A PROFILING AND PERFORMANCE ANALYSIS BASED SELF-TUNING SYSTEM FOR OPTIMIZATION OF HADOOP MAPREDUCE CLUSTER CONFIGURATION

By

Dili Wu

Thesis

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2013

Nashville, Tennessee

To my family and friends for their support and prayers over the years

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Aniruddha Gokhale for his encouragement, guidance and support during my time at Vanderbilt. He is not only an advisor for my research, but also the mentor for my life. Thank you for all doors you have opened and the help you have done for me.

Then I would thank Dr. Yi Cui for his time and support in reviewing this thesis. Also, I would like to thank all members in DOC group for discussing and sharing their opinions with me since the day I joined.

Last but not least, I would thank all members in Department of Electrical Engineering and Computer Science. Because of you, I could have achieved as much in my short time here.

TABLE OF CONTENTS

Page

TABLE OF FIGURES

Figure **Page**

TABLE OF TABLES

Table **Page**

CHAPTER I

INTRODUCTION

With the speedy development of Information Technology, Big Data becomes the most important issue for all companies. No matter whether they are in the high-tech industry, such as social network or cloud service, or they are in the traditional fields, such as finance or biology, companies have to deal with tons of data generated every second [1] from digital media, web authoring, market research, and so on [2], and then get the computing result and analysis report from it as fast as possible. However, when faced with this rapid grow in the amount of data, especially when the size of which varies from hundreds of gigabytes to hundreds of terabytes, classical parallel data warehousing, despite traditionally being the best technology to store, manage and analyze data for years, tends to be inefficient, inflexible and thus incredibly [3] expensive for both commercial and technical consideration nowadays.

In contrast, more and more data scientists and computer engineers are turning to work with a new parallel programming model which can not only meet the requirement of scalability, but also automatically handle resource management, fault tolerance and other issues on distributed system [4]. This parallel data processing framework is MapReduce, which was firstly stated by Google in *MapReduce: Simplified data processing on large clusters* [5]. The basic conception of MapReduce is breaking a computation task into two groups: map tasks and reduce tasks [6]. At the beginning of the Map step, the input job is divided and then assigned to several worker nodes by a master node. Then the map function running on worker nodes converts the original data to a

series of key-value pairs for future use. While in the Reduce step, the master node collects the output of map function from worker nodes, and then assigns reduce tasks with aggregating, combining, filtering or transforming these key-value pairs in some way, based on the target of input job, to form the final output [7] [8]. Even though there are still several limitations with Mapreduce, for instance, high overhead and high CPU utilization is possible to lead to a bunch of unsatisfactory performance [9], MapReduce has been proven as the most successful technology to process Big Data in a massively parallel manner [10].

In recent years, MapReduce has been improved and implemented by hundreds of research groups and companies [11], such as HBase, Nutch and Hadoop [12]. As an open-source platform developed by Yahoo in 2008, Apache Hadoop is considered to be the most well-known MapReduce framework; besides Yahoo, it is widely used by many companies such as Facebook, which has the biggest Hadoop cluster in the world, LinkedIn and IBM. In addition, as one of the major Cloud Service provider, Amazon has also set up its Elastic MapReduce web service of Hadoop cluster on top of its EC2 Cloud. It is reported that Microsoft is building Hadoop-based service, called HDInsight, on its Azure Cloud service, too.

I.1 Motivation and Our Approach

Despite the advantages of Hadoop MapReduce, it becomes the most important issue for Hadoop engineers to determine how to make full use of the cluster resources, and thus optimize its performance for both general jobs, such as sorting and searching, and other specific applications [13]. According to recent research on this topic, researchers have

shown that Hadoop cluster configuration has a significant effect on its performance, that is to say, even a tiny change to one parameter's value is very likely to make a huge difference [14] when running the same MapReduce job with the same size of data input. Moreover, because of its blackbox-like feature, it is also incredibly difficult to find a straight forward mathematical model between the cluster configuration and a specific job. On one hand, it is very inefficient to set the same configuration for all kinds of jobs; on the other hand, it is unreasonable for developers to find an optimal configuration solution for each job.

