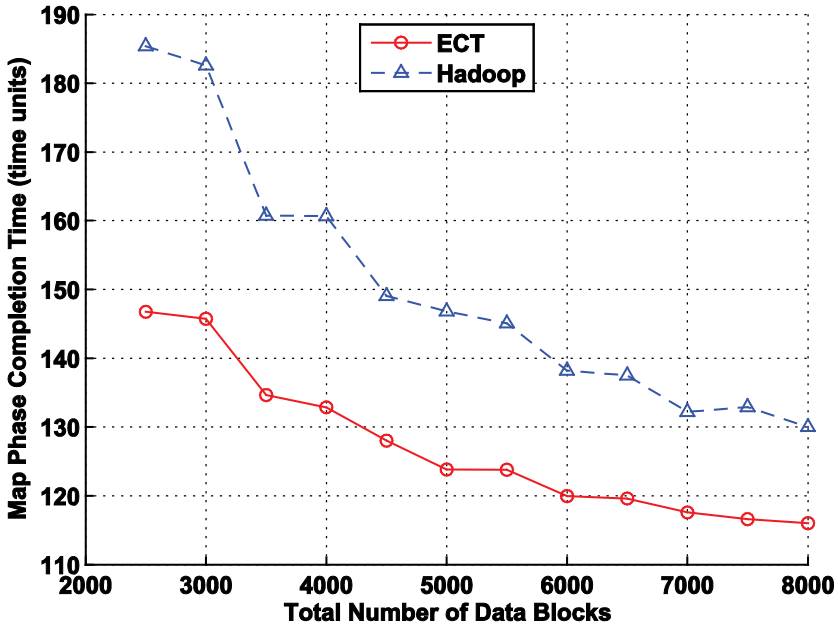

Figure 4.2: Map phase completion times of scenario two (slightly heterogeneous)

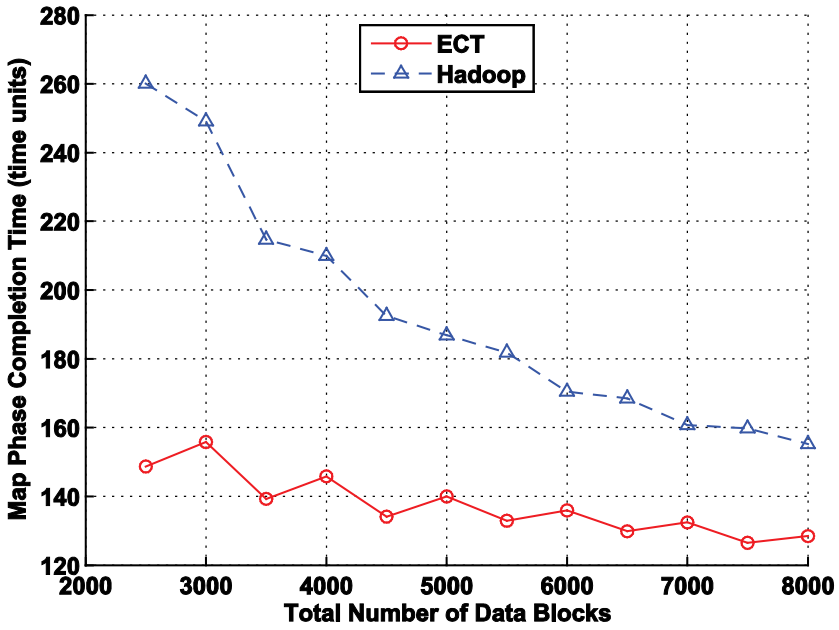

Figure 4.3: Map phase completion times of scenario three (slightly heterogeneous)

Table 4.3: Simulation results of scenario one (slightly heterogeneous environment, low remote fetching overhead, and stable slot processing speeds)

RPC = 1.5 **VPPT = ± 2.5%**	**Total Number of Data Blocks**						
	2500	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**
Number of Blocks Assigned According to the ECT Rule	500	1000	2000	3000	4000	5000	6000
MPCT of Hadoop (time units)	141.4	140.8	129.4	123.2	118.7	115.6	113.6
MPCT of ECT (time units)	133.7	123.2	118.4	115.4	113.5	111.6	110.4
ECT Reduction in MPCT	5.4%	12.5%	8.5%	6.3%	4.4%	3.4%	2.8%
NBRP under Hadoop	297.3	85.7	222.4	315.2	375.5	413.9	449.9
NBRP under ECT	159.4	56.4	143.5	214.7	265.7	300.4	325.6
ECT Reduction in NBRP	46.4%	34.2%	35.5%	31.9%	29.2%	27.4%	27.6%

Table 4.4: Simulation results of scenario two (slightly heterogeneous environment, medium remote fetching overhead, and relatively stable slot processing speeds)

RPC = 2.5 **VPPT = ± 5%**	**Total Number of Data Blocks**						
	2500	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**
Number of Blocks Assigned According to the ECT Rule	500	1000	2000	3000	4000	5000	6000
MPCT of Hadoop (time units)	185.4	182.6	160.7	146.8	138.2	132.2	130.0
MPCT of ECT (time units)	146.8	145.7	132.9	123.8	120.0	117.6	116.0
ECT Reduction in MPCT	20.8%	20.2%	17.3%	15.6%	13.2%	11.1%	10.8%
NBRP under Hadoop	285.2	89.6	214.1	306.9	365.5	411.2	443.1
NBRP under ECT	0.9	15.0	23.6	36.4	85.0	135.6	176.6
ECT Reduction in NBRP	99.7%	83.3%	89.0%	88.1%	76.7%	67.0%	60.1%

Table 4.5: Simulation results of scenario three (slightly heterogeneous environment, high remote fetching overhead, and less stable slot processing speeds)

RPC = 4.0 **VPPT = ± 10%**	**Total Number of Data Blocks**						
	2500	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**
Number of Blocks Assigned According to the ECT Rule	500	1000	2000	3000	4000	5000	6000
MPCT of Hadoop (time units)	260.1	249.1	210.0	186.9	170.5	160.8	155.3
MPCT of ECT (time units)	148.7	155.8	145.9	140.0	135.9	132.5	128.4
ECT Reduction in MPCT	42.8%	37.4%	30.5%	25.1%	20.3%	17.6%	17.3%
NBRP under Hadoop	260.4	95.4	204.8	298.7	358.3	402.1	437.8
NBRP under ECT	0	0	0.1	1.7	6.6	14.6	26.6
ECT Reduction in NBRP	100%	100%	100%	99.4%	98.2%	96.4%	93.9%

The average results of *all* different total numbers of data blocks for each scenario is shown in Table 4.6, which indicate that the average MPCT achieved by ECT is 5.6%, 15.3% and 28.6% less than the one achieved by the Hadoop scheme, and the average NBRP under ECT is 29.5%, 73.8% and 97.5% less than the one under the Hadoop scheme in scenarios one, two and three, respectively. The MPCT reduction of ECT is mostly attributed to its capability to reduce remote processing, which could seriously impair the MPCT performance. As mentioned earlier, ECT is designed based on an optimal algorithm, which assumes that all slot processing speeds remain the same throughout the whole processing period.

Table 4.6: Simulation results of slightly heterogeneous environment

	Scenario			
	One	**Two**	**Three**	**Four**
RPC	1.5	2.5	4.0	1.0
VPPT	± 2.5%	± 5%	± 10%	± 2.5%
Average of Hadoop MPCT (time units)	124.2	150.1	192.5	111.9
Standard Deviation of Hadoop MPCT (time units)	9.3	18.8	34.7	4.9
Average of ECT MPCT (time units)	117.2	127.1	137.5	110.4
Standard Deviation of ECT MPCT (time units)	7.0	10.8	8.8	5.1
ECT Reduction in Average MPCT	5.6%	15.3%	28.6%	1.4%
Average NBRP under Hadoop	321.5	314.4	306.2	—
Standard Deviation of NBRP under Hadoop	99.4	97.4	96.4	—
Average NBRP under ECT	226.6	82.4	7.5	—
Standard Deviation of NBRP under ECT	80.7	60.2	10.9	—
ECT Reduction in Average NBRP	29.5%	73.8%	97.5%	—

If this assumption held, ECT would be able to yield the minimum possible MPCT. Unfortunately, in the case of map task assignment, the slot processing speeds always fluctuate, and hence the optimal solution does not actually exist. However, since the remote fetching overhead is relatively high compared with the fluctuation of the slot processing times, ECT can still effectively reject unnecessary remote processing despite the estimation error of processing time.

It can also be observed from the simulation results that ECT is much more robust to network congestion and/or disk contention than the Hadoop scheme. As shown in Table 4.6, when the RPC increases from 1.5 to 4.0, the average MPCT of Hadoop rises remarkably from 124.2 to 192.5 time units (a 55.0% increase), whereas the average MPCT of ECT only rises from 117.2 to 137.5 time units (a 17.3% increase). The robustness of ECT is a result of its capability to reduce

the NBRP accordingly when the remote fetching overhead rises. As shown in Table 4.6, when the RPC increases from 1.5 to 4.0, the average NBRP under ECT drops sharply from 226.6 to 7.5 (a 96.7% decrease), whereas the average NBRP under Hadoop only decreases slightly from 321.5 to 306.2 (a 4.8% decrease). As mentioned earlier, ECT projects and sorts all task slots' completion times of their next data block and assigns blocks to slots at the sorted order, from the earliest to the latest. When the remote fetching overhead increases, it becomes more and more unlikely that a data block would be assigned to a remote slot. In contrast, the Hadoop scheme doesn't consider either the remote fetching overhead or the MPCT when assigning data blocks. The decrease of the NBRP under Hadoop is actually related to the VPPT instead of the RPC, because when the fluctuation of slot processing times increases, the whole cluster of 1000 task slots becomes slightly less heterogeneous due to the fact that the base slot processing times are evenly distributed within a fixed range. Consequently, the average NBRP under Hadoop decreases insignificantly.

Another important observation from the simulation results is the total number of data blocks has a significant impact on the MPCT's achieved by both schemes. It can be seen from Figures 4.1, 4.2 and 4.3, in general, the MPCT decreases while the total number of data blocks increases for both schemes. The data processing proceeds at the highest speed before it enters the ending stage, because all slots run in parallel to process data blocks. After it enters the ending stage, the processing proceeds slower and slower, as more and more slots stop running. All slots would eventually stop and mostly they stop at different times. In general, the smaller the data blocks, the closer the stop times of different slots. When the total number of data blocks increases, the size of them decreases accordingly, thus overall the stop times of different slots get closer, which has the same effect as increasing the average speed of the data processing in the ending stage and hence

decreases the MPCT. Although they can yield shorter MPCT's, larger values of the total number of data blocks will increase the amount of time it takes to duplicate the data blocks and distribute all the copies to different VM's, and will also increase the maintenance overhead of the Hadoop Distributed File System and the task assignment overhead of the JobTracker in a Hadoop cluster, which necessitates a wise tradeoff between performance and overhead. Moreover, as the data blocks become smaller, the distinction of the MPCT performance between different task assignment schemes also gets smaller, because in general a bad decision on the assignment of a data block would increase the MPCT less than it would when the data block is bigger. Consequently, the MPCT reduction of ECT over the Hadoop scheme decreases when the total number of data blocks increases, as shown in Figures 4.1, 4.2 and 4.3 where the curves of both schemes approach to each other while stretching to the right.

The processing time traces of one typical case are shown in Figure 4.4. It can be seen that the time traces of both schemes match perfectly except for the tail part. This is because both schemes keep all slots running in parallel until the ending stage, when there are not enough remaining data blocks for the schemes to do so. And both schemes have fairly close block completion times as simulation results are the average of ten simulation runs. The two time traces diverge at the tail. While the ECT trace rises in slightly accelerated rate, the Hadoop trace rises sharply. Since there are less and less slots running in the ending stage, the data processing gradually slows down, which causes the accelerated rising of the ECT trace. On the other hand, the Hadoop scheme works reasonably well until most slots run out of local blocks. From this point on until the end of the map phase, the majority of the data blocks would be processed by remote slots, and large amount of remote processing causes the sharp rise of the Hadoop trace.

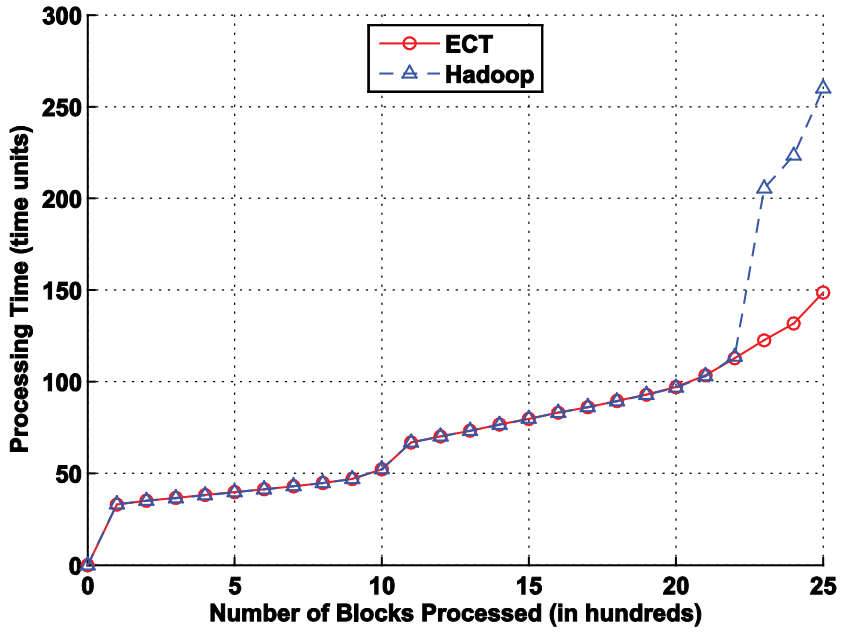

Figure 4.4: Processing time traces of scenario three (slightly heterogeneous environment, Total Number of Data Blocks = 2500)

Table 4.6 also includes the standard deviations of all simulation results in addition to the averages. It can be observed that, in scenarios one, two, and three where there exists remote fetching overhead, the variance of ECT results is always lower than the one of Hadoop results for both MPCT and NBRP, and the higher the overhead, the bigger the difference. The most significant difference occurs in scenario three, where the standard deviation of ECT MPCT is 8.8 time units compared with Hadoop's 34.7 time units, and the standard deviation of ECT NBRP is 10.9 compared with Hadoop's 96.4. This is because ECT has the capability to automatically adapt to the congestion (contention) level of the cluster network (VM disks), which the Hadoop scheme doesn't have. When the remote fetching overhead is sufficiently high, remote processing of data blocks in the ending stage is mostly rejected by ECT. As shown in Table 4.5, compared with the

Hadoop scheme, ECT only allows very limited (if any) number of data blocks to be processed remotely at different total numbers of data blocks in scenario three, which causes much more stable NBRP and hence much more stable MPCT. The benefit of stable NBRP and MPCT is that better performance can be achieved by splitting the input file into less number of data blocks, and hence incurring less overhead.

4.1.2 Highly Heterogeneous Environment

For the highly heterogeneous environment, it can be reasoned out from the simulation settings that the shortest possible processing time of one data block is 0.9 x 0.5$\boldsymbol{T}$ = 0.45$\boldsymbol{T}$ time units, and the longest possible processing time 1.1 x 1.5$\boldsymbol{T}$ = 1.65$\boldsymbol{T}$ time units, which is larger than three times the shortest time. As a result, ECT needs to assign more data blocks during its first stage to obtain the processing time estimates of all slots than it does in the slightly heterogeneous environment. Therefore, for each of the four scenarios, I ran the simulation with the total number of data blocks taking on values of 3000, 3500, 4000, …, 8500, which were slightly larger than the values in the slightly heterogeneous environment.

Figures 4.5, 4.6 and 4.7 present the MPCT's achieved by both schemes at different total numbers of data blocks in scenarios one, two and three, respectively. Simulation results of different total numbers of data blocks in scenarios one, two and three are shown in Tables 4.7, 4.8 and 4.9, respectively. (Only partial results are included due to length limitation. All results are the average of ten simulation runs.)

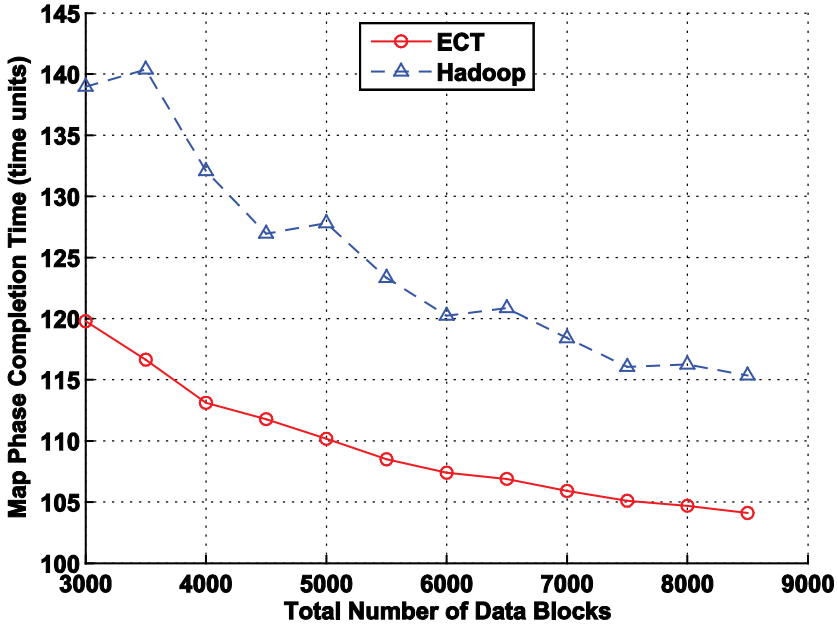

Figure 4.5: Map phase completion times of scenario one (highly heterogeneous)

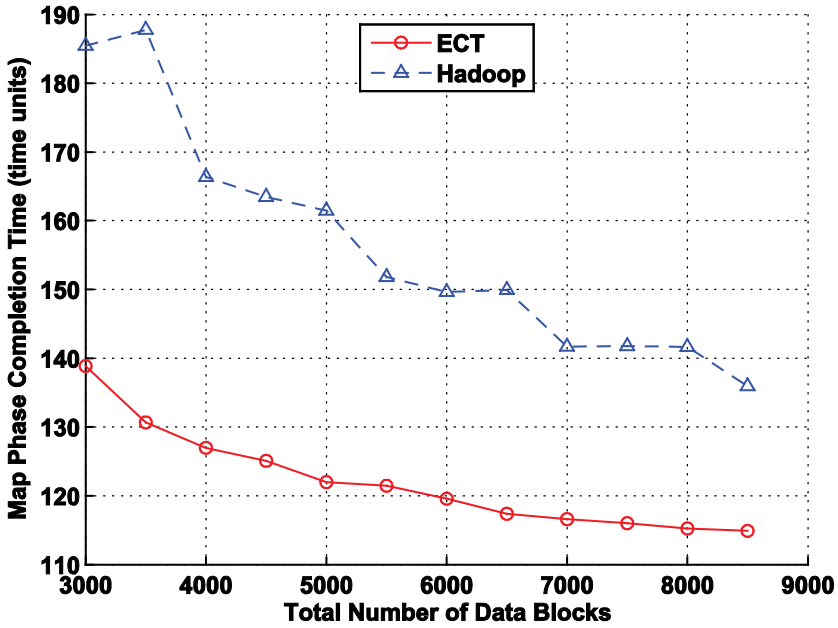

Figure 4.6: Map phase completion times of scenario two (highly heterogeneous)