Aimed at solving this problem, we developed *Profiling and Performance Analysis Based Self-Tuning System* (PPABS) in this thesis. First of all, Profiling of MapReduce job performance and Data Mining technique is combined in this system to dynamically divide jobs into groups. Secondly, based on previous analysis and research of MapReduce parameters, Simulated Annealing, a probabilistic metaheuristic algorithm for global optimization, is imported and modified to find the optimum solution and tune the cluster configuration for the job groups we got from the first step. Thirdly, after running an incoming job with only a small part of its input data set, a Pattern Recognition technique is also used to classify this new job. Finally, the cluster configuration is updated by PPABS to match this job's features before running the whole entire job.

I.2 Thesis Organization

The rest of this thesis is organized as follows. In Chapter II we discuss and compare the related work in this field with our system. Then in Chapter III we present the design of this Profiling and Performance Analysis Based Self-Tuning System and in Chapter IV

we evaluate and analyze the results of experiments. Finally in Chapter V we discuss the future work in this field and provide conclusions derived working on this thesis.

CHAPTER II

RELATED WORK

Several research aimed at improving the performance of MapReduce cluster has been done since this technology was presented by Google [5]. At the beginning of this chapter we briefly discuss related work in this field focusing on analyzing the cluster performance. Then we also present other research efforts that show the existence of a relation between MapReduce applications' performance and their utilization pattern. At the end of this chapter, we discuss and compare our approach with the most recent researches in optimizing the Hadoop configuration [13] [14].

Research on performance analysis of MapReduce started more than six years ago. Most of the earliest works studied the scheduling and fault tolerance mechanisms of MapReduce, such as the one presented by Sun et al [15] that discussed the mathematical model of MapReduce. Another work focused on analyzing the variant effect of resource consumption of different settings for Map and Reduce slots, which was described in [16]. Based on experiments, the authors also showed that the difference of computation utilization pattern depends on different kinds of MapReduce jobs. In [17], the authors classified MapReduce applications into three categories based on their CPU and I/O utilization. Even though it was the first time to indicate the way of improving performance by categorizing jobs, the downside of this approach is also very obvious: it considered the average utilization of CPU and I/O as the only criteria to classify MapReduce jobs and thus overlooked the more important part: the overall pattern of the performance.

In fact, there are not many approaches on solving the performance problem by focusing on tuning configuration parameters. A related work that characterized resource and job utilization pattern from analyzing 10 months of MapReduce logs of Yahoo appears in [18]. This article pointed out two MapReduce applications can be considered to be of the same kind if their performance pattern is similar; moreover, it concluded that similar jobs would have similar optimal solution of cluster configuration. In one of the recent studies combining MapReduce with Machine Learning algorithms, researchers in University of Sydney modeled the relation between applications' configuration parameters and their CPU usage history in [19]. However, instead of studying performance improvement with this model, they just used it to predict unknown jobs' whole CPU usage in time clock cycles.

AROMA [20] and Starfish [21] are two recent research efforts that are related to our approach. The former system could automatically allocate the cluster resources by adapting the cluster to new jobs if their resource utilization signature matches the previously executed jobs. Whereas, such an approach does not guarantee the performance of jobs whose utilization pattern is different with any previous ones. More importantly, this paper did not present a clear way to find the optimal configuration solution even for the executed jobs. The latter one is an attempt to find the optimal cluster settings with collecting the profile information of previous jobs, simulating executing time of new jobs, and then searching for the parameter space for solutions. However, since Hadoop MapReduce is such a complex framework with a lot of parts that is not well-defined by mathematical model, the What-If Engine theory introduced in this system has a potential to fail in its searching strategy.

Despite several attempts to improve the performance of MapReduce cluster exist, we believe these related work did not provide a comprehensive and reliable solution to optimize the configuration settings of MapReduce cluster. Even though there are some slight similarities, our research is different from all the works above; we not only use data mining technique to analyze job profile, but also optimize the performance of MapReduce applications, no matter whether they are executed or totally new, in a more robust way.

CHAPTER III

Design of the PPABS System

This chapter describes the details of the Profiling and Performance Analysis Based Self-Tuning System (PPABS). The objective of our system is to find the optimal configuration settings for Hadoop MapReduce cluster so that it could manage resources well for different applications running on this cluster. We present the overall structure of this system in section III.1, and then we present our work in details from section III.2 to section III.5.