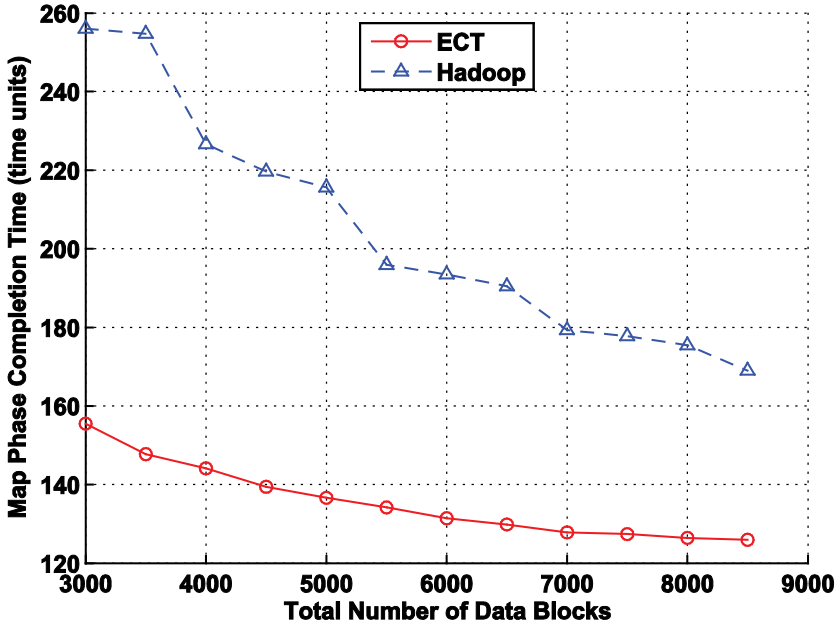

Figure 4.7: Map phase completion times of scenario three (highly heterogeneous)

Table 4.7: Simulation results of scenario one (highly heterogeneous environment, low remote fetching overhead, and stable slot processing speeds)

RPC = 1.5 **VPPT = ± 2.5%**	**Total Number of Data Blocks**						
	3000	**3500**	**4500**	**5500**	**6500**	**7500**	**8500**
Number of Blocks Assigned According to the ECT Rule	725.1	1225.5	2226.0	3224.3	4225.7	5227.8	6224.3
MPCT of Hadoop (time units)	139.0	140.4	127.0	123.3	120.9	116.1	115.3
MPCT of ECT (time units)	119.8	116.7	111.8	108.5	106.9	105.1	104.1
ECT Reduction in MPCT	13.8%	16.9%	11.9%	12.0%	11.6%	9.4%	9.7%
NBRP under Hadoop	435.3	468.4	631.3	733.7	883.4	1030.4	1145.0
NBRP under ECT	394.0	459.0	558.3	664.0	780.4	879.2	993.6
ECT Reduction in NBRP	9.5%	2.0%	11.6%	9.5%	11.7%	14.7%	13.2%

Table 4.8: Simulation results of scenario two (highly heterogeneous environment, medium remote fetching overhead, and relatively stable slot processing speeds)

RPC = 2.5 **VPPT = ± 5%**	**Total Number of Data Blocks**						
	3000	**3500**	**4500**	**5500**	**6500**	**7500**	**8500**
Number of Blocks Assigned According to the ECT Rule	700.1	1199.5	2198.2	3200.3	4196.4	5200.1	6195.4
MPCT of Hadoop (time units)	185.4	187.8	163.5	151.8	149.9	141.8	135.9
MPCT of ECT (time units)	138.9	130.7	125.1	121.4	117.4	116.0	114.9
ECT Reduction in MPCT	25.1%	30.4%	23.5%	20.0%	21.7%	18.2%	15.5%
NBRP under Hadoop	423.6	427.7	571.7	665.9	742.7	878.7	984.0
NBRP under ECT	241.2	273.3	378.4	455.3	527.5	625.1	699.8
ECT Reduction in NBRP	43.1%	36.1%	33.8%	31.6%	29.0%	28.9%	28.9%

Table 4.9: Simulation results of scenario three (highly heterogeneous environment, high remote fetching overhead, and less stable slot processing speeds)

RPC = 4.0 **VPPT = ± 10%**	**Total Number of Data Blocks**						
	3000	**3500**	**4500**	**5500**	**6500**	**7500**	**8500**
Number of Blocks Assigned According to the ECT Rule	645.2	1140.3	2138.2	3142.3	4140.6	5136.1	6139.9
MPCT of Hadoop (time units)	256.0	254.7	219.7	195.9	190.5	177.8	169.0
MPCT of ECT (time units)	155.5	147.8	139.5	134.2	129.9	127.4	125.9
ECT Reduction in MPCT	39.3%	42.0%	36.5%	31.5%	31.8%	28.3%	25.5%
NBRP under Hadoop	417.9	418.9	536.8	607.6	672.7	758.4	845.8
NBRP under ECT	119.1	151.2	206.7	268.4	340.5	400.7	457.3
ECT Reduction in NBRP	71.5%	63.9%	61.5%	55.8%	49.4%	47.2%	45.9%

The average results of *all* different total numbers of data blocks for each scenario is shown in Table 4.10, which indicate that the average MPCT achieved by ECT is 12.2%, 22.0% and 33.7% less than the one achieved by the Hadoop scheme, and the average NBRP under ECT is 11.4%, 31.4% and 53.7% less than the one under the Hadoop scheme in scenarios one, two and three, respectively. Even in scenario four, where the remote fetching overhead is zero, the average MPCT achieved by ECT is still 7.0% less than the one achieved by the Hadoop scheme.

Table 4.10: Simulation results of highly heterogeneous environment

	Scenario			
	One	**Two**	**Three**	**Four**
RPC	1.5	2.5	4.0	1.0
VPPT	± 2.5%	± 5%	± 10%	± 2.5%
Average of Hadoop MPCT (time units)	124.7	156.4	204.5	108.1
Standard Deviation of Hadoop MPCT (time units)	8.7	17.0	29.9	5.3
Average of ECT MPCT (time units)	109.5	122.1	135.6	100.5
Standard Deviation of ECT MPCT (time units)	5.0	7.3	9.5	3.4
ECT Reduction in Average MPCT	12.2%	22.0%	33.7%	7.0%
Average NBRP under Hadoop	780.6	685.1	621.8	—
Standard Deviation of NBRP under Hadoop	242.3	189.3	145.7	—
Average NBRP under ECT	691.6	469.7	287.7	—
Standard Deviation of NBRP under ECT	195.4	151.6	113.1	—
ECT Reduction in Average NBRP	11.4%	31.4%	53.7%	—

The MPCT reduction in this case is solely attributed to the ECT rule, which can yield better results than the simple Hadoop strategy even when the slot processing speeds fluctuate over time. And, the more stable the slot processing speeds, the less the MPCT achieved by ECT is expected to be due to the optimality of the ECT rule.

When comparing the results in Table 4.10 with the ones in Table 4.6, we can see that both schemes assigned more data blocks to remote slots in the highly heterogeneous environment due to the increased heterogeneity. As a result, the average MPCT of the Hadoop scheme increases slightly in scenarios one, two and three, where there exists remote fetching overhead. There is no remote fetching overhead in scenario four, hence the faster slots would get assigned more remote blocks than they would in the first three scenarios. In the ending stage of scenario four, most

running slots would be those faster ones processing remote blocks. Since the faster slots in the highly heterogeneous environment are faster than the ones in the slightly heterogeneous environment due to the simulation settings, the average MPCT of the Hadoop scheme in the former is less. On the other hand, ECT achieves smaller MPCT's in all four scenarios in the highly heterogeneous environment than it does in the slightly heterogeneous environment. This is because, in the former, the slot base processing time t's are evenly distributed within a wider range compared with the latter, whereas the VPPT still takes on the same values in all four scenarios. As a result, the distinction of slot processing times in the former is larger. Consequently, the estimation error of slot processing time impairs the optimality of the ECT rule to a less extent when ECT projects and compares the completion times of different slots, which yields less MPCT's.

Simulation results of the highly heterogeneous environment confirm again that ECT is much more robust to network congestion (disk contention) than the Hadoop scheme. As shown in Table 4.10, when RPC increases from 1.5 to 4.0, the average NBRP under ECT drops sharply from 691.6 to 287.7 (a 58.4% decrease), and the average MPCT achieved by ECT rises from 109.5 to 135.6 time units (a 23.8% increase). In contrast, the average NBRP under Hadoop only decreases from 780.6 to 621.8 (a 20.3% decrease), and the average MPCT achieved by Hadoop increases considerably from 124.7 to 204.5 time units (a 64.0% increase). The processing time traces of one typical case are shown in Figure 4.8, which exhibit the similar pattern as the ones in Figure 4.4 due to the same reason discussed in the previous section. Table 4.10 also includes the standard deviations of all simulation results in addition to the averages, which indicate again that the ECT results are more stable than the Hadoop results.

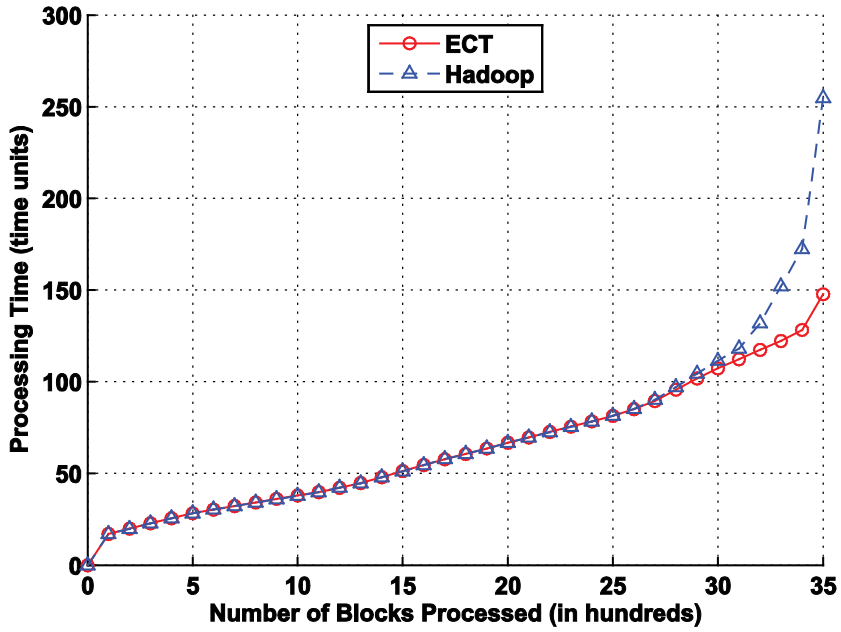

Figure 4.8: Processing time traces of scenario three (highly heterogeneous environment, Total Number of Data Blocks = 3500)

4.2 ECT Limitations

The proposed ECT scheme has its limitations. First, the performance improvement of MPCT decreases when the total number of data blocks increases for the reason explained earlier. Secondly, the performance improvement of MPCT decreases when the remote fetching overhead decreases. As discussed earlier, after most task slots run out of local blocks, the Hadoop scheme will incur large amount of remote processing which accounts for most part of the MPCT performance difference between ECT and the Hadoop scheme. Therefore, when the remote fetching overhead decreases, the performance difference decreases as well. Finally, the performance improvement of NBRP decreases when the remote fetching overhead decreases. This is because ECT will assign more data blocks to remote task slots when the remote fetching

overhead decreases. Although the Hadoop scheme will do so as well, ECT is much more sensitive to the remote fetching overhead, and hence will assign more data blocks than the Hadoop scheme.

4.3 Summary

In chapters three and four, I discuss the issues with the Hadoop task assignment scheme when Hadoop running in the clouds, and present an improved scheme ECT based on an optimal algorithm for a related deterministic scheduling problem. Extensive simulation results confirm that ECT could significantly outperform the Hadoop scheme with respect to both the completion time of map phase and the amount of remote processing employed.

CHAPTER FIVE: PARTITION REPLICA PLACEMENT POLICY FOR HADOOP DISTRIBUTED FILE SYSTEM

Today, Hadoop Distributed File System (HDFS) is widely used to provide scalable and fault-tolerant storage of large volumes of data. One of the key issues that affect the performance of HDFS is the placement of data replicas. Although the current HDFS replica placement policy can achieve both fault tolerance and read/write efficiency, the policy cannot evenly distribute replicas across cluster nodes, and has to rely on load balancing utility to balance replica distributions. In this chapter, I present a new replica placement policy for HDFS [71], which can generate replica distributions that are not only perfectly even but also meet all HDFS replica placement requirements.

5.1 Introduction

Today, Hadoop is widely used in many enterprises as a general purpose platform for distributed storage and processing of large data sets on commodity computer clusters. Prominent Hadoop users include Yahoo, Facebook, IBM, Twitter, and Adobe [3]. Many well-known enterprise vendors have been offering either commercial Hadoop products or technical support for Hadoop, including Amazon, Microsoft, Oracle and specialist Hadoop companies, such as Cloudera. HDFS is the storage part of the Hadoop framework, which is a distributed, scalable, and portable file system designed to run on commodity hardware. Although it has many similarities with other existing distributed file systems, HDFS is especially designed to be highly fault-tolerant, to provide high throughput access to application data, and to deal with very large data files (typically gigabytes to Terabytes in size).

In HDFS, each data file is stored as a sequence of blocks which are replicated for both data reliability and performance improvement. By default, each block has three replicas. The placement of these replicas is crucial to the performance of HDFS. The current HDFS Replica Placement Policy (RPP) is a rack-aware policy which can improve write performance while maintaining data reliability and read performance. The drawback of the policy is that it cannot evenly distribute replicas to cluster nodes. Since an unbalanced HDFS cluster can seriously degrade the performance of Hadoop applications, HDFS provides a balancing utility to address the issue. The utility can analyze replica placement and rebalance replicas across the cluster nodes at the cost of extra system resources and running time.

In this chapter, I present an innovative replica placement policy which addresses the above issue from a completely different perspective — it can distribute replicas to cluster nodes as evenly as possible, and also meet all replica placement requirements of HDFS. As a result, there is no need to run the balancing utility. To the best of my knowledge, the new policy is the first that addresses the load balancing issue by generating an even replica distribution in the first place.

The rest of the chapter is organized as follows. Section 2 introduces the HDFS RPP. Section 3 describes the proposed policy in detail. Evaluation results are presented in section 4, and I conclude in section 5.

5.2 Background

The current HDFS RPP (as of Hadoop 2.7.1) consists of the following three steps [2]. Assume the HDFS client runs outside the Hadoop cluster. The policy first places one replica on a random node. It then randomly selects another node on the same rack as the first node, and places

the second replica on the node. Finally, the policy places the third replica on a random node on a rack that is different from the one where the first two replicas reside. Since the replicas of a block are placed on only two unique racks instead of three, this policy can cut down the inter-rack write traffic when distributing data to cluster nodes. On the other hand, replicas on two different racks are mostly sufficient for maintaining good read performance and high fault tolerance. However, this policy cannot generate an even replica distribution due to its inherent unbalanced placement logic.

5.3 Partition Replica Placement Policy

The fundamental reason for the uneven replica distributions generated by the HDFS RPP is that it needs to place two replicas on one random rack and the third replica on another random rack. To overcome this problem, my new policy divides all available nodes into three sections. Section 1 has about two thirds of the nodes, and is used to store the first two replicas on the same rack. Sections 2 and 3 have about one third of the nodes, and are used to store the third replica. Since the partition scheme is the key of my new policy, which makes it possible to generate an even replica distribution, I call my new policy the Partition Replica Placement Policy (PRPP). Under PRPP, the whole replica placement process consists of two phases: section formation phase, and replica distribution phase.

In section formation phase, PRPP divides all nodes into three sections. If the total number of nodes is not a multiple of three, the policy will try to minimize the disparity between section 1 and the other two sections. Let $\boldsymbol{R}$ be a collection of $\boldsymbol{m}$ racks, each of which has $\boldsymbol{r_i}$ ($\boldsymbol{i}$ = 0, 1, 2, …,

m-1) available nodes on it. Let ***N*** be the total number of all available nodes, N_1 be the number of nodes to be assigned to section 1. ***N*** and N_1 are calculated according to the following formulas:

$$N = \sum_{i=0}^{m-1} r_i \tag{5.1}$$

$$d = N / 3, \quad r = N \% 3, \quad N_1 = \begin{cases} 2d & (r = 0 \text{ or } 1) \\ 2d + 2 & (r = 2) \end{cases} \tag{5.2}$$

, where "/" denotes integer division, and "%" integer remainder. (Note that N_1 is always even.)

After figuring out N_1, PRPP will assign nodes to section 1 from racks 0, 1, 2 and so on, until the number of nodes in section 1 has reached N_1. During this process, PRPP only assigns even number of nodes from each rack to section 1, and it will assign nodes to section 2 instead of section 1 in the following two cases:

- During the process of assigning nodes to section 1, if it encounters a rack having odd numbers of nodes, PRPP will assign all the nodes on that rack to section 1, except for the last node which will be assigned to section 2 instead.
- At the end of the process of assigning nodes to section 1, when the number of nodes in section 1 has reached N_1 , PRPP will assign all the remaining nodes on current rack (if any) to section 2.

If neither of the above situations occurs, there will be no node in section 2. After N_1 nodes have been assigned to section 1, all the remaining nodes will be assigned to section 3. Figure 5.1 shows an example of the formation of the three sections. Assume there are three racks, rack 0 has ten available nodes, rack 1 ten, and rack 2 seven. In this example: ***N*** = 27, ***d*** = ***N*** / 3 = 9, ***r*** = ***N*** % 3 = 0, and so $N_1 = 2d = 18$. PRPP first assigns all ten nodes on rack 0 to section 1, and then the

first eight nodes on rack 1 to section 1. Now the number of nodes in section 1 has reached 18, thus the last two nodes on rack 1 are assigned to section 2, and all the nodes on rack 2 are assigned to section 3. The following is the pseudocode of the section formation algorithm.

Algorithm 5.1: section formation algorithm

Input:

A collection of m racks, each of which has r_i (i = 0, 1, 2, …, m-1) available nodes on it.

Output:

S_1: Node section 1; S_2: Node section 2; S_3: Node section 3.

```
1.    calculate N and N1 according to formulas (5.1) and (5.2) respectively;
2.    TN = N1 ;              // Total number of nodes to be assigned to node section 1.
3.    for ( i = 0 to m -1 )  // Assign m racks of nodes to three node sections.
4.        if (TN > 0)        // Node section 1 still needs more nodes.
5.            if (TN ≥ ri )
6.                if ( ri is even )
7.                    assign all ri nodes on rack i to S1 ;
8.                    TN = TN - ri ;
9.                else
10.                   assign the first (ri -1) nodes on rack i
                      to S1 , and the last node to S2 ;
11.                   TN = TN - (ri -1) ;
```

12. **end if**

13. **else**

14. assign the first ***TN*** nodes on rack ***i*** to S_1 ,

    and the remaining nodes on rack ***i*** to S_2 ;

15. $TN = 0$;

16. **end if**

17. **else** *// The number of nodes in section 1 has reached* N_1 .

18. assign all the nodes on rack ***i*** to S_3 ;

19. **end if**

20. **end for**

In the following replica distribution phase, PRPP distributes replicas to all cluster nodes in the three sections formed in the previous phase. PRPP first constructs three replica assignment tables, one for each node section as shown in Figure 5.1.

Table	1								2		3			
Section	1								2		3			
Rack	0				1				1		2			
Node	0	1	…	9	10	11	…	17	18	19	20	21	…	26
Block No. of Replica														
	……								……		……			

Figure 5.1: Example of section formation and replica assignment tables

Table	1																	
Section	1																	
Rack	0										1							
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Block No. of Replica	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8
	9	9	10	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17
	……										……							

Figure 5.2: Replica placement before randomization

Table	1																	
Section	1																	
Rack	0										1							
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Block No. of Replica	2	0	3	1	3	4	1	0	2	4	6	5	5	8	6	7	7	8
	11	10	13	13	12	12	10	9	9	11	15	16	14	16	17	17	15	14
	……										……							

Figure 5.3: Replica placement after randomization

Each column in the replica assignment tables corresponds to the replica assignment of one node. All three tables have the same number of lines ***k***, which is calculated according to the following formula:

$$d = (n \times f) / N,\ r = (n \times f)\%N,\ k = \begin{cases} d & (r = 0) \\ d + 1 & (r \neq 0) \end{cases} \tag{5.3}$$

, where ***n*** is the total number of blocks, ***f*** the duplication factor of Hadoop (3 in this case), ***N*** the total number of nodes.

PRPP then uses a tabular scheme to distribute the two replicas that are supposed to be placed on the same rack to nodes in section 1 (table 1). As shown in Figure 5.2, replicas are filled into table 1 in block number order, from left to right, and from top to bottom. Each column in table 1 corresponds to the replica assignment of one node. For example, Node 0 has the replicas of blocks 0 and 9 assigned to it. Since the number of nodes on each rack in section 1 is even, the two replicas of all the blocks filled into table 1 will be in two different columns within the same rack portion, which means they are on two different nodes on the same rack. The replica placement in Figure 5.2 already meets all HDFS requirements except for that the adjacent nodes on each rack have identical replica assignment. To reduce the impact of the failure of one node on other nodes, the replica placement needs to be randomized so that no nodes have the same replica assignment. In the example shown in Figure 5.2, PRPP first randomizes each line in the rack 0 portion of table 1, then each line in the rack 1 portion. The result is shown in Figure 5.3. The randomization is done within each line and within each rack portion. Therefore, after the randomization, the two replicas of the same block will remain in two different columns in the same rack portion, or on two different nodes on the same rack per the HDFS replica placement rules. The following is the pseudocode of the replica distribution algorithm for node section 1.

Algorithm 5.2: replica distribution algorithm for node section 1

Input:

$S_1[0, 1, \ldots, N_1 - 1]$: Node section 1, which has N_1 nodes in it.

n data blocks to be distributed, which are numbered 0 through $(n - 1)$.

Output:

$T_1[k][N_1]$: Replica assignment table for node section 1.

```
1.    calculate k according to formula (5.3);
2.    for (i = 0 to N1 -1)
3.        CL[i] = 0;        // Reset current line number in table T1 for each node.
4.    end for
5.    P = 0;    // Reset current node pointer.
6.    for (i = 0 to n -1)
7.        CN = S1[P];        // Retrieve the next node in S1 as current node.
8.        T1[CL[CN]][CN] = i;        // Put the first replica of
                                     // current block on current node.
9.        CL[CN] = CL[CN] + 1;       // Increment current line number of
                                     // current node.
10.       P = P + 1;                 // Increment current node pointer.
11.       CN = S1[P];                // Retrieve the next node in S1 as current node.
12.       T1[CL[CN]][CN] = i;        // Put the second replica of
                                     // current block on current node.
13.       CL[CN] = CL[CN] + 1;
14.       P = P + 1;
15.       if (P == N1)
16.           P = 0;                 // Reset current node pointer
17.       end if                     // when it reaches the end.
```

18. **end for**

The following is the pseudocode of the randomization algorithm for node section 1.

Algorithm 5.3: randomization algorithm for node section 1

Input:

$S_1[0, 1, \ldots, N_1 - 1]$: Node section 1, which has N_1 nodes in it.

$T_1[k][N_1]$: Replica assignment table before randomization.

Output:

$T_1[k][N_1]$: Replica assignment table after randomization.

1. $TN = N_1$; *// Total number of nodes to be processed.*
2. $P = 0$; *// Current node pointer.*
3. $CN = S_1[P]$; *// Retrieve the first node in S_1 as current node.*
4. **while** ($TN > 0$)
5. CR = rackNumberOf(CN) *// rackNumberOf(**CN**) is the rack // number of **CN**.*
6. $C = 0$; *// The count of nodes retrieved that are on the same rack.*
7. $RP = 0$; *// Retrieved nodes pointer.*
8. **do** *// Retrieve nodes in S_1 that are on the same rack.*
9. $S[RP] = CN$; *// Save current node in **S**.*
10. $RP = RP + 1$;

```
11.                 C = C + 1;
12.                 TN = TN - 1;
13.                 P = P + 1;       // Increment current node pointer.
14.                 if (P == N1)     // Reset the pointer when it reaches the end.
15.                     P = 0;
16.                 end if
17.                 CN = S1[P];    // Retrieve the next node in S1 as current node.
18.             while (CR == rackNumberOf(CN))
19.             for (i = 0 to k -1 )   // Randomize the replica placement on the
                                       // retrieved nodes.
20.                 for (j = 0 to C-1)          // Retrieve one line of replicas from the
21.                     L[j] = T1[i][S[j]];  // portion of T1 that corresponds to the above
22.                 end for                 // retrieved nodes and save it in L[0,1, ..., C-1].
23.                 randomize the elements of L[0, 1, …, C-1];
24.                 for (j = 0 to C-1)          // Copy the randomized result back to T1.
25.                     T1[i][S[j]] = L[j];
26.                 end for
27.             end for
28.     end while
```

After distributing the first two replicas to nodes in section 1, PRPP continues to distribute the third replicas to nodes in sections 2 and 3. The distribution to section 3 nodes is fairly simple

as any replica can be placed on any node. However this is not true for the nodes in section 2. Each section 2 node is on the same rack as certain section 1 nodes where the first two replicas of certain blocks reside. Thus, the third replica of those blocks cannot be put on that section 2 node. Under PRPP, the replica distribution to all nodes will be as even as possible. If the total number of blocks is a multiple of the total number of nodes in sections 2 and 3, each node in sections 2 and 3 will have the same number of replicas assigned after the distribution process ends. Otherwise, PRPP will have to assign one more block to certain nodes, and it will choose section 3 nodes first as any replica can be distributed to section 3 nodes. Before the distribution process starts, PRPP calculates n_2 and n_3, which are the total number of replicas to be distributed to the nodes in sections 2 and 3 respectively, according to the following formulas:

$$d = n/(N_2+N_3) \quad , \quad r = n\%(N_2+N_3) \quad , \quad n_2 = \begin{cases} d \times N_2 & (r \le N_3) \\ d \times N_2 + r - N_3 & (r > N_3) \end{cases} \tag{5.4}$$

$$n_3 = n - n_2 \tag{5.5}$$

, where n is the total number of blocks, N_2 the number of nodes in section 2, N_3 the number of nodes in section 3.

Since not all replicas can be distributed to section 2 nodes, PRPP starts with section 2. The replicas are filled into the replica assignment table for section 2 in block number order, from left to right, and from top to bottom. Before it assigns the third replica of current block to current node, PRPP checks whether the first two replicas of current block reside on the same rack as current node. If so, PRPP discards current block and keeps trying the next block in block number order

until it finds one that meets the requirement and assigns it to current node. Otherwise, PRPP assigns the third replica of current block to current node, and proceeds to next block and next node. The process continues until PRPP has assigned n_2 replicas. Finally, PRPP fills the third replicas of the remaining n_3 blocks into the replica assignment table for section 3 in the same way without performing the check it does for section 2 nodes. The following is the pseudocode of the replica distribution algorithm for node sections 2 and 3.

Algorithm 5.4: replica distribution algorithm for node sections 2 and 3

Input:

$S_2[0, 1, \ldots, N_2 - 1]$: Node section 2, which has N_2 nodes in it.

$S_3[0, 1, \ldots, N_3 - 1]$: Node section 3, which has N_3 nodes in it.

n data blocks to be distributed, which are numbered 0 through (n - 1).

Output:

$T_2[k][N_2]$: Replica assignment table for node section 2.

$T_3[k][N_3]$: Replica assignment table for node section 3.

```
1.    calculate n2 and n3 according to formulas (5.4) and (5.5) respectively;
2.    for (i = 0 to n -1)
3.        A[i] = false;    // A[i] indicates whether block i has been assigned.
4.    end for
5.    for (i = 0 to N2 -1)
6.        CL[i] = 0;    // Reset current line number in table T2 for each node.
```

```
7.      end for
8.      P = 0;              // Reset current node pointer.
9.      CB = 0;             // Reset current block number.
10.     for (i = 1 to n2)           // Distribute n2 replicas to section 2 nodes.
11.         CN = S2[P];             // Retrieve the next node in S2 as current node.
12.         P = P + 1;
13.         if (P == N2)            // Reset current node pointer when it reaches
14.             P = 0;              // the end.
15.         end if
16.         while (rackNumberOf(CB) == rackNumberOf(CN)
                or A[CB] == true)
    // Find the CB that meets the requirement; rackNumberOf (CB) is
    // the number of the rack where the first two replicas of CB reside;
    // rackNumberOf (CN) is the rack number of CN.
17.             CB = CB + 1;
18.             if (CB == n)                // Reset current block number
19.                 CB = 0;                 // when it reaches the end.
20.             end if
21.         end while
22.         T2[CL[CN]][CN] = CB;        // Place the above found CB on
                                        // current node.
23.         CL[CN] = CL[CN] + 1;
```

```
24.          A[CB] = true;                    // CB has been assigned.
25.          CB = CB + 1;
26.          if (CB == n)
27.              CB = 0;
28.          end if
29.      end for
30.      P = 0;              // Reset current node pointer.
31.      CB = 0;             // Reset current block number.
32.      for (i = 1 to n3)           // Distribute the remaining n3 replicas to
                                     // section 3 nodes.
33.          CN = S3[P];             // Retrieve the next node in S3 as current node.
34.          P = P + 1;
35.          if (P == N3)     // Reset current node pointer when it reaches the end.
36.              P = 0;
37.          end if
38.          while (A[CB] == true)        // Find the next unassigned block CB.
39.              CB = CB + 1;
40.          end while
41.          T3[CL[CN]][CN] = CB;         // Place the above found CB on
                                          // current node.
42.          CL[CN] = CL[CN] + 1;
43.          CB = CB + 1;
```

44. **end for**

5.4 Evaluation

In statistics, the number of replicas on one single node in the replica distributions generated by the HDFS RPP is a Discreet Random Variable (DRV). To examine the distribution of a DRV, theoretical simulation is more appropriate than actual implementation because much more distribution samples can be obtained by simulation. Therefore, I conducted large scale simulation to examine the replica distribution generated by HDFS RPP. The simulation was conducted in three scenarios as shown in Table 5.1. Scenarios one and two resemble the two common situations Hadoop applications encounter in practice. In scenario one, the application has a dedicated cluster where all nodes on all racks are available to it. While in scenario two, the application shares the cluster with other applications, and hence only partial cluster nodes are available to it.

Table 5.1: Simulation settings

	Scenario One	**Scenario Two**	**Scenario Three**
Total Number of Blocks	8,000	8,000	54
Duplication Factor	3	3	3
Total Number of Replicas	24,000	24,000	162
Total Number of Nodes	600	600	27
Average Number of Replicas on One Node	40	40	6
Total Number of Racks	30	45	3
Number of Available Nodes on Rack *i*	20	20 (i = 0, 3, 6, …, 42) 15 (i = 1, 4, 7, …, 43) 5 (i = 2, 5, 8, …, 44)	10 (i = 0, 1) 7 (i = 2)

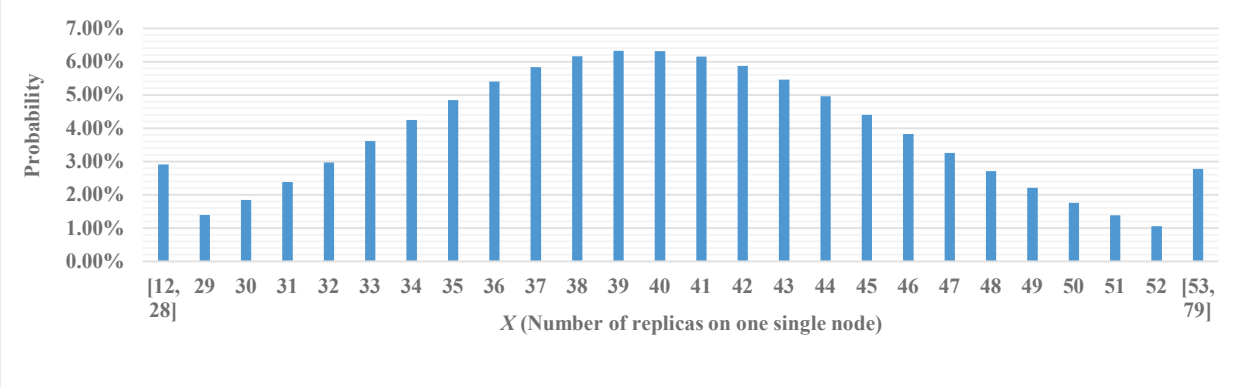

Figure 5.4: Probability distribution of X in scenario one

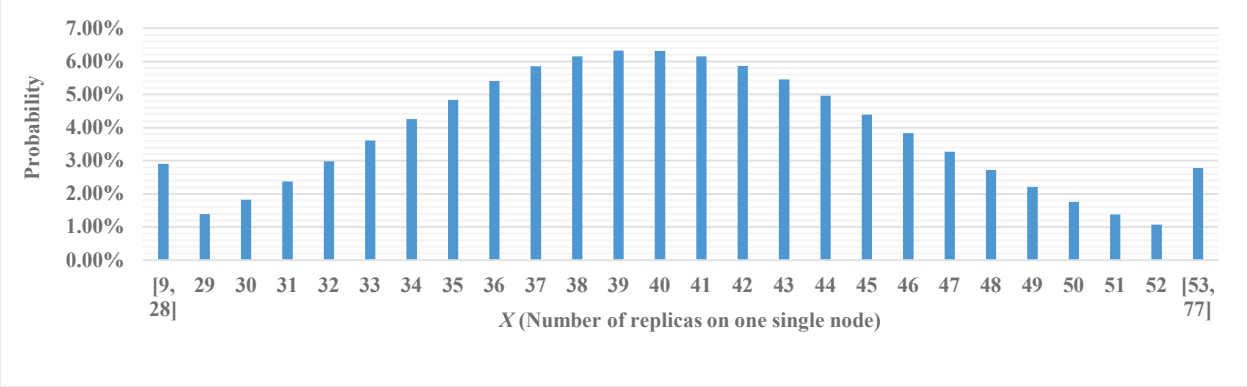

Figure 5.5: Probability distribution of X in scenario two

Table 5.2: Statistics results of HDFS RPP samples

	Scenario One	**Scenario Two**
Minimum Value	12	9
Maximum Value	79	77
Sample Mean	40.0	40.0
Sample Variance	39.83	39.80
Sample Standard Deviation	6.31	6.31

Let X be the number of replicas on one single node in the replica distributions generated by HDFS RPP. Figure 5.4 and Figure 5.5 show the probability distribution of X in Scenarios one and two respectively, based on 10,000 simulation runs. The simulation results confirm that, in both scenarios one and two, HDFS RPP generates uneven replica distributions. The number of replicas on one node spreads over a wide range from 12 to 79 in scenario one and from 9 to 77 in scenario two. Table 5.2 shows the statistics results calculated based on all the samples generated by HDFS RPP in the simulation. It can be observed that the two probability distributions shown in Figure 5.4 and Figure 5.5 are very close. This is because HDFS RPP randomly selects cluster nodes directly instead of selecting a cluster rack first, and hence the node distribution across racks doesn't significantly affect the replica distribution. If the policy randomly selects a rack first, the replica distribution would be more uneven when the node distribution is heterogeneous, because in general the nodes on racks with less nodes would have more replicas than the nodes on racks with more nodes.

On the other hand, the simulation also confirms that the proposed PRPP can generate perfectly even replica distributions in both scenarios one and two, where each node has 40 replicas on it. Due to the large scale of the first two scenarios, the replica distributions generated in them cannot be presented in this dissertation. Therefore, I include scenario three, which is in reduced scale as shown in Table 5.1, to present the replica distributions generated by both HDFS RPP and PRPP.

Figure 5.6 shows one replica distribution generated by HDFS RPP in scenario three. The replica distribution is uneven with number of replicas on one single node ranging from 2 to 10. In

contrast, as shown in Figure 5.7, the replica distribution generated by PRPP in scenario three is not only perfectly even, but also meets all HDFS replica placement requirements.

Rack	0										1										2						
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Block No. of Replica	7	2	7	9	20	9	0	3	3	4	8	15	1	20	4	17	1	6	13	6	6	0	16	2	0	9	11
	16	11	8	12	40	16	2	32	22	8	10	25	5	27	17	20	7	28	14	15	14	22	18	5	1	24	19
	26	18	24	25	41	27	4	52	53	11	13	26	19	30	23	23	21	32	40	19	21	31	24	10	3		22
	29	49	29	31		30	12	53		18	26	36	34	47	35	28	30	36	46	29	45	35	28	12	5		39
	33	52	43	33		31	13			23	38	42	37		44	46	37	41	48	39		36	50	17	10		45
	34		51	50		45	25			34	47	48	40			53	38					43	51	21	14		51
	37					49	27			38		49					42					46		35	15		
	41						32			43												50		39	33		
							42			47														44	44		
										48														52			

Figure 5.6: Replica distribution generated by HDFS RPP in scenario three

Rack	0										1										2						
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Section	1										1								2		3						
Block No. of Replica	2	0	3	1	3	4	1	0	2	4	6	5	5	8	6	7	7	8	0	1	5	6	7	8	14	15	16
	11	10	13	13	12	12	10	9	9	11	15	16	14	16	17	17	15	14	2	3	17	20	21	22	23	24	25
	21	21	19	19	20	22	20	22	18	18	23	24	24	26	23	26	25	25	4	9	26	27	28	29	30	31	32
	28	31	31	28	27	29	30	27	30	29	33	35	34	34	33	32	32	35	10	11	33	34	35	36	37	38	39
	36	37	39	40	40	39	36	38	37	38	41	42	42	43	44	43	41	44	12	13	40	41	42	43	44	45	46
	49	48	49	47	47	48	45	46	45	46	52	52	51	50	51	53	50	53	18	19	47	48	49	50	51	52	53

Figure 5.7: Replica distribution generated by PRPP in scenario three

5.5 Summary

In this chapter, I present PRPP, a new replica placement policy for HDFS, which addresses the load balancing issue by evenly distributing replicas to cluster nodes, and hence eliminates the need for running any load balancing utility. The simulation results confirm that PRPP can generate replica distributions that are perfectly even and also comply with all HDFS replica placement rules.

CHAPTER SIX: SLOT REPLICA PLACEMENT POLICY FOR HADOOP DISTRIBUTED FILE SYSTEM

Nowadays, Big Data problems are ubiquitous, which in turn creates huge demand for data-intensive computing. The advent of Cloud Computing has made data-intensive computing much more accessible and affordable than ever before. One of the crucial issues that can significantly affect the performance of data-intensive applications is the load balance among cluster nodes. In Chapter five of this dissertation, I address the load balance problem in the context of HDFS, a widely used file system for data-intensive computing on Cloud platforms, and present PRPP, a new replica placement policy, which can distribute replicas evenly to cluster nodes according to the HDFS replica placement rules. The drawback of PRPP is that it only works for homogeneous cluster environments. In this chapter, I present an improved replica placement policy for HDFS [72], which can perfectly balance the computing load among all cluster nodes in both homogeneous and heterogeneous cluster environments.

6.1 Introduction

We are now in the age of Big Data. The need to process large volumes of data in a timely manner has become a necessity for many enterprises across a wide range of industries. Big data problems have been emerging everywhere, as well as the technological solutions. Since it was first proposed by Google in 2004, MapReduce [1] has become the most popular programming model for parallel and distributed computing. And its open source implementation Hadoop [2] has obtained great popularity by making data-intensive computing possible for ordinary users without any extensive experience with parallel and distributed computing. Many well-known enterprises

have been using Hadoop for their data-intensive processing needs, including Yahoo, Facebook, IBM, Twitter, and Adobe [3]. And big IT companies, such as Microsoft, Oracle and Amazon are providing either commercial Hadoop products or technical support for Hadoop. Moreover, the innovative idea of utility computing has made data-intensive computing much more feasible for users who either cannot afford or do not have the interest to purchase and maintain computer clusters of sufficient size for the data-intensive problems they face in their business operation. Today, any user can pay a modest fee to rent a cluster consisting of virtual machines running Hadoop, from a utility cloud provider, such as Amazon Elastic Compute Cloud (EC2) [5] to process Terabyte datasets within a reasonable time period.

One of the key issues that can significantly affect the performance of data-intensive applications is load balance among cluster nodes, which can optimize resource usage, maximize throughput, minimize job completion time, and avoid overload on cluster nodes. In this chapter, I address the load balance problem in the context of HDFS, the underlying distributed file system of Hadoop. I propose an improved Replica Placement Policy (RPP) for HDFS. With this new policy, my contribution is twofold. First, the HDFS RPP can only generate unbalanced replica distributions among cluster nodes, and has to rely on a load balancing utility to balance the load. In contrast, my new policy can generate replica distributions that are perfectly even, and hence eliminates the need for running any load balancing utility. Second, the HDFS RPP cannot deal with any heterogeneity in processing capabilities of different cluster nodes. In contrast, my policy is devised for both homogeneous and heterogeneous cluster environments, and it can achieve perfect load balance even when cluster nodes have different processing capabilities.

The rest of the chapter is organized as follows. Section 2 describes the proposed policy in detail. Evaluation results are presented in Section 3, and I conclude in Section 4.

6.2 Slot Replica Placement Policy

Since the key of my new replica placement policy is to assign replicas to slots on nodes instead of nodes themselves, I call it Slot Replica Placement Policy (SRPP). To deal with heterogeneous cluster environments, SRPP assigns processing slots to cluster nodes before it starts distributing replicas to nodes. Assume we have a cluster consisting of m nodes, each of which has C_i (i = 0, 1, 2, …, m-1) units of Processing Capability (PC). One node having C_i units of PC means it can process C_i data blocks within one unit time period. In this example, SRPP will first assign C_i processing slots to each node in the cluster, and then assign replicas to processing slots on nodes instead of nodes themselves. As long as the replica distribution over all processing slots is even, the load balance among all nodes can be achieved. Note that SRPP works for homogeneous cluster environments as well, which are just one special case, where C_i = 1 (i = 0, 1, 2, …, m-1). To generate even replica distribution over slots, SRPP first divides all slots into different sections, and then distributes replicas to the slots. Under SRPP, the replica placement process consists of two phases: slot section construction phase and block replica distribution phase.

In the slot section construction phase, SRPP divides the processing slots on all nodes into three sections. Slot section 1 contains about two thirds of the slots, and is used to store the first two replicas that are supposed to be on the same rack. Slot sections 2 and 3 have about one third of the slots, and are used to store the third replicas. In the case the total number of slots is not a multiple of three, SRPP will try to minimize the disparity between section 1 and the other two

sections. Let N be the total number of processing slots on all nodes, N_1 be the number of slots to be assigned to section 1. N_1 is calculated according to the following formula:

$$d = N / 3, \quad r = N \% 3, \quad N_1 = \begin{cases} 2d & (r = 0 \text{ or } 1) \\ 2d + 2 & (r = 2) \end{cases} \tag{6.1}$$

, where "/" denotes integer division, and "%" integer remainder. (Note that N_1 is always even.)

Since different generations of hardware are usually located on different racks, I assume that nodes on different racks may belong to different hardware generations, but nodes on the same rack have the same processing capability. To construct the three slot sections, SRPP first assigns slots to section 1 from nodes 0, 1, 2 and so on, until the number of slots in section 1 has reached N_1. The slot assignment is performed in runs. For each run, SRPP takes two consecutive nodes on the same rack (the reason for this will be explained later in this chapter), and assigns the slots on those two nodes to section 1. If there are certain odd number of nodes on the rack, the slots on the last node are assigned to section 2 instead. When the process reaches the end, the last two nodes taken by SRPP may have more slots than what is needed to be assigned to section 1. Assume section 1 only needs m more slots (m is always even), while either of the two nodes taken in the last run has n slots, and $2n > m$. In this case, SRPP assigns the first $m/2$ slots on both nodes to section 1, and the remaining slots on both nodes to section 2. If there are any nodes left on the same rack as the last two nodes, all slots on those nodes are assigned to section 2 as well. Finally, SRPP assigns all slots on all remaining nodes to section 3.

Rack	0							1				2				3
Node	0	1	2	3	4	5	6	7	8	9	10	11		12		13
Slot	0	1	2	3	4	5	6	7, 8	9, 10	11, 12	13, 14	15, 16	17, 18	19, 20	21, 22	23 - 26
Slot Section	1						2	1				1	2	1	2	3
Block No. of Replica	Replica Assignment															

Figure 6.1: Example of slot section construction and replica assignment table

Figure 6.1 shows an example of the construction of the three slot sections. There are four racks in total: rack 0 has seven nodes with 1 unit of PC, rack 1 has four nodes with 2 units of PC, rack 2 has two nodes with 4 units of PC, and rack 3 has one node with 4 units of PC. In this example, $N = 27$, $d = N / 3 = 9$, $r = N \% 3 = 0$, and $N_1 = 2d = 18$. Rack 0 has seven nodes, so SRPP assigns slots 0 through 5 to section 1 in the first three runs: slots 0 and 1 in the first run, slots 2 and 3 in the second run, and slots 4 and 5 in the third run. Since nodes 6 and 7 are on different racks, slot 6 is assigned to section 2. Rack 1 has four nodes, so all slots on rack 1 are assigned to section 1 in two consecutive runs. Now section 1 only needs four more slots, but both node 11 and node 12 has four slots. Therefore, the first two slots on both node 11 and node 12 are assigned to section 1, and the other two slots are assigned to section 2. As a result, nodes 11 and 12 have slots in both section 1 and section 2. Finally, SRPP assigns all slots on node 13 to section 3. The following is the pseudocode of the slot section construction algorithm.

Algorithm 6.1: slot section construction algorithm

Input:

A collection of m (n_0, n_1, …, n_{m-1}) nodes, each of which has r_i (i = 0, 1, 2, …, m -1) slots on it.

Output:

S_1: Slot section 1; S_2: Slot section 2; S_3: Slot section 3.

1. calculate N_1 according to formula (6.1);
2. $TN = N_1$; *// Total number of slots to be assigned to slot section 1.*
3. $x = 0$; *// Current node number.*
4. **while** ($TN > 0$)
5. **while** (n_x and n_{x+1} are not on the same rack)
6. assign all slots on n_x to S_2 ;
7. $x = x + 1$;
8. **end while**
9. **if** ($TN \geq r_x + r_{x+1}$)
10. assign all slots on n_x and n_{x+1} to S_1 ;
11. $TN = TN - r_x - r_{x+1}$;
12. **else**
13. assign the first (TN / 2) slots on n_x and n_{x+1} to S_1 ;
14. assign the remaining slots on n_x and n_{x+1} to S_2 ;
15. $TN = 0$;
16. **end if**
17. **end while**
18. **if** (there are remaining nodes on the rack of n_x and n_{x+1})
19. assign all the slots on all remaining nodes to S_2 ;
20. **end if**

21. assign all the slots on all remaining nodes to S_3;

In the following block replica distribution phase, SRPP distributes replicas to all slots in the three slot sections constructed in the previous phase. The replica assignment is stored in a table as shown in Figure 6.1. Each column in the replica assignment table corresponds to the replica assignment of one slot and ***k***, the number of lines in the table, is calculated according to the following formula:

$$d = (n \times f) / N,\ r = (n \times f)\%N,\ k = \begin{cases} d & (r = 0) \\ d + 1 & (r > 0) \end{cases} \tag{6.2}$$

, where ***n*** is the total number of blocks, ***f*** the duplication factor of HDFS (3 in this case), ***N*** the total number of slots.

Rack	**0**				**1**			
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Slot 0	1	3	7	1	2	2	4	5
Slot 1	8	7	8	3	4	6	5	6
Slot 2	9	9	10	X				
Slot 3	11	10	11	X				

Figure 6.2: Example one of replica distribution to slot section 1

Rack	0				1				Run
Node	0	1	2	3	4	5	6	7	
Slot 0	1	3	1	3	2	2	4	4	1
Slot 1	8	7	7	8	5	6	5	6	2
Slot 2	9	9	10	10					3
Slot 3	12	11	11	12					4

Figure 6.3: Example two of replica distribution to slot section 1

To distribute replicas evenly to slots, SRPP fills out the replica assignment table line by line from top to bottom. SRPP starts with slot section 1. The process is more complicated than it looks like, because it doesn't work to just randomly select two slots on the same rack from slot section 1, and place the first two replicas of certain block on them as shown in Figure 6.2. In this example, rack 0 has four nodes, each of which has four slots, and rack 1 also has four nodes, each of which has two slots. Slots are randomly selected. The numbers in the table represent the order in which they are selected. In Figure 6.2, slot 0 on node 0 and slot 0 on node 3 are first selected, then slot 0 on node 4 and slot 0 on node 5. The random selection continues until slot 3 on node 0 and slot 3 on node 2 are chosen. Now the last two slots left are slots 2 and 3 on node 3, which have become useless as they are on the same node. To avoid this problem, SRPP distributes replicas to section 1 slots in runs as shown in Figure 6.3. For each run, SRPP only picks one slot from each node and assigns one replica to the slot. When slots are assigned to section 1 in the slot section construction phase, nodes are selected in pairs from each rack. Therefore, there are always even number of nodes on each rack. If one slot is selected from each node, the number of slots selected on each rack is always even as well. As a result, if SRPP selects one slot on certain rack, it can always find another slot on the same rack but on a different node. In the example shown in Figure 6.3, each slot will get one replica assigned after four runs of distribution. Under SRPP, this process

continues until the first two replicas of all data blocks are distributed evenly to section 1 slots. The following is the pseudocode of replica distribution algorithm for slot section 1.

Algorithm 6.2: replica distribution algorithm for section 1

Input:

S_1 [0, 1, …, N_1 -1]: Slot section 1, which has N_1 slots in it.

NS_1 [0, 1, …, M_1 -1]: Node section 1, each node in NS_1 has at least one slot in S_1 .

n data blocks to be distributed, which are numbered 0 through (n - 1).

Output:

Replica assignment for S_1 .

```
1.    CB = 0;              // Current block number.
2.    for (i = 0 to M1 -1)
3.        P[i] = 0;        // Pointer pointing to the next slot on node NS1 [i].
4.    end for
5.    while (CB < n)
6.        SC = 0;          // Slot count.
7.        clear SP;        // SP is a slot pool for random selection.
8.        for (i = 0 to M1 -1)
9.            if (the slot pointed by P[i] is in S1)    // Some nodes may have
                                                        // slots in both S1 and S2 .
10.               store the slot pointed by P[i] in SP;
```

11. $P[i] = P[i] + 1$;
12. $SC = SC + 1$;
13. **end if**
14. **end for**
15. **if** ($SC == 0$) *// After one run of replica distribution is completed,*
16. **for** ($i = 0$ **to** $M_1 - 1$) *// reset the pointers to start the next run.*
17. $P[i] = 0$;
18. **end for**
19. **end if**
20. **while** ($SC > 0$ **and** $CB < n$)
21. randomly select slot RS_1 from SP;
22. $SC = SC - 1$;
23. assign the first replica of block CB to RS_1 ;
24. randomly select slot RS_2 on the same rack as RS_1 from SP;
25. $SC = SC - 1$;
26. assign the second replica of block CB to RS_2 ;
27. $CB = CB + 1$;
28. **end while**
29. **end while**

After the replica distribution to section 1 slots is completed, SRPP continues with slot section 2. Since section 2 slots are on the same rack as the first two replicas of certain blocks, the

replica distribution has to meet the requirement that the third replica must be put on a rack that is different from the one where the first two replicas reside. Finally, SRPP distributes all remaining replicas to slots in section 3. The replica distribution to section 3 slots is simpler, as any replica can be put on any slot. If the total number of blocks is a multiple of the total number of slots in sections 2 and 3, each slot will get the same number of replicas assigned. Otherwise, SRPP will select some slots that will get one more replica. In this case, SRPP will choose section 3 slots first. Before it assigns replicas to slots in sections 2 and 3, SRPP first calculates n_2 and n_3, which are the total number of replicas to be distributed to the slots in sections 2 and 3 respectively, according to the following formulas:

$$d = n/(N_2+N_3) \quad , \quad r = n\%(N_2+N_3) \quad , \quad n_2 = \begin{cases} d \times N_2 & (r \le N_3) \\ d \times N_2 + r - N_3 & (r > N_3) \end{cases} \tag{6.3}$$

$$n_3 = n - n_2 \tag{6.4}$$

, where n is the total number of blocks, N_2 the number of slots in section 2, N_3 the number of slots in section 3.

SRPP distributes replicas by filling out the portion of replica assignment table for slot sections 2 and 3 line by line from top to bottom. Replicas are filled into the assignment table in block number order. For each replica assignment to section 2 slots, SRPP needs to find one unassigned block of which the first two replicas are not on the same rack as current slot. The replica distribution process continues until n_2 replicas have been assigned. For each replica assignment to section 3 slots, SRPP only needs to find one unassigned block. The process continues until n_3

replicas have been assigned. The following is the pseudocode of the replica distribution algorithm for slot sections 2 and 3.

Algorithm 6.3: replica distribution algorithm for sections 2 and 3

Input:

S_2 [0, 1, …, N_2 -1]: Slot section 2, which has N_2 slots in it.

S_3 [0, 1, …, N_3 -1]: Slot section 3, which has N_3 slots in it.

n data blocks to be distributed, which are numbered 0 through (n - 1).

Output:

Replica assignment for S_2 and S_3 .

```
1.    calculate n2 and n3 according to formulas (6.3) and (6.4) respectively;
2.    for (i = 0 to n -1)
3.        A[i] = false;    // A[i] indicates whether block i has been assigned.
4.    end for
5.    P = 0;               // Current slot pointer.
6.    CB = 0;              // Current block number.
7.    for (i = 1 to n2)    // Distribute n2 replicas to slots in S2 .
8.        CS = S2 [P];     // Retrieve the next slot in S2 as current slot.
9.        P = P + 1;
10.       if (P == N2)     // Reset current slot pointer when it reaches the end.
11.           P = 0;
```

```
12.         end if
13.         while (the first two replicas of block CB are on the same rack as CS
                    or A[CB] == true)
14.             CB = CB + 1;
15.             if (CB == n)    // Reset current block number
16.                 CB = 0;     // when it reaches the end.
17.             end if
18.         end while
19.         assign the third replica of block CB to CS;
20.         A[CB] = true;
21.         CB = CB + 1;
22.         if (CB == n)
23.             CB = 0;
24.         end if
25.     end for
26.     P = 0;                  // Reset current slot pointer.
27.     CB = 0;                 // Reset current block number.
28.     for (i = 1 to n3)       // Distribute the remaining n3 replicas to slots in S3 .
29.         CS = S3 [P];        // Retrieve the next slot in S3 as current slot.
30.         P = P + 1;
31.         if (P == N3)        // Reset current slot pointer when it reaches the end.
32.             P = 0;
```

```
33.         end if
34.         while (A[CB] == true)      // Find the next unassigned block CB.
35.             CB = CB + 1;
36.         end while
37.         assign the third replica of block CB to CS;
38.         CB = CB + 1;
39.     end for
```

6.3 Evaluation

I conducted extensive simulation to examine the replica distributions generated by SRPP. The simulation results confirm that SRPP can generate perfectly even replica distributions except for certain rare cases, where either there are too few or no slots in section 3 or SRPP cannot find enough slots for section 1. The problem only occurs when the node (slot) distribution across racks is dramatically irregular, and it can usually be solved by randomly changing the rack order in which SRPP adds slots to the three slot sections.

Due to the page limitation, I can only present the results of example simulation scenarios that are in reduced scale. Table 6.1 shows the settings of three example simulation scenarios. Scenario one represents a typical heterogeneous cluster environment where there are three different generations of hardware: nodes on racks zero and one have 1 unit of PC, nodes on rack two 2 units, and nodes on rack three 4 units.

Table 6.1: Simulation settings

	Scenario One	Scenario Two	Scenario Three
Total Number of Blocks	60	60	60
Duplication Factor	3	3	3
Total Number of Replicas	180	180	180
Number of Racks	4	4	3
Total Number of Nodes	21	13	11
Number of Nodes on Each Rack	{9, 5, 6, 1}	{6, 4, 2, 1}	{5, 4, 2}
Processing Capability of Nodes on Each Rack (Units)	{1, 1, 2, 4}	{1, 2, 4, 8}	{2, 3, 4}
Number of Slots on Nodes on Each Rack	{1, 1, 2, 4}	{1, 2, 4, 8}	{2, 3, 4}
Total Number of Slots	30	30	30
Average Number of Replicas on One Slot	6	6	6

Scenario two represents a highly heterogeneous environment where there exist four different generations of hardware: nodes on rack zero have 1 unit of PC, nodes on rack one 2 units, nodes on rack two 4 units, and the node on rack three 8 units. Finally, scenario three represents a heterogeneous cluster where nodes on different racks have different but relatively close processing capabilities compared with the first two scenarios.

As shown in Figures 6.4, 6.6 and 6.8, HDFS RPP generates unbalanced replica distributions in all three scenarios. (Note that HDFS RPP treats all nodes in a heterogeneous cluster the same, and distributes replicas to nodes instead of slots. The slot columns in Figures 6.4, 6.6 and 6.8 are only for comparison purpose.)

Rack	0									1					2												3			
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15		16		17		18		19		20			
Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
Block No. of Replica	1	0	9	2	6	0	1	4	4	15	0	5	11	13	14	18	14	21	1	4	7	16	10	21	3	7	34	49		
	28	3	12	20	19	12	2	5	5	25	2	6	17	25	28	31	30	50	26	41	17	28	22	27	11	16				
	29	8	15	21	40	14	3	6	7	33	8	9	19	26	32	34			43	58	31	34	46	51	32	40				
	36	9	22	23	43	22	13	8	18	37	10	10	23	30	44	52			59		35	38	58		41	48				
	39	35	42	25		24	16	13	42	47	12	11	24	36	53	55					46	53			50	51				
	44	44	57	45		32	18	33	43	52	15	20	27	38	56						55				52	56				
		56		49		37	20	39	46	54	17	24	29	41											57					
				57		47	29	40	58		19	27	31	50																
				59		59	35	47			23	30	48																	
							36	54			26	38	51																	
							49				33	42																		
							53				37	45																		
											39	48																		
											45	54																		
												55																		

Figure 6.4: Replica distribution generated by HDFS RPP in scenario one

Rack	0									1					2												3			
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15		16		17		18		19		20			
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Section	1								2	1				2	1								2				3			
Block No. of Replica	5	2	4	4	5	0	0	2	1	6	7	6	7	2	3	8	1	8	3	9	1	9	4	5	6	7	0	3	12	16
	17	14	11	13	11	13	17	14	8	10	16	10	16	9	12	19	15	18	15	19	12	18	10	11	13	14	18	19	22	24
	27	25	26	21	21	25	27	26	15	20	20	23	23	17	22	28	24	29	24	29	22	28	20	21	23	25	26	27	33	38
	30	34	31	31	32	30	32	34	28	35	37	37	35	29	33	39	36	38	36	39	33	38	30	31	32	34	39	43	47	48
	41	43	46	40	46	40	43	41	35	42	44	44	42	36	45	48	47	48	45	49	47	49	37	40	41	42	49	53	54	55
	51	50	53	50	51	52	53	52	44	54	54	55	55	45	57	59	56	58	57	59	56	58	46	50	51	52	56	57	58	59

Figure 6.5: Replica distribution generated by SRPP in scenario one

Rack	0						1								2								3							
Node	0	1	2	3	4	5	6		7		8		9		10				11				12							
Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
Block No. of Replica	1	3	7	11	2	2	0	2	1	6	0	1	8	13	4	7	9	12	3	4	7	12	0	5	10	11	23	42	51	56
	9	5	8	13	3	5	4	6	25	29	12	20	15	17	16	21	24	25	14	24	27	28								
	14	10	9	18	8	6	15	18	31	39	22	33	20	21	26	27	29	30	30	35	38	39								
	17	11	13	19	10	24	19	21	40	43	34	37	22	34	31	32	36	38	42	46	53									
	23	16	15	23	14	35	25	29	45	46	40	46	39	43	41	42	47	53												
	26	20	16	28	17	43	31	33	51	53	48	55	44	49	57															
	41	28	19	32	18	47	37	38	56		56	58	52	54																
	45	35	26	37	22	48	49	50					55	58																
	50	40	30	47	27	52	51	54					59																	
	52	44	32	48	33	55																								
	57	45	36	54	34	58																								
	59	50	44		36																									
		57	59		41																									
					49																									

Figure 6.6: Replica distribution generated by HDFS RPP in scenario two

Rack	0						1								2								3							
Node	0	1	2	3	4	5	6		7		8		9		10				11				12							
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Section	1						1								1			2	1			2	3							
Block No. of Replica	5	2	2	1	5	1	4	8	0	6	4	6	0	8	3	7	9	0	3	7	9	1	3	7	9	10	16	17	18	19
	13	15	12	12	15	13	14	16	14	16	11	18	11	18	10	17	19	2	10	17	19	4	20	21	22	23	24	25	26	27
	23	20	20	23	24	24	21	26	21	27	25	26	25	27	22	28	29	5	22	28	29	6	28	29	30	31	32	33	34	35
	32	33	30	32	30	33	34	37	31	37	34	36	31	36	35	38	39	8	35	38	39	11	36	37	38	39	40	41	42	43
	42	44	40	44	40	42	45	46	43	47	43	46	45	47	41	48	49	12	41	48	49	13	44	45	46	47	48	49	50	51
	50	52	52	53	53	50	54	56	55	57	55	56	54	57	51	58	59	14	51	58	59	15	52	53	54	55	56	57	58	59

Figure 6.7: Replica distribution generated by SRPP in scenario two

Rack	0										1												2							
Node	0		1		2		3		4		5			6			7			8			9				10			
Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
Block No. of Replica	7	8	1	5	1	3	2	9	0	6	1	6	13	3	13	14	3	7	10	4	5	6	0	2	4	8	0	2	4	9
	16	28	9	12	5	16	13	17	8	11	15	18	19	18	20	21	12	15	17	7	23	26	10	11	16	21	10	11	15	20
	29	32	14	22	19	25	38	42	12	14	22	31	32	23	26	28	20	21	27	27	29	30	25	27	40	41	23	24	30	35
	33	38	24	29	26	31	49	55	17	18	35	47	50	31	33	34	30	35	36	32	34	36	43	45	47	48	36	40	41	43
	44	46	33	39	34		58		19	22	51	57		38	40	42	37	39	51	37	41	44	49	56	59		45	46	47	48
	51	52	48	55					24	25				45	52	56	53	54		50	53	54					49	50	55	56
	53	59							28	37										57	58						57	59		
									39	42																				
									43	44																				
									46	52																				
									54	58																				

Figure 6.8: Replica distribution generated by HDFS RPP in scenario three

Rack	0										1												2							
Node	0		1		2		3		4		5			6			7			8			9				10			
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Section	1								2		1												3							
Block No. of Replica	0	5	3	4	0	5	3	4	1	2	1	7	8	2	6	9	2	6	8	1	7	9	0	3	4	5	10	11	16	17
	10	16	10	17	11	16	11	17	6	7	12	14	18	13	15	19	13	15	19	12	14	18	20	21	22	23	24	25	26	27
	20	26	23	24	23	26	20	24	8	9	21	25	29	21	27	29	22	25	28	22	27	28	28	29	30	31	32	33	34	35
	30	37	30	37	32	34	32	34	12	13	31	35	39	33	35	39	31	36	38	33	36	38	36	37	38	39	40	41	42	43
	42	44	43	44	43	45	42	45	14	15	40	46	48	41	46	49	40	47	48	41	47	49	44	45	46	47	48	49	50	51
	51	56	52	57	51	56	52	57	18	19	50	54	58	53	55	59	50	55	59	53	54	58	52	53	54	55	56	57	58	59

Figure 6.9: Replica distribution generated by SRPP in scenario three

It can be observed from the extensive simulation results that, in general, the more heterogeneous the cluster, the more skewed the replica distribution generated by HDFS RPP. That's why the replica distributions generated in scenarios one and two (Figures 6.4 and 6.6) are more unbalanced than the distribution generated in scenario three (Figure 6.8). Also, in a remarkably heterogeneous cluster, the more powerful nodes tend to be underloaded, the less powerful ones tend to be overloaded, and the more heterogeneous the cluster the more obvious this trend as shown in Figures 6.4 and 6.6. Overall, the load distribution generated by HDFS RPP is highly skewed. In contrast, SRPP generates perfectly even replica distributions in all three scenarios as shown in Figures 6.5, 6.7 and 6.9. Since the replica distribution over all slots is even, the load distribution among all heterogeneous nodes is even as well.

6.4 Summary

In this chapter, I present SRPP, a new replica placement policy for HDFS, which can remarkably improve the load balance for data-intensive applications without the running of any load balancing utility. Moreover, SRPP is specifically designed for heterogeneous cluster environments where multiple generations of hardware coexist. The effectiveness of SRPP is verified by extensive simulations.

CHAPTER SEVEN: IMPROVED SLOT REPLICA PLACEMENT POLICY FOR HADOOP DISTRIBUTED FILE SYSTEM

Load balance is a crucial issue for data-intensive computing on cloud platforms, because a load balanced cluster can significantly improve the completion time of data-intensive jobs. In this chapter, I present an improved replica placement policy for Hadoop Distributed File System (HDFS) [73], which is specifically designed for heterogeneous clusters. The HDFS replica placement policy cannot generate balanced replica assignment, and hence has to rely on a load balance utility to balance the load among cluster nodes. In contrast, my proposed policy can generate perfectly even replica assignment, and also achieve load balance among cluster nodes in any heterogeneous or homogeneous environments without the running of the load balance utility.

7.1 Introduction

Big Data has become one of the most important technology trends that is revolutionizing the way businesses are conducted. Various organizations across different industries are facing Big Data problems. The capability to analyze large volumes of data in an effective and timely manner has become a necessity for many enterprises, as it is already widely believed that innovative insights can be generated by analyzing huge datasets of interest. As a result, the innovations in data-intensive computing have been springing up rapidly. MapReduce [1] has become the most popular programming model for data-intensive computing since it was originally developed by Google. Although Google owns the proprietary implementation, Hadoop [2], an open source implementation of MapReduce, has gained wide adoption as a data management framework for data-intensive computing on commodity clusters. One of the prerequisites to data-intensive

computing is the access to computer clusters of sufficient size. However most organizations facing Big Data problems either cannot afford such a cluster or don't have the interest to purchase and maintain one. The utility computing service on cloud platforms, such as the Amazon Elastic Compute Cloud (EC2) [5], has become the perfect solution to them, with which they can provision a cluster of virtual machines on Cloud platforms only when they need it, and only pay for the time period of usage.

Load balance is one of the crucial issues that data-intensive computing on cloud platforms has to address, as it can remarkably affect the performance of data-intensive applications. The reason is load balance technology can improve job completion time, maximize cluster throughput, optimize resource usage, and avoid node overload. In this chapter, I present an improved Replica Placement Policy (RPP) for Hadoop Distributed File System (HDFS), which is the underlying file system of Hadoop. The proposed RPP has two important advantages compared with the HDFS RPP. First, the replica assignment generated by the HDFS RPP is imbalanced. Therefore HDFS has to run a load balancing utility to balance the load. In contrast, the proposed RPP can generate perfectly even replica assignment which also meets the replica placement requirements of HDFS. Second, HDFS RPP is developed for homogeneous clusters. It treats all the cluster nodes the same even in heterogeneous environments where cluster nodes have different processing capabilities. In contrast, the proposed RPP is specifically designed for heterogeneous environments, and can achieve load balance among nodes in any heterogeneous or homogeneous cluster.

The rest of the chapter is organized as follows. Section II describes the proposed policy in detail. Evaluation results are presented in Section III, and I conclude in Section IV.

7.2 The Proposed Replica Placement Policy

The proposed replica placement policy is an improved version over the SRPP, therefore it is called Improved Slot Replica Placement Policy (ISRPP). Unlike SRPP, ISRPP doesn't make the assumption that the nodes on the same rack have the same processing capability, and hence it can work in any heterogeneous environments. ISRPP employs the same strategy as SRPP to deal with the heterogeneity in processing capability of cluster nodes. Assume the cluster consists of m nodes, each of which has C_i (i = 0, 1, 2, …, m-1) units of Processing Capability (PC). (One node having C_i units of PC means it can process C_i data replicas within one unit time period.) ISRPP first assigns C_i processing slots to node i, and then assigns data replicas to the slots on nodes instead of the nodes themselves. The load balance can be achieved, if each slot is assigned the same number of data replicas. Note that ISRPP works for homogeneous clusters as well, which is a special case of heterogeneous clusters where C_i = 1 (i = 0, 1, 2, …, m-1).

Since ISRPP doesn't make any assumption about the distribution of node processing capability, the nodes on both the same rack and the different racks can have different processing capabilities. ISRPP uses partition strategy to partition all processing slots into four sections. Section 0 is used to be assigned the first replica and section 1 the second replica, which are supposed to reside on the same rack per HDFS replica placement requirements. Sections 2 and 3 are used to be assigned the third replica that is supposed to reside on a different rack. For those racks that are used to store the two replicas on the same rack, the essence of the problem is to partition the nodes on each of those racks into two sections, which satisfies the requirement that the total number of slots in one section is equal to the one in the other section. This is a classical NP-hard problem, the Number Partitioning Problem (NPP) [32].

7.2.1 Number Partitioning Problem

Given a list of positive numbers a_1 , a_2 , … , a_n , the NPP is to find a subset $A \subset \{1, 2, \ldots, n\}$, such that the discrepancy $D(A) = \left| \sum_{i \in A} a_i - \sum_{j \notin A} a_j \right|$ is minimized. The NPP is of considerable significance in theoretical computer science. It is one of the six basic NP-hard problems that is essential to the theory of NP-completeness, and the only one related to numbers. Therefore, it is often selected as the basis for NP-hardness proof of other problems dealing with numbers, such as bin packing, multiprocessor scheduling, and knapsack problems.

Since the NPP is NP-hard, there is no known algorithm that is essentially faster than the exhaustive search through all possible partitions. Although the exhaustive search can generate the optimal partition, it has a time complexity of $O(2^n)$. As a result, the exhaustive search is only feasible for small-sized problems with limited values of n. There are also polynomial algorithms that can find approximations instead of the optimal partition. The greedy heuristic is to first sort all the numbers in descending order, and then always assign the largest number remaining to the subset with the smaller sum. Although the greedy heuristic can find the optimal partition in most cases, the set differencing (or KK) heuristic [33] performs even better. At each step, the heuristic replaces the two largest numbers remaining by the absolute value of their difference, which is equivalent to assigning the two numbers to different subsets. This process continues until there is only one number left in the list, which is the discrepancy of the partition. The actual partition can be constructed by working backward through the replacing process. The Complete KK (CKK) algorithm [34] is extended from the KK heuristic. The KK heuristic always assigns the two largest numbers remaining to different subsets, while the CKK algorithm adds the other option of

assigning them to the same subset, which is done by replacing the two largest numbers remaining by their sum. The CKK searches a binary tree where the left branch of each node replaces the two largest numbers by their difference, and the right branch replaces the numbers by their sum.

In the problem of slot partition, the numbers in the list to be partitioned is the slot numbers on different nodes on the same rack. Since there are usually only very limited number of nodes on each rack, the exhaustive search is a feasible solution to the problem of slot partition. On the other hand, the proposed ISRPP is able to deal with the situation where the slot partition discrepancy is larger than zero. Therefore, a good tradeoff between time complexity and partition quality would be using the exhaustive search when the number of nodes on the same rack is small, and switching to the other polynomial algorithms when the number is large.

7.2.2 Improved Slot Replica Placement Policy

Under ISRPP, the replica placement process consists of two phases: the slot sections construction phase and the block replica assignment phase. In the slot section construction phase, all processing slots on all nodes are partitioned into four sections. Sections 0 and 1 contain about two thirds of the slots, and are used to be assigned the two replicas that are supposed to reside on the same rack per HDFS replica placement requirement. Sections 2 and 3 contain about one third of the slots, and are used to be assigned the third replica that is supposed to reside on a different rack.

Rack	0								1							2						3						
Node	0	1	2		3				4	5		6				7		8				9		10				
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Slot Section	1				0				1			0			2	1		0		2		3						
Block No. of Replica	Replica Assignment																										

Figure 7.1: Example of slot sections construction and replica assignment table

Let N be the total number of processing slots on all nodes, N_{01} be the total number of slots to be assigned to sections 0 and 1. N_{01} is calculated according to the following formula:

$$d = N / 3, \quad r = N \% 3, \quad N_{01} = \begin{cases} 2d & (r = 0 \text{ or } 1) \\ 2d + 2 & (r = 2) \end{cases} \tag{7.1}$$

, where "/" denotes integer division, and "%" integer remainder. Note that N_{01} is always even. Sections 0 and 1 contain the same number of slots, which is equal to N_{01} / 2.

ISRPP first assigns slots to sections 0, 1 and 2 from nodes on rack 0, 1, 2 and so on, until the total number of slots in sections 0 and 1 reaches N_{01} . For each rack, ISRPP first partitions the nodes into two subsets, which satisfies the requirement that the difference between the total number of slots in the two subsets is as small as possible. If the difference is zero, ISRPP will assign all the slots in one subset to section 0 and all the slots in the other subset to section 1. If one subset has more slots than the other, ISRPP will assign all the slots in the latter subset to section 0 or 1 and the same amount of slots in the former subset to the other section. The remaining slots in the former subset will be assigned to section 2. When this process reaches the end, the current rack may have more slots than what is needed to be assigned to sections 0 and 1. In this case, the extra slots will be assigned to section 2 as well. Finally, all the slots on all remaining racks will be assigned to section 3.

Figure 7.1 shows an example of the construction of the four slot sections. There are 11 nodes with different PC's across 4 racks. In this example, N = 27, d = N / 3 = 9, r = N % 3 = 0, and N_{01} = 2d = 18. Rack 0 has 8 slots, so half of them are assigned to section 0, and the other half

to section 1. Rack 1 has 7 slots, and the discrepancy of the optimal partition is one. Thus 3 slots are assigned to both section 0 and section 1. The remaining slot is assigned to section 2. Rack 2 has 6 slots, but sections 0 and 1 only need 4 more slots. Therefore 2 slots are assigned to both section 0 and section 1. The remaining 2 slots are assigned to section 2. Finally all slots on the remaining rack 3 are assigned to section 3. The following is the pseudocode of the detailed slot sections construction algorithm.

Algorithm 7.1: slot sections construction algorithm

Input:

A collection of $\boldsymbol{m}$ ($\boldsymbol{n_0}$, $\boldsymbol{n_1}$, …, $\boldsymbol{n_{m-1}}$) nodes, each of which has $\boldsymbol{r_i}$ ($\boldsymbol{i}$ = 0, 1, 2, …, $\boldsymbol{m}$ -1) slots on it.

Output:

$\boldsymbol{S_0}$: Slot section 0; $\boldsymbol{S_1}$: Slot section 1; $\boldsymbol{S_2}$: Slot section 2; $\boldsymbol{S_3}$: Slot section 3.

1. calculate $\boldsymbol{N_{01}}$ according to formula (7.1);
2. $\boldsymbol{TN} = \boldsymbol{N_{01}}$; *// Total number of slots to be assigned to* $\boldsymbol{S_0}$ *and* $\boldsymbol{S_1}$.
3. $\boldsymbol{CR}$ = 0; *// Current rack number.*
4. **while** ($\boldsymbol{TN}$ > 0) *// Assign* $\boldsymbol{TN}$ *slots to* $\boldsymbol{S_0}$ *and* $\boldsymbol{S_1}$.
5. ***Partition*(*CR*)**;

// ***Partition*(*CR*)** partitions the nodes on rack $\boldsymbol{CR}$ into two subsets $\boldsymbol{Sub_0}$ and $\boldsymbol{Sub_1}$, which satisfies the requirement that the difference in total slot numbers between the two subsets is minimum or at least as small as possible.

6. **if** (total number of slots in $Sub_0 \geq$ total number of slots in Sub_1)

7. $T =$ total number of slots in Sub_1 ;

8. **else**

9. T = total number of slots in Sub_0 ;

10. **end if**

11. **if** ($2T \leq TN$)

12. $T_0 = T_1 = T$;

13. **else**

14. $T_0 = T_1 = TN / 2$;

15. **end if**

16. **for** (each node n_i on rack CR)

17. **if** (node n_i is in Sub_0)

18. **if** ($r_i \leq T_0$)

19. assign all slots on node n_i to S_0;

20. $T_0 = T_0 - r_i$; $TN = TN - r_i$;

21. **else**

22. assign the first T_0 slots on node n_i to S_0;

23. assign the remaining slots on node n_i to S_2;

24. $TN = TN - T_0$; $T_0 = 0$;

25. **end if**

26. **else**

27. **if** ($r_i \leq T_1$)

28. assign all slots on node n_i to S_1;

29. $T_1 = T_1 - r_i$; $TN = TN - r_i$;

30. **else**

31. assign the first T_1 slots on node n_i to S_1;

32. assign the remaining slots on node n_i to S_2;

33. $TN = TN - T_1$; $T_1 = 0$;

34. **end if**

35. **end if**

36. **end for**

37. $CR = CR + 1$;

38. **end while**

39. **while** (CR < total number of racks)

40. assign all slots on CR to S_3;

41. $CR = CR + 1$;

42. **end while**

In the following block replica assignment phase, ISRPP assigns the three replicas of all data blocks to all the slots in the four slot sections constructed in the previous phase. Replicas are evenly assigned to all slots. The replica assignment is stored in a table as shown in Figure 7.1. Each column in the replica assignment table corresponds to the replica assignment of one slot. ISRPP starts with the slots in sections 0 and 1. It assigns replicas in runs. For each run, ISRPP first randomly selects slot 0 in section 0 and assigns the first replica of current block to the slot. It then

randomly selects slot 1 in section 1 that is on the same rack as slot 0, and assigns the second replica of current block to the slot. Since slot sections 0 and 1 on the same rack always have the same number of slots, ISRPP can always find a slot 1 for the slot 0. Also, slots 0 and 1 must be on two different nodes, as they are from two different sections (subsets). The replica assignment continues until the first two replicas of all blocks are assigned. The following is the pseudocode of the detailed replica assignment algorithm for slot sections 0 and 1.

Algorithm 7.2: replica assignment algorithm for sections 0 and 1

Input:

S_0 [0, 1, …, N_0 -1]: Slot section 0, which has N_0 slots in it.

S_1 [0, 1, …, N_1 -1]: Slot section 1, which has N_1 slots in it.

n data blocks to be assigned, which are numbered 0 through (n - 1).

Output:

Replica assignment for S_0 and S_1 .

```
1.    CB = 0;              // Current block number.
2.    while (CB < n)
3.        SP0 = S0 ; SP1 = S1 ;   // SP0 and SP1 are the slot pools for
                                  // random selection.
4.        while (SP0 (SP1) ≠ Ø and CB < n)    // assign one replica to each
                                              // slot in SP0 and SP1 .
5.            randomly select slot RS0 from SP0 ;
```

6. assign the first replica of block ***CB*** to ***RS₀*** ;

7. remove ***RS₀*** from ***SP₀*** ;

8. randomly select slot ***RS₁*** from ***SP₁*** , which is on the same rack as slot ***RS₀*** ;

9. assign the second replica of block ***CB*** to ***RS₁*** ;

10. remove ***RS₁*** from ***SP₁*** ;

11. ***CB*** = ***CB*** + 1;

12. **end while**

13. **end while**

After it finishes the replica assignment to the slot sections 0 and 1, ISRPP proceeds to sections 2 and 3. Before it assigns replicas to sections 2 and 3, ISRPP first needs to determine the number of replicas to be assigned to sections 2 and 3, respectively. Under ISRPP, the replica assignment will be as even as possible. If the total number of blocks is a multiple of the total number of slots in sections 2 and 3, the number of replicas assigned to each slot will be the same. Otherwise, ISRPP will have to assign one more replica to some of the slots, and it will select the slots in section 3 first as the replica assignment to slots in section 3 is simpler than the one to slots in section 2. Let n_2 and n_3 be the total number of replicas to be assigned to the slots in sections 2 and 3, respectively. n_2 and n_3 are calculated according to the following formulas:

$$d = n / (N_2 + N_3) \quad , \quad r = n \% (N_2 + N_3)$$

$$n_2 = \begin{cases} d \times N_2 & (r \le N_3) \\ d \times N_2 + r - N_3 & (r > N_3) \end{cases} \tag{7.2}$$

$$n_3 = n - n_2 \tag{7.3}$$

, where $\boldsymbol{n}$ is the total number of blocks, $\boldsymbol{N_2}$ the number of slots in section 2, $\boldsymbol{N_3}$ the number of slots in section 3.

ISRPP continues the replica assignment with slot section 2 first, because not all replicas can be assigned to any slot in section 2. Section 2 slots are on the same rack as the first two replicas of certain data blocks, thus the third replica of those data blocks cannot be assigned to those slots per HDFS replica placement rules. Therefore, for each slot, ISRPP needs to find one unassigned block of which the first two replicas are not on the same rack as the slot, and assign the third replica of the block to the slot. The process continues until $\boldsymbol{n_2}$ replicas are assigned. Finally, ISRPP assigns all remaining replicas to section 3 slots. For each section 3 slot, ISRPP only needs to find one unassigned block, and assign the third replica of the block to the slot. The process continues until $\boldsymbol{n_3}$ replicas are assigned. The following is the pseudocode of the detailed replica assignment algorithm for slot sections 2 and 3.

Algorithm 7.3: replica assignment algorithm for sections 2 and 3

Input:

$\boldsymbol{S_2}$ [0, 1, …, $\boldsymbol{N_2}$ -1]: Slot section 2, which has $\boldsymbol{N_2}$ slots in it.

$\boldsymbol{S_3}$ [0, 1, …, $\boldsymbol{N_3}$ -1]: Slot section 3, which has $\boldsymbol{N_3}$ slots in it.

$\boldsymbol{n}$ data blocks to be assigned, which are numbered 0 through ($\boldsymbol{n}$ - 1).

Output:

Replica assignment for S_2 and S_3 .

1. calculate n_2 and n_3 according to formulas (7.2) and (7.3), respectively;
2. **for** (i = 0 **to** n -1)
3. A[i] = **false**; *// A[i] indicates whether the third replica of block i*
   // has been assigned.
4. **end for**
5. P = 0; *// Current slot pointer.*
6. CB = 0; *// Current block number.*
7. **for** (i = 1 **to** n_2) *// Distribute n_2 replicas to slots in S_2 .*
8. $CS = S_2$ [P]; *// Retrieve the next slot in S_2 as current slot.*
9. $P = P$ + 1;
10. **if** ($P == N_2$) *// Reset current slot pointer when it reaches the end.*
11. P = 0;
12. **end if**
13. **while** (the first two replicas of block CB are on the same rack as CS
    or A[CB] == **true**)
14. $CB = CB$ + 1;
15. **if** ($CB == n$) *// Reset current block number*
16. CB = 0; *// when it reaches the end.*
17. **end if**

```
18.         end while
19.         assign the third replica of block CB to CS;
20.         A[CB] = true;
21.         CB = CB + 1;
22.         if (CB == n)
23.             CB = 0;
24.         end if
25.     end for
26.     P = 0;      // Reset current slot pointer.
27.     CB = 0;                 // Reset current block number.
28.     for (i = 1 to n3)       // Assign the remaining n3 replicas to slots in S3 .
29.         CS = S3 [P];        // Retrieve the next slot in S3 as current slot.
30.         P = P + 1;
31.         if (P == N3)        // Reset current slot pointer when it reaches the end.
32.             P = 0;
33.         end if
34.         while (A[CB] == true)   // Find the next unassigned block.
35.             CB = CB + 1;
36.         end while
37.         assign the third replica of block CB to CS;
38.         CB = CB + 1;
39.     end for
```

7.3 Evaluation

Extensive simulation was conducted to examine the replica assignment generated by ISRPP. The simulation results confirm that ISRPP can generate perfectly even replica assignments. Due to page limitation, only the results of example simulation scenarios that are in reduced scale can be presented here. The settings of two example scenarios are shown in Table 7.1. Scenario one represents a heterogeneous cluster where there are four generations of hardware, which have 1, 2, 3 and 4 units of PC's respectively. The cluster in scenario two is much more heterogeneous, as it consists of nodes with 1, 2, 4 and 8 units of PC's.

Table 7.1: Simulation settings

Scenario	**One**	**Two**
Total Number of Blocks	100	100
Duplication Factor	3	3
Total Number of Replicas	300	300
Number of Racks	4	4
Total Number of Nodes	23	16
Number of Nodes on Each Rack	{7, 5, 5, 6}	{4, 5, 3, 4}
Processing Capability of Nodes on Each Rack (Units)	{1, 1, 1, 2, 2, 3, 4} {2, 3, 3, 4, 4} {2, 2, 3, 3, 4} {1, 2, 2, 3, 4, 4}	{1, 2, 4, 8} {1, 2, 2, 4, 8} {1, 4, 8} {1, 2, 4, 8}
Total Number of Slots	60	60
Average Number of Replicas on One Slot	5	5

Rack	0														1													
Node	0	1	2	3		4		5			6				7		8		9		10				11			
Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
Number of Replicas	14	13	15	5	5	4	3	6	6	6	3	2	2	2	5	5	6	6	5	5	4	4	5	5	5	4	4	4	4	4
Rack	2														3													
Node	12		13		14			15			16				17	18		19		20			21				22			
Slot	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*	*45*	*46*	*47*	*48*	*49*	*50*	*51*	*52*	*53*	*54*	*55*	*56*	*57*	*58*	*59*
Number of Replicas	6	5	3	3	5	5	4	4	4	4	5	5	4	4	16	8	8	7	6	4	4	3	3	3	3	3	3	3	2	2

Figure 7.2: Number of replicas on each slot under HDFS RPP in scenario one

Rack	0														1													
Node	0	1	2	3		4		5			6				7		8		9		10				11			
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Section	1							0							1								0							
Block No. of Replica	11	2	15	17	12	19	13	11	12	13	2	17	19	15	1	6	16	9	4	3	10	0	4	9	3	10	6	16	1	0
	35	36	22	38	34	23	37	38	37	23	36	34	35	22	30	20	31	24	25	26	21	28	28	21	26	25	31	20	30	24
	42	54	41	50	46	43	56	54	46	50	43	42	56	41	44	45	52	51	40	59	57	58	44	40	59	45	57	52	51	58
	68	64	67	75	74	71	63	75	71	74	67	63	64	68	66	62	72	69	77	73	76	61	61	72	66	76	77	73	69	62
	93	97	89	82	81	95	92	97	81	93	92	95	89	82	84	90	87	91	99	83	85	80	80	87	83	99	84	91	85	90
Rack	2														3													
Node	12		13		14			15			16				17	18		19		20			21				22			
Slot	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Section	1					2		0					2		3															
Block No. of Replica	5	14	18	8	7	0	1	18	14	8	5	7	2	3	5	7	8	14	18	25	26	27	28	29	30	31	32	33	34	35
	27	33	39	29	32	4	6	29	32	39	33	27	9	10	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
	49	55	53	47	48	11	12	53	47	49	48	55	13	15	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
	65	79	78	60	70	16	17	70	65	78	79	60	19	20	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
	94	98	96	88	86	21	22	94	86	98	88	96	23	24	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99

Figure 7.3: Replica assignment generated by ISRPP in scenario one

Rack	0															1														
Node	0	1		2				3								4	5		6		7				8						
Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
Number of Replicas	19	8	7	6	6	6	5	3	3	3	3	3	3	3	3	18	12	12	7	7	5	5	4	4	3	2	2	2	2	2
Rack	1			2												3														
Node	8		9	10				11								12	13		14				15							
Slot	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*	*45*	*46*	*47*	*48*	*49*	*50*	*51*	*52*	*53*	*54*	*55*	*56*	*57*	*58*	*59*
Number of Replicas	2	2	17	4	4	3	3	3	3	3	3	3	3	2	2	22	10	10	5	4	4	4	2	2	2	2	2	2	2	2

Figure 7.4: Number of replicas on each slot under HDFS RPP in scenario two

Rack	0															1														
Node	0	1		2				3								4	5		6		7				8						
Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Section	1							0							2	0	1								0						
Block No. of Replica	7	1	9	13	6	5	2	6	2	7	9	5	13	1	0	15	0	14	8	16	10	15	18	11	18	16	0	11	8	10
	35	33	36	38	30	20	29	30	29	35	36	38	33	20	8	34	34	24	37	28	21	25	22	23	23	22	25	28	21	37
	52	40	46	45	54	50	47	54	40	45	50	47	52	46	14	51	56	59	41	51	55	58	49	57	56	58	49	59	41	55
	79	76	71	60	67	74	78	60	67	71	79	76	78	74	22	77	69	77	64	70	68	72	63	73	70	72	64	68	69	63
	82	86	89	84	90	80	97	82	86	90	84	97	89	80	31	83	93	85	88	87	83	94	99	95	95	88	93	99	87	85
Rack	1			2												3														
Node	8		9	10				11								12	13		14				15							
Slot	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Section	0	2	1					0					2			3														
Block No. of Replica	14	1	4	12	19	17	3	12	17	3	19	4	2	5	6	3	4	7	12	15	16	19	23	24	25	27	36	37	38	39
	24	9	31	27	26	32	39	26	27	31	39	32	10	11	13	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
	57	17	44	48	43	53	42	53	44	48	42	43	18	20	21	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
	73	26	65	62	61	66	75	65	75	61	66	62	28	29	30	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84
	94	32	91	96	81	98	92	81	98	96	92	91	33	34	35	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99

Figure 7.5: Replica assignment generated by ISRPP in scenario two

HDFS RPP generates imbalanced replica assignments in both simulation scenarios. Figures 7.2 and 7.4 show the number of replicas assigned to each slot under HDFS RPP. Note that, unlike ISRPP, HDFS RPP treats nodes with different PC's the same, and assigns replicas to nodes instead of slots. The "*slot*" rows in Figures 7.2 and 7.4 are just for comparison purpose. To compare HDFS RPP with ISRPP, replicas assigned to one node by HDFS RPP are evenly distributed across all the slots on the node as much as possible. For example, in Figure 7.2, node 6 is assigned 9 replicas by the HDFS RPP, and after 2 replicas are distributed to each of slots 10 through 13, the remaining one is distributed to slot 10. Under HDFS RPP, the more powerful nodes tend to get seriously underloaded, and the less powerful nodes tend to get seriously overloaded. And the more heterogeneous the cluster, the more obvious this trend. As an example, in Figure 7.4, the least powerful node 12 has 22 replicas on its single slot, while the most powerful node 15 has only 2 replicas on each of its 8 slots. In contrast, as shown in Figures 7.3 and 7.5, ISRPP generates perfectly even replica assignments, which also complies with the HDFS replica placement rules. Since the replica assignment is perfectly even across all slots, the load balance among all cluster nodes with different PC's is achieved as well.

7.4 Summary

In this chapter, I analyze the mathematical model behind the node partition problem for replica placement of HDFS. I present ISRPP, an improved replica placement policy for HDFS. The simulation results indicate that ISRPP can generate replica assignments that are not only perfectly even but also satisfies the replica placement requirements of HDFS.

CHAPTER EIGHT: AN IMPROVED STRAGGLER IDENTIFICATION SCHEME FOR DATA-INTENSIVE COMPUTING ON CLOUD PLATFORMS

One of the challenges faced by data-intensive computing is the problem of stragglers, which can significantly increase the job completion time. Various proactive and reactive straggler mitigation techniques have been developed to address the problem. The straggler identification scheme is a crucial part of the straggler mitigation techniques, as only when stragglers are detected not only correctly but also early enough, the improvement in job completion time can make a real difference. Although the classical standard deviation method is a widely adopted straggler identification scheme, it is not an ideal solution due to certain inherent limitations. In this chapter, I present an improved straggler identification scheme [74] that is based on Tukey's method, another statistical method for outlier detection, which is more suitable for the identification of stragglers for two reasons. First, it is robust to extreme observations from stragglers. Second, it can identify stragglers and, more importantly, start speculative execution earlier than the standard deviation method. Extensive simulation results confirm that Tukey's method can remarkably outperform the standard deviation method.

8.1 Introduction

Big Data has become a reality of today's world. As Terabyte and Petabyte datasets rapidly become the new norm, the capability to perform data-intensive computing is already a necessity, instead of a luxury or curiosity, for various organizations. Due to the scale of the data-intensive jobs, the only feasible way to solve them in a reasonable time period is to partition them into

independent small tasks which can be processed in parallel across a large number of cluster nodes. Cloud computing dramatically lowers the barrier to data-intensive computing by allowing the users to provision clusters of virtual machines when they have the data-intensive processing needs and pay only as much as what is needed to solve their problem.

One of the crucial issues data-intensive computing must deal with is the problem of stragglers. A straggler in parallel processing is a node that is still working, but performing much less efficiently than normal nodes. On cloud platforms, stragglers are the norm instead of exception, especially in the case of data-intensive computing jobs, which can run on clusters easily consisting of hundreds of nodes. There are two major reasons. First, data centers use commonly available commodity hardware instead of expensive, highly reliable hardware. As a result, the chance of part failure is high, and nearly inevitable for large clusters. Part failure may just degrade the node performance instead of causing complete node failure, which would generate stragglers performing poorly. For example, a node with a faulty hard drive may experience frequent read errors that are correctable. As a result, the node can still work but with very slow disk read speed. Second, virtualization technology is widely used by service providers of utility computing, such as the Amazon Elastic Compute Cloud (EC2) [5], to provide an abstraction of the underlying hardware, which can simplify resource usage, improve resource utilization, and isolate users for security purposes. Although virtualization technology can effectively isolate the CPU and memory usage, both disk and network bandwidth are still shared among virtual machines (nodes) residing on the same physical host, which can cause notable heterogeneity in node performance as the resource contention is likely to be of different intensities on different hosts. In this case, all virtual machines on a seriously overloaded physical host will become straggler.

Since stragglers can seriously delay the completion of parallel processing jobs, most data-intensive computing frameworks, such as MapReduce [1] or Hadoop [2], employ a mechanism called speculative execution to deal with the straggler issue, which runs a speculative copy of a straggler's task on another normal node. Since, in most cases, the speculative copy completes much earlier than the original task running on the straggler, speculative execution can dramatically reduce the job completion time. For speculative execution, the most crucial part is straggler identification, because to significantly improve job completion time, the stragglers must be detected correctly, and more challengingly, early enough. One commonly used straggler identification scheme is the classical Standard Deviation (SD) method, which constantly monitors the performance of all processing nodes, and marks any node as straggler if its performance is significantly lower than the sample mean of the task completion times of all running nodes. In spite of its wide adoption, the SD method has certain inherent limitations, which makes it not an ideal solution to the problem of straggler identification. In this chapter, I suggest the application of Tukey's method, another statistical method for outlier detection, which is arguably more suitable than the SD method. Extensive simulation results confirm that Tukey's method can remarkably outperform the SD method with respect to the prompt identification of stragglers and the early start of speculative execution.

The rest of the chapter is organized as follows. Section II describes both the SD method and Tukey's method in detail. Evaluation results are presented in Section III, and I conclude in Section IV.

8.2 Tukey's Method

The statistical model behind straggler identification in parallel processing is outlier detection. In statistics, outliers are unusually large or small values in a data set. Informal test is one major category of outlier detection methods, which generates certain criterion interval for outlier detection. The observed values beyond this interval are identified as possible or probable outliers. One commonly used informal test is the classical SD method. The outlier detection interval generated by this method is $[\overline{x} - m \times SD,\ \overline{x} + m \times SD]$, where $\overline{x}$ is the sample mean, SD is the sample standard deviation, and *m* is a positive integer. The probabilities that any observation is within $m \times SD$ of the sample mean can be calculated according to the Chebyshev inequality. For example, at least 75% of the observations are within 2 standard deviations, at least 89% within 3 standard deviations, and at least 94% within 4 standard deviations, which is true for any observation from any distribution. In practice, *m* is usually set to either 2 or 3, and any observation beyond 2 or 3 standard deviations of the sample mean is marked as an outlier. Although it is widely used in parallel processing to identify straggler nodes, the SD method has certain inherent drawbacks. First, the method depends on sample mean and standard deviation, which are themselves subject to extreme observations, therefore the results can be negatively affected by extreme observations. Second, since the method has to calculate the sample mean and standard deviation, the identification interval can only be generated after all observations have been obtained. This drawback can seriously delay the straggler identification and speculative execution as well, and hence significantly prolong the completion of parallel processing jobs.

Tukey's method is another well-known and effective informal test for outlier detection, which is applicable to both mound-shaped and skewed distributions. The details on Tukey's method are as follows.

- Assume there is a population of observations sorted in ascending order, and Q_1 is the lower quartile (25th percentile), Q_3 is the upper quartile (75th percentile).
- The Inter Quartile Range (IQR) is defined as the distance between the lower quartile Q_1 and the upper quartile Q_3.
- The inner fence is defined as the interval $[Q_1 - 1.5\times IQR, Q_3 + 1.5\times IQR]$, and outer fence $[Q_1 - 3\times IQR, Q_3 + 3\times IQR]$.
- Any observation either in the interval $[Q_1 - 3\times IQR, Q_1 - 1.5\times IQR]$ or in the interval $[Q_3 + 1.5\times IQR, Q_3 + 3\times IQR]$, is a possible outlier.
- Any observation beyond the outer fence is a probable outlier.

Tukey's method has two unique features that make it specifically suitable for straggler identification. First, Tukey's method only depends on the lower and upper quartiles, and hence is robust to extreme observations from stragglers, unless more than 25% of the nodes are straggler, which is unlikely in practice. In contrast, the SD method is sensitive to extreme observations. Although the SD method can mark any node with an ongoing processing time beyond certain predefined threshold as stragglers directly, the task completion times from moderate stragglers, which are less than or equal to the threshold will be included in the computation of the sample mean and standard deviation. In certain cases, the sample mean and standard deviation can be seriously skewed by the extreme observations from the moderate stragglers so that some of the

moderate stragglers cannot be identified. Second, Tukey's method can generate the identification interval, and start the speculative execution earlier than the SD method. As soon as the upper quartile observation (Q_3) is available, the identification interval can be calculated, and the speculative execution can be started as early as at the time point of ($Q_3 + 1.5 \times IQR$). While the SD method has to wait until all nodes complete their task except for those with an ongoing processing time greater than the predefined threshold. The early detection of stragglers is extremely important to the success of speculative execution, because late speculative executions on straggler tasks cannot make a significant difference in the job completion time. Therefore, Tukey's method has a crucial advantage over the SD method when it comes to the prompt detection of stragglers and the early start of speculative execution.

The following is the adaptation of Tukey's method to the problem of straggler identification.

- T_1 and T_3 are defined as the 25th percentile and 75th percentile, respectively, of task completion times of all the nodes sorted in ascending order. T_{max} is defined as the maximum completion time of all the nodes except for those with an ongoing processing time greater than the predefined threshold.
- $IQR = T_3 - T_1$
- The Lower Bound (LB) of straggler identification is ($T_3 + 1.5 \times IQR$), and the Upper Bound (UB) is ($T_3 + 3 \times IQR$).
- Any node with an ongoing task processing time between LB and UB is a potential straggler node.

- Any node with an ongoing task processing time greater than *UB* is a likely straggler node.

Under Tukey's method, the straggler identification interval can be calculated at T_3, and the speculative execution can begin at T_s ($LB \leq T_s \leq UB$). It is confirmed by extensive simulation that, in general, both *LB* and *UB* are remarkably less than T_{max}, when the task completion time follows typical heavy-tailed distributions. As a result, the moderate stragglers with an ongoing task processing time greater than or equal to T_s will be identified as straggler by Tukey's method at time T_s. However, under the SD method, these nodes will not be marked as straggler before they eventually finish. Moreover, it is not guaranteed that all of them will be identified even after the sample mean and standard deviation are calculated, because the sample mean and standard deviation can be seriously distorted by the completion times of these moderate stragglers.

8.3 Evaluation

Extensive simulation was conducted to compare both *LB* and *UB* with T_{max} under Tukey's method. It has been observed that the distribution of task completion time on cloud platforms can be modeled as heavy-tailed distributions that are one-tailed. Four typical one-tailed distributions are used in the simulation, Pareto distribution, log-normal distribution, Weibull distribution, and Burr distribution. To generate a large enough observation population, the simulation was repeated 10,000 times for each distribution, and 1000 random observations (representing 1000 different nodes) were generated from the distribution for each run of the simulation. The sample mean and standard deviation are calculated for T_1, T_3 and T_{max} based on the 10,000 observations generated

for each distribution. The *IQR*, *LB* and *UB* are calculated based on the sample mean of T_1 and T_3. Finally, the time reduction percentage is calculated as $[T_{max} - LB(UB)] / T_{max}$, which indicates how much earlier Tukey's method can begin the speculative execution compared with the SD method.

Note that the relationship between T_{max} and *LB* (*UB*) does depend on the values of the distribution parameters. Therefore the values of the parameters are set in such a way that the simulation will not generate any extreme values of T_{max}, because even the SD method will mark any node with an ongoing processing time beyond certain predefined threshold as straggler directly, and those extreme values will not be included when the SD method calculates the sample mean and standard deviation for straggler identification. In the simulation, only relatively large values of task completion time bounded by T_{max} are generated, which represent moderate stragglers with task completion time less than or equal to T_{max} that the SD method can only identify based on the sample mean and standard deviation.

8.3.1 Pareto Distribution

Pareto distribution is a heavy-tailed distribution with one polynomially decaying tail. It has been used to model various real-life phenomena, such as household incomes, city populations, and lifetime of manufactured items. The Cumulative Distribution Function (CDF) of a Pareto random variable *X* is

$$F_X(x) = \begin{cases} 1 - (\frac{x_m}{x})^{\alpha} & x \geq x_m \\ 0 & x < x_m \end{cases} \tag{8.1}$$

, and the Probability Density Function (PDF) is

$$f_X(x) = \begin{cases} \dfrac{\alpha x_m^{\alpha}}{x^{\alpha+1}} & x \geq x_m \\ 0 & x < x_m \end{cases} \tag{8.2}$$

, where x_m ($x_m > 0$) is the scale parameter and α ($\alpha > 0$) the shape parameter. The scale parameter x_m is the minimum possible value of variable X, which is set to 1 (time unit) in the simulation.

The CDF and PDF with different values of parameter α used in the simulation are plotted in Figure 8.1 and Figure 8.2, respectively. It can be observed that, as $x \rightarrow +\infty$, the CDF approaches 100% and the PDF approaches the X axis asymptotically due to the heavy-tailed feature of the distribution, and the distribution becomes less heavy-tailed while the value of α increases. As shown in Table 8.1, both LB and UB are remarkably less than T_{max} in all simulation scenarios.

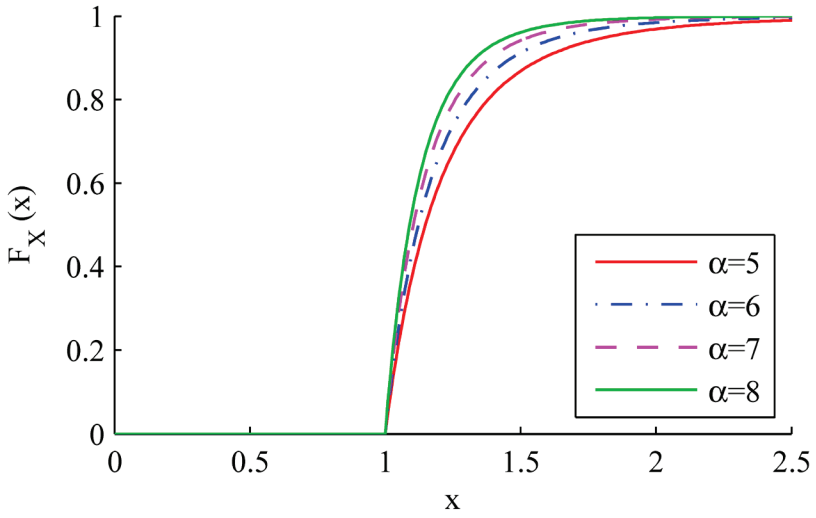

Figure 8.1: CDF of Pareto distribution

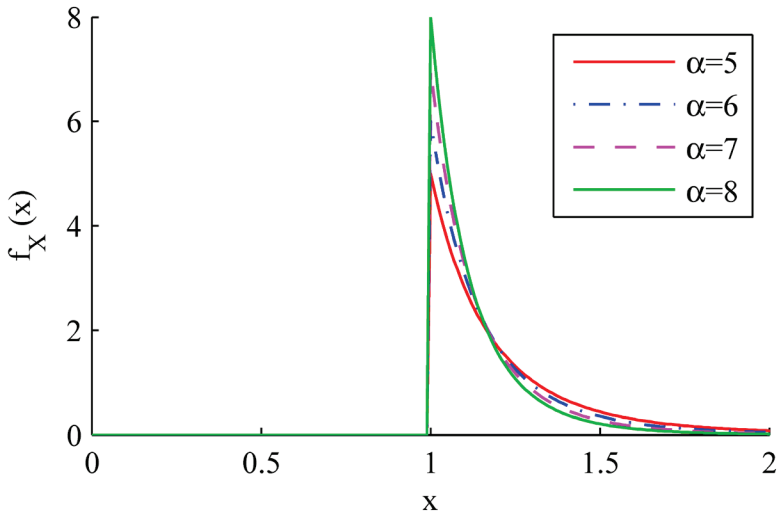

Figure 8.2: PDF of Pareto distribution

Table 8.1: Simulation results of Pareto distribution

	α = 5	**α = 6**	**α = 7**	**α = 8**
Mean of T_1	1.059	1.049	1.042	1.037
Standard Deviation of T_1	0.004	0.003	0.003	0.002
Mean of T_3	1.319	1.260	1.219	1.189
Standard Deviation of T_3	0.014	0.011	0.009	0.008
Mean of T_{max}	4.641	3.578	2.974	2.578
Standard Deviation of T_{max}	1.437	0.913	0.615	0.459
Lower Bound (*LB*)	1.709	1.576	1.484	1.418
Time Reduction of *LB*	63.2%	56.0%	50.1%	45.0%
Upper Bound (*UB*)	2.099	1.892	1.750	1.646
Time Reduction of *UB*	54.8%	47.1%	41.2%	36.2%

8.3.2 Log-normal Distribution

Log-normal distribution is another one-tailed distribution that is commonly used in the modeling of the life spans of units in mechanical systems. The CDF of a log-normal random variable X is

$$F_X(x) = \Phi\left(\frac{\ln x - \mu}{\sigma}\right) \quad (x > 0) \tag{8.3}$$

(Φ is the CDF of the standard normal distribution), and the PDF is

$$f_X(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}} \quad (x > 0) \tag{8.4}$$

, where μ is the location parameter and σ ($\sigma > 0$) the scale parameter. The location parameter μ is set to 0 in the simulation. The CDF and PDF with different values of parameter σ used in the simulation are plotted in Figure 8.3 and Figure 8.4, respectively. It can be observed that the distribution becomes more heavy-tailed while the value of σ increases. As shown in Table 8.2, both *LB* and *UB* are less than T_{max} in all simulation scenarios.

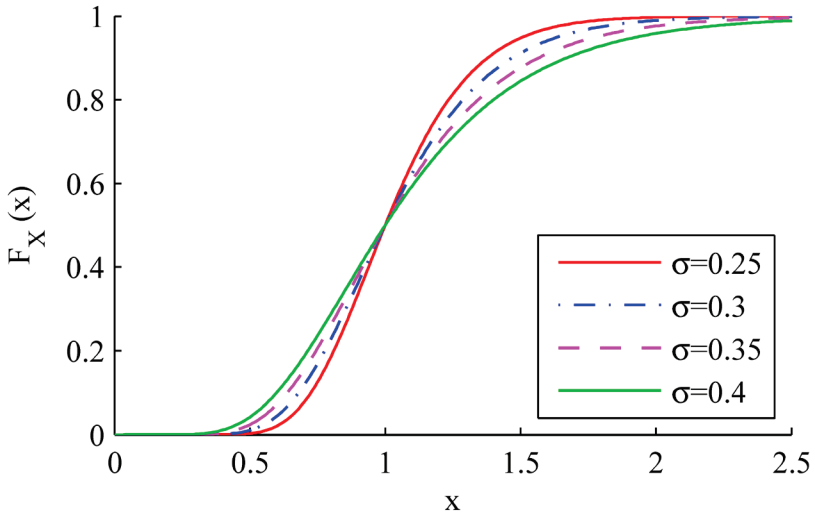

Figure 8.3: CDF of log-normal distribution

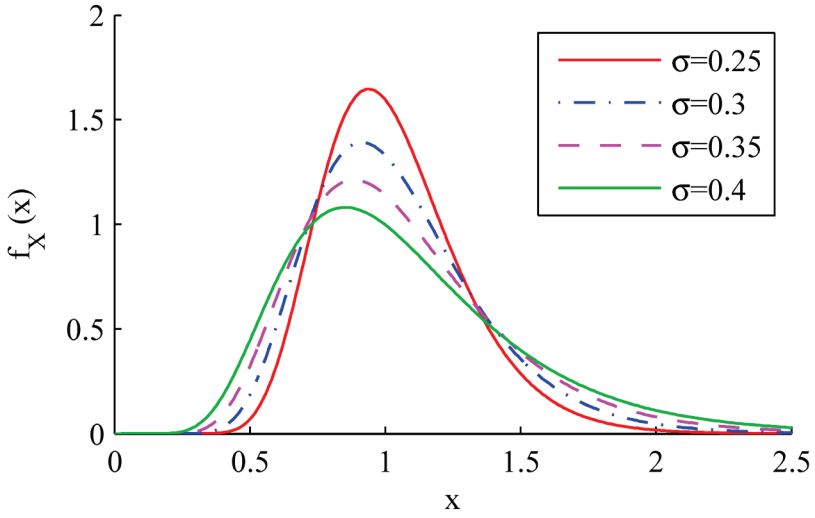

Figure 8.4: PDF of log-normal distribution

Table 8.2: Simulation results of log-normal distribution

	σ = 0.25	**σ = 0.3**	**σ = 0.35**	**σ = 0.4**
Mean of T_1	0.845	0.817	0.790	0.763
Standard Deviation of T_1	0.009	0.011	0.012	0.013
Mean of T_3	1.183	1.224	1.265	1.309
Standard Deviation of T_3	0.013	0.016	0.019	0.022
Mean of T_{max}	2.259	2.662	3.136	3.694
Standard Deviation of T_{max}	0.210	0.297	0.415	0.542
Lower Bound (*LB*)	1.691	1.834	1.979	2.127
Time Reduction of *LB*	25.2%	31.1%	36.9%	42.4%
Upper Bound (*UB*)	2.198	2.445	2.693	2.946
Time Reduction of *UB*	2.7%	8.2%	14.1%	20.2%

8.3.3 Weibull Distribution

Weibull distribution is widely used in reliability engineering to model the lifetime distributions. The CDF of a Weibull random variable X is

$$F_X(x) = \begin{cases} 1 - e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{8.5}$$

, and the PDF is

$$f_X(x) = \begin{cases} \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{8.6}$$

, where λ ($\lambda > 0$) is the scale parameter and k ($k > 0$) the shape parameter. Since the scale parameter only stretches or shrinks the distribution, λ is set to 1 in the simulation.

On the other hand, the shape parameter is more important, as it affects the shape of the distribution. In the simulation, variable X represents "finish time" of processing nodes. If $k < 1$, the "finish rate" decreases over time, which means the nodes are less and less likely to complete their tasks as time goes by. If $k = 1$, the "finish rate" is constant over time, which may suggest that it is mostly determined by certain random factors whether a node can complete its task or not. In this case, the Weibull distribution reduces to an exponential distribution, which is not heavy-tailed. If $k > 1$, the "finish rate" increases over time, which can happen when defective nodes fail early and the remaining nodes are more likely to complete their task.

The CDF and PDF with different values of parameter k used in the simulation are plotted in Figure 8.5 and Figure 8.6, respectively. It can be observed that the distribution becomes less heavy-tailed while the value of k increases. As shown in Table 8.3, in the case of Weibull distribution, although UB is greater than T_{max} , LB is still less than T_{max} , which can serve as the threshold for straggler identification under Tukey's method.

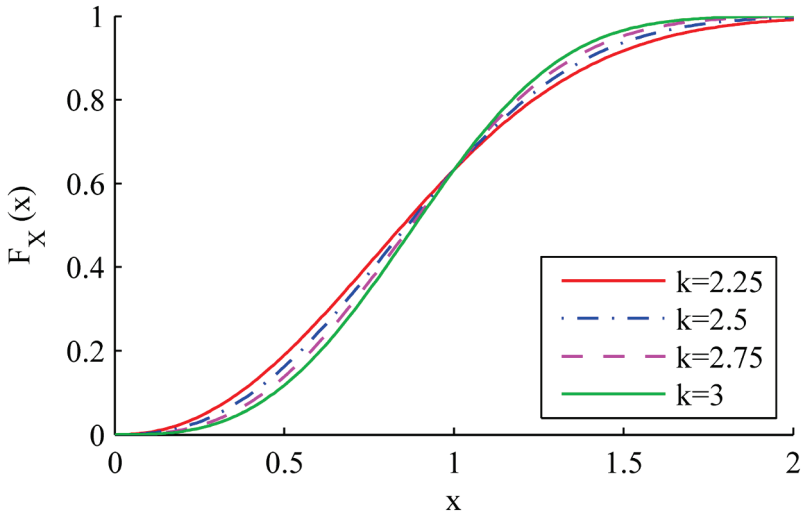

Figure 8.5: CDF of Weibull distribution

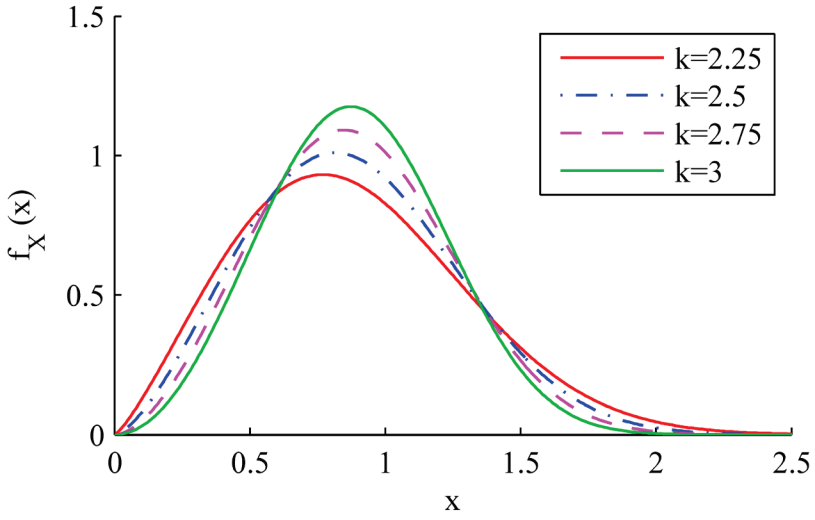

Figure 8.6: PDF of Weibull distribution

Table 8.3: Simulation results of Weibull distribution

	$k = 2.25$	$k = 2.5$	$k = 2.75$	$k = 3$
Mean of T_1	0.574	0.607	0.635	0.660
Standard Deviation of T_1	0.016	0.015	0.015	0.014
Mean of T_3	1.156	1.139	1.125	1.114
Standard Deviation of T_3	0.020	0.018	0.016	0.015
Mean of T_{max}	2.436	2.228	2.071	1.950
Standard Deviation of T_{max}	0.181	0.149	0.124	0.107
Lower Bound (*LB*)	2.028	1.937	1.860	1.796
Time Reduction of *LB*	16.8%	13.1%	10.2%	7.9%
Upper Bound (*UB*)	2.900	2.735	2.596	2.478
Time Reduction of *UB*	-19.0%	-22.8%	-25.3%	-27.0%

8.3.4 Burr Distribution

Burr distribution is another one-tailed distribution with a wide variety of shapes. Burr distribution is more complicated than the previous distributions as it has two shape parameters. It is often used to model crop prices, household incomes, and option market prices. The CDF of a Burr random variable X is

$$F_X(x) = 1 - (1 + x^c)^{-k} \quad (x > 0) \tag{8.7}$$

, and the PDF is

$$f_X(x) = ck\frac{x^{c-1}}{(1+x^c)^{k+1}} \quad (x>0) \tag{8.8}$$

, where c ($c > 0$) and k ($k > 0$) are the shape parameters. The CDF and PDF with different values of parameters c and k used in the simulation are plotted in Figure 8.7 and Figure 8.8, respectively. As shown in Table 8.4, both LB and UB are less than T_{max} in all simulation scenarios.

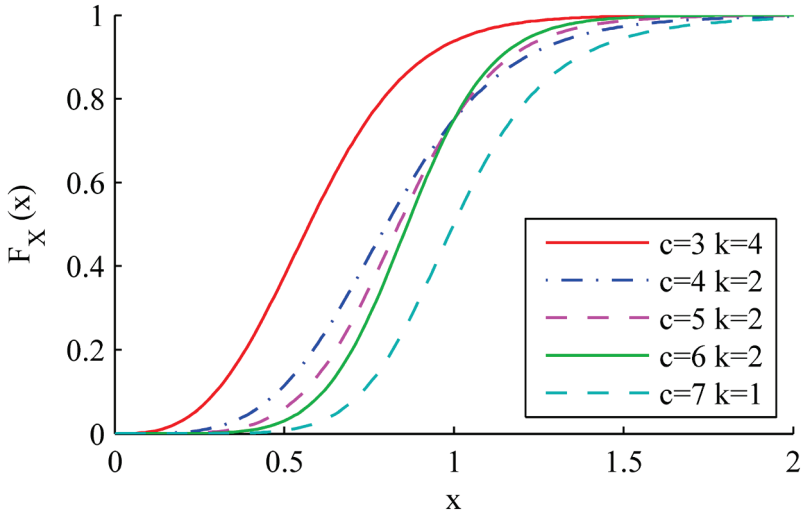

Figure 8.7: CDF of Burr distribution

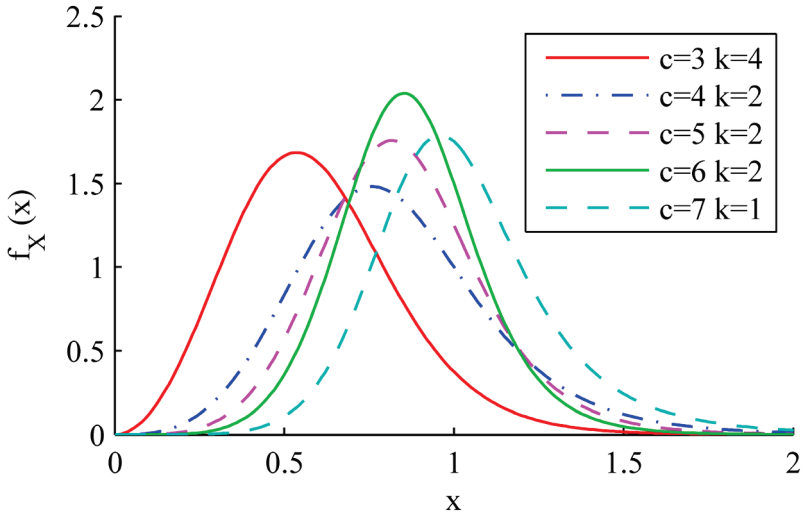

Figure 8.8: PDF of Burr distribution

Table 8.4: Simulation results of Burr distribution

	$c=3$ $k=4$	$c=4$ $k=2$	$c=5$ $k=2$	$c=6$ $k=2$	$c=7$ $k=1$
Mean of T_1	0.420	0.627	0.688	0.732	0.854
Standard Deviation of T_1	0.009	0.011	0.009	0.008	0.009
Mean of T_3	0.745	0.999	1.000	1.000	1.170
Standard Deviation of T_3	0.012	0.014	0.011	0.009	0.012
Mean of T_{max}	1.772	2.567	2.118	1.870	2.971
Standard Deviation of T_{max}	0.236	0.466	0.304	0.223	0.637
Lower Bound (*LB*)	1.232	1.558	1.467	1.401	1.642
Time Reduction of *LB*	30.5%	39.3%	30.8%	25.1%	44.7%
Upper Bound (*UB*)	1.718	2.116	1.934	1.802	2.115
Time Reduction of *UB*	3.1%	17.5%	8.7%	3.7%	28.8%

To conclude, *LB* is considerably less than T_{max} in all simulation scenarios, and *UB* is less than T_{max} in most scenarios. Therefore, under Tukey's method, the threshold for marking stragglers can be set between *LB* and *UB* based on the tradeoff between job completion time and computing resources usage made by the users. It is confirmed by the simulation that Tukey's method can start the speculative execution much earlier than the SD method in general, which can in turn reduce the overall job completion time.

8.4 Summary

In this chapter, I analyze the limitations of the commonly used SD method for straggler detection. I propose Tukey's method as a better solution, and adapt Tukey's method to an improved straggler identification scheme. Extensive simulation was conducted to evaluate the performance of Tukey's method compared with the SD method. The simulation results indicate that overall Tukey's method can identify stragglers and start the speculative execution much earlier than the SD method.

CHAPTER NINE: CONCLUSION AND FUTURE WORK

In this dissertation, I address the performance issue of data-intensive computing on Cloud platforms from three different aspects: task assignment, replica placement, and straggler identification. From the first aspect, I discuss the problems of Hadoop task assignment scheme and present an improved task assignment scheme, ECT, which can balance the processing load of map task of data-intensive applications among all cluster nodes. Extensive simulation results indicate that ECT can significantly outperform the Hadoop scheme with respect to map phase completion time. Moreover, ECT employs much less remote processing than the Hadoop scheme. Therefore, running under ECT, the data-intensive applications are much less vulnerable to both network congestion and disk contention, which can be frequently experienced at data centers.

From the second aspect, I discuss the drawback of HDFS replica placement policy and present PRPP, a new replica placement policy, which can generate perfectly even replica assignment that meets all HDFS replica placement requirements, and hence completely eliminate the need to run any load balancing utility. In contrast, the HDFS replica placement policy can only generate unbalanced replica assignment, and has to rely on load balancing utility to balance the load among cluster nodes at the price of extra system resources and running time. I then present SRPP, an improved replica placement policy. While PRPP only works in homogeneous clusters, SRPP works in both homogeneous clusters and heterogeneous ones where nodes on the same rack have the same processing capability. Finally, I discuss the mathematical model related to my research work on replica placement, and present ISRPP, a more advanced and general solution that works in any homogeneous or heterogeneous cluster.

From the third aspect, I analyze the limitations of the commonly used SD method for straggler detection. I propose Tukey's method as a better solution, and explain the two unique features of Tukey's method that make it more suitable for straggler detection than the SD method. I then present an improved straggler identification scheme based on Tukey's method. I also present the results of extensive simulation, which confirm that overall Tukey's method can identify stragglers and start the speculative execution much earlier than the SD method on Cloud platforms.

As to the first aspect task assignment, one possible direction in future work is to implement the proposed scheme in Hadoop, and evaluate the performance of the scheme in a practical environment instead of a logic simulation. With a real implementation, more performance metrics can be evaluated in a more precise way. As a result, the proposed scheme can be examined and studied in a more comprehensive way.

As to the second aspect replica placement, the correctness of the logic of the proposed replica placement policies has been proved by the extensive simulations. Therefore possible future work would be to implement the policies in HDFS to examine the overhead incurred by the policies compared with the overhead of the load balancing utility of Hadoop.

As to the third aspect straggler identification, there exist other statistical methods for outlier detection in addition to Tukey's method. Therefore one possible direction in future research work is to explore the possibility of applying these statistical methods or certain variants of them to the problem of straggler identification in data-intensive computing on Cloud platforms, and compare the performance of these methods with the one of Tukey's method or SD method with respect to the prompt identification of stragglers and the early start of speculative execution.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004.

[2] Official Apache Hadoop Website, http://hadoop.apache.org.

[3] Hadoop Wiki, http://wiki.apache.org/hadoop/PoweredBy.

[4] Wikipedia: Apache Hadoop, http://en.wikipedia.org/wiki/Hadoop.

[5] Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/.

[6] D. Jiang, B. C. Ooi, L. Shi and S. Wu, "The performance of MapReduce: an in-depth study", in *Proceedings of the VLDB Endowment*, Vol. 3, Issue 1-2, pp. 472-483, 2010.

[7] K. Lee, Y. Lee, H. Choi, Y. D. Chung and B. Moon, "Parallel data processing with MapReduce: a survey", in *ACM SIGMOD Record*, Vol. 40, Issue 4, pp. 11-20, 2011.

[8] V. Vijayalakshmi, A. Akila and S. Nagadivya, "The survey on MapReduce", in *International Journal of Engineering Science and Technology*, Vol. 4, No. 07, 2012.

[9] M. Zaharia, A. Konwinski, A. Joseph, R. Katz and I. Stoica, "Improving MapReduce performance in heterogeneous environments", in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pp. 29-42, Berkeley, CA, USA, 2008.

[10] T. Gonzalez, J.Y.-T. Leung and M. Pinedo, "Minimizing total completion time on uniform machines with deadline constraints", in *ACM Transactions on Algorithms*, Vol. 2, pp. 95-115, 2006.

[11] A. Federgruen and H. Groenevelt, "Preemptive scheduling of uniform machines by ordinary network flow techniques", in *Management Science*, Vol. 32, pp. 341–349, 1986.

[12] J. Labetoulle, E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, "Preemptive scheduling of uniform machines subject to release dates", in *Progress in Combinatorial Optimization*, W.R. Pulleyblank (ed.), pp. 245–261, Academic Press, New York, 1984.

[13] S. T. McCormick and M. Pinedo, "Scheduling n independent jobs on m uniform machines with both flow time and makespan objectives: a parametric analysis", in *ORSA Journal of Computing*, Vol. 7, pp. 63–77, 1995.

[14] A. K. Agrawala, E. G. Coffman Jr., M. R. Garey and S. K. Tripathi, "A stochastic optimization algorithm minimizing exponential flow times on uniform processors", in *IEEE Transactions on Computers*, C-33, pp. 351–356, 1984.

[15] E. G. Coffman Jr., L. Flatto, M. R. Garey and R. R. Weber, "Minimizing expected makespans on uniform processor systems", in *Advances in Applied Probability*, Vol. 19, pp. 177–201, 1987.

[16] G. Dobson, "Scheduling independent tasks on uniform processors", in *SIAM Journal of Computing*, Vol. 13, pp. 705–716, 1984.

[17] D. K. Friesen, "Tighter bounds for LPT scheduling on uniform processors", in *SIAM Journal of Computing*, Vol. 16, pp. 554–560, 1984.

[18] D. K. Friesen and M. A. Langston, "Bounds for multifit scheduling on uniform processors", in *SIAM Journal of Computing*, Vol. 12, pp. 60–70, 1983.

[19] T. Gonzalez and S. Sahni, "Preemptive scheduling of uniform processor systems", in *Journal of the Association of Computing Machinery*, Vol. 25, pp. 92–101, 1978.

[20] R. Righter, "Job scheduling to minimize expected weighted flow time on uniform processors", in *System and Control Letters*, Vol. 10, pp 211–216, 1988.

[21] S. Sahni and Y. Cho, “Scheduling independent tasks with due times on a uniform processor system”, in *Journal of the Association of Computing Machinery*, Vol. 27, pp. 550–563, 1979.

[22] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, New York, 2012.

[23] E. Lawler, J. Lenstra and A. Rinnooy Kan, “Recent developments in deterministic sequencing and scheduling”, in *Deterministic and Stochastic Scheduling*, pp. 35-73, 1982.

[24] M. Dessouky, B. Lageweg, J. Lenstra, and S. van de Velde, “Scheduling identical jobs on uniform parallel machines”, in *Statistica Neerlandica* 44, pp. 115-123, 1990.

[25] T. P. Shabeera and S. D. Madhu Kumar, “Bandwidth-aware data placement scheme for Hadoop”, in *Proceedings of the 2013 IEEE Recent Advances in Intelligent Computational Systems*, pp. 64-67, Trivandrum, Kerala, India, 2013.

[26] Q. Zhang, S. Q. Zhang, A. Leon-Garcia and R. Boutaba, “Aurora: adaptive block replication in distributed file systems”, in *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, Columbus, USA, 2015.

[27] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares and X. Qin, “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters”, in *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1-9, Atlanta, USA, 2010.

[28] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek and J. McPherson, “CoHadoop: flexible data placement and its exploitation in Hadoop”, in *Proceedings of the VLDB Endowment*, Vol. 4, Issue 9, pp. 575-585, June 2011.

[29] B. Alidaee, F. Glover, G. A. Kochenberger and C. Rego, “A new modeling and solution approach for the number partitioning problem”, in *JAMDS*, 9(2):113-121, 2005.

[30] E. G. Coman Jr. and G. S. Lueker, *Probabilistic Analysis of Packing and Partitioning Algorithms*, John Wiley & Sons, 1991.

[31] F. F. Ferreira and J. F. Fontanari, "Probabilistic analysis of the number partitioning problem", in *Journal of Physics* A, 31:3417-3428, 1998.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1997.

[33] N. Karmarkar and R. Karp, "The differencing method of set partitioning", *Technical Report UCB/CSD 82/113*, Computer Science Division, University of California, Berkeley, 1982.

[34] R. Korf, "A complete anytime algorithm for number partitioning", in *Artificial Intelligence*, 106(2):181–203, December 1998.

[35] S. Mertens, "The easiest hard problem: number partitioning", in A. G. Percus, G. Istrate and C. Moore (editors), *Computational Complexity and Statistical Physics*, pp. 125-139, Oxford University Press, New York, 2006.

[36] B. Yakir, "The differencing algorithm LDM for partitioning: a proof of a conjecture of Karmarkar and Karp", in *Mathematics of Operations Research*, 21(1):85-99, 1996.

[37] R. E. Korf, "From approximate to optimal solutions: a case study of number partitioning", in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 266-272, 1995.

[38] M. F. Argüello, T. A. Feo and O. Goldschmidt, "Randomized methods for the number partitioning problem", in *Computers & Operations Research*, Vol. 23, No. 2, pp. 103-111, February 1996.

[39] P. M. Regina Berretta and C. Cotta, “Enhancing the performance of memetic algorithms by using a matching-based recombination algorithm: results on the number partitioning problem”, in Resende MGC, J. Souza (editors), *Metaheuristics: Computer Decision-Making*, Norwell, MA: Kluwer Academic Publishers, pp. 65–90, 2004.

[40] P. C. Pop and O. Matei, “A genetic algorithm approach for the multidimensional two-way number partitioning problem”, in *Lecture Notes in Computer Science*, Vol. 7997, pp. 81–86, 2013.

[41] J. Kratica, J. Kojić and A. Savić, “Two metaheuristic approaches for solving multidimensional two-way number partitioning problem”, in *Computers & Operations Research*, Vol. 46, pp. 59-68, June 2014.

[42] N. Abe, B. Zadrozny and J. Langford, “Outlier detection by active learning”, in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 504-509, ACM Press, New York, NY, USA, 2006.

[43] B. Abraham and A. Chuang, “Outlier detection and time series modeling”, in *Technometrics* 31, 2, pp. 241-248, 1989.

[44] M. Agyemang, K. Barker and R. Alhajj, “A comprehensive survey of numeric and symbolic outlier mining techniques”, in *Intelligent Data Analysis* 10, 6, pp. 521-538, 2006.

[45] Z. Bakar, R. Mohemad, A. Ahmad and M. Deris, “A comparative study for outlier detection techniques in data mining”, in *Proceedings of 2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1-6, 2006.

[46] V. Barnett and T. Lewis, *Outliers in Statistical Data*, John Wiley & Sons, 1994.

[47] G. E. P. Box and G. C. Tiao, "Bayesian analysis of some outlier problems", in *Biometrika* 55, 1, pp. 119-129, 1968.

[48] C. Campbell and K. Bennett, "A linear programming approach to novelty detection", in *Proceedings of Advances in Neural Information Processing*, Vol. 14, Cambridge Press, 2001.

[49] N. V. Chawla, N. Japkowicz and A. Kotcz, "Editorial: special issue on learning from imbalanced data sets", in *SIGKDD Explorations* 6, 1, pp. 1-6, 2004.

[50] A. J. Fox, "Outliers in time series", in *Journal of the Royal Statistical Society, Series B (Methodological)* 34, 3, pp. 350-363, 1972.

[51] F. Grubbs, "Procedures for detecting outlying observations in samples", in *Technometrics* 11, 1, pp. 1-21, 1969.

[52] D. Hawkins, "Identification of outliers", in *Monographs on Applied Probability and Statistics*, 1980.

[53] Z. He, S. Deng, X. Xu and J. Z. Huang, "A fast greedy algorithm for outlier mining", in *Proceedings of the 10th Pacific-Asia Conference on Knowledge and Data Discovery*, pp. 567-576, 2006.

[54] V. Hodge and J. Austin, "A survey of outlier detection methodologies", in *Artificial Intelligence Review* 22, 2, pp. 85-126, 2004.

[55] M. Markou and S. Singh, "Novelty detection: a review-part 1: statistical approaches", in *Signal Processing* 83, 12, pp. 2481-2497, 2003.

[56] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: existing solutions and latest technological trends", in *Computer Networks* 51, 12, pp. 3448-3470, 2007.

[57] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, “Large-scale cluster management at Google with Borg”, in *Proceedings of the 10th European Conference on Computer Systems*, Bordeaux, France, April 2015.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: cluster computing with working sets”, in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, Boston, MA, USA, June 2010.

[59] S. P. Lloyd, "Least squares quantization in PCM", in *IEEE Transactions on Information Theory*, 28(2), pp. 129–137, 1982.

[60] W. Iba and P. Langley, “Induction of one-level decision trees”, in *Proceedings of the 9th International Conference on Machine Learning*, pp. 233–240, San Francisco, CA, USA, 1992.

[61] J. R. Quinlan, “Induction of decision trees”, in *Machine Learning*, 1(1), pp. 81-106, 1986.

[62] D. Pan and A. Moore, "X-means: extending k-means with efficient estimation of the number of clusters”, in *Proceedings of the 17th International Conference on Machine Learning*, pp. 727-734, Stanford, CA, USA, June 2000.

[63] T. Bailey and A. K. Jain, “A note on distance-weighted k-nearest neighbor rules”, in *IEEE Transactions on Systems, Man, and Cybernetics*, 8(4), pp. 311-313, 1978.

[64] G. D. Ghare and S. T. Leutenegger, “Improving speedup and response times by replicating parallel programs on a SNOW”, in *Proceedings of the 10th international conference on Job Scheduling Strategies for Parallel Processing*, pp. 264-287, New York, NY, USA, January 2005.

[65] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoic, Y. Lu, B. Saha and E. Harris, “Reining in the outliers in MapReduce clusters using Mantri”, in *Proceedings of the 9th*

USENIX conference on Operating Systems Design and Implementation, pp. 265-278, Vancouver, Canada, October 2010.

[66] Q. Chen, C. Liu and Z. Xiao, "Improving Mapreduce performance using smart speculative execution strategy", in *IEEE Transactions on Computers*, Vol. 63, Issue 4, pp. 954-967, April 2014.

[67] G. Ananthanarayanan, A. Ghodsi, S. Shenker and I. Stoica, "Effective straggler mitigation: attack of the clones", in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pp. 185-198, Lombard, IL, USA, April 2013.

[68] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz and I. Stoica, "Improving MapReduce performance in heterogeneous environments", in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pp. 29-42, San Diego, CA, USA, December 2008.

[69] W. Dai and M. Bassiouni, "An improved task assignment scheme for Hadoop running in the Clouds", in *Journal of Cloud Computing*, Springer Publishing, Vol. 2:23, pp. 1-16, December 2013.

[70] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting and A. Toncheva, "The diverse and exploding digital universe: an updated forecast of worldwide information growth through 2011", *IDC White Paper* – sponsored by EMC, 2008.

[71] W. Dai, I. Ibrahim and M. Bassiouni, "A new replica placement policy for Hadoop Distributed File System", in *Proceedings of the 2016 IEEE 2nd International Conference on High Performance and Smart Computing*, pp. 262-267, New York, NY, USA, April 2016.

[72] W. Dai, I. Ibrahim and M. Bassiouni, “Improving load balance for data-intensive computing on cloud platforms”, in *Proceedings of the 2016 IEEE International Conference on Smart Cloud*, pp. 140-145, New York, NY, USA, November 2016.

[73] W. Dai, I. Ibrahim and M. Bassiouni, “An improved replica placement policy for Hadoop Distributed File System running on Cloud platforms”, in *Proceedings of the 4th IEEE International Conference on Cyber Security and Cloud Computing*, New York, NY, USA, June 2017.

[74] W. Dai, I. Ibrahim and M. Bassiouni, “An improved straggler identification scheme for data-intensive computing on Cloud platforms”, in *Proceedings of the 4th IEEE International Conference on Cyber Security and Cloud Computing*, New York, NY, USA, June 2017.