III.1 Overall view of PPABS System

The overall design of our system can be viewed as comprising two major parts: the

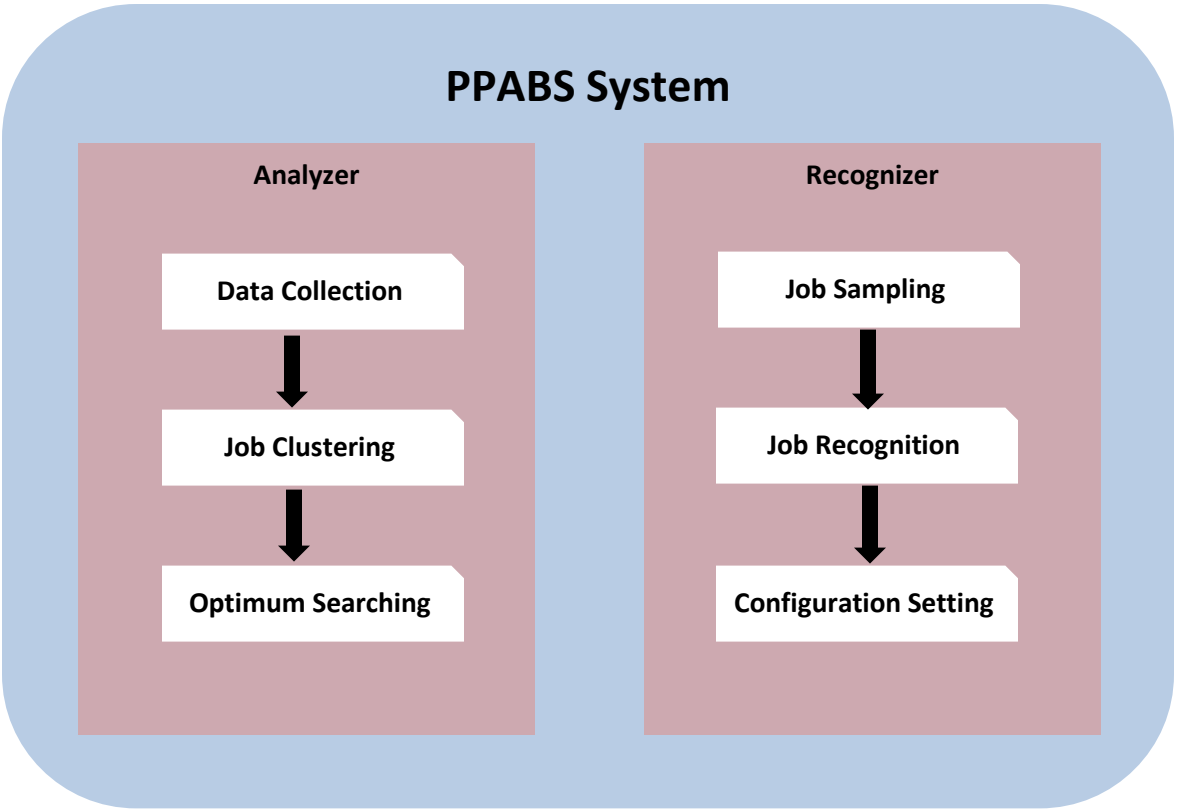

Figure 1 Structure of PPABS

Analyzer and the Recognizer. The Analyzer is based on history is a completely offline step; while the Recognizer runs each time when a client submits a job, therefore, it is semi-online. Figure III.1 shows the architecture of the PPABS system.

There are three steps in the analyzing part: Data Collection, Job Clustering and Optimum Searching. In the first step, we collect data from previous jobs and model a job's performance pattern along several attributes and their values. Using these attributes, in the second step we group previous jobs using a clustering algorithm. Subsequently we search the configuration parameters space to find an optimal solution for each cluster with our modified Simulated Annealing algorithm.

The recognition part consists of three steps: Job Sampling, Job Recognition and Configuration Setting. When a new job is submitted by the client, PPABS system samples this job by running it with only part of its input data at first. Based on the job profile gathered from the first step, in the Job Recognition step the system recognizes this unknown job and classifies it into one of the groups created by the Analyzer. After completing these steps above, the Recognizer then loads the configuration files corresponding to the previously identified group and runs the submitted job with its entire input data set.

III.2 Collecting History Data of Job Performance

Based on a study of previous works [16] [18], we arrived at a very important conclusion: MapReduce applications can be classified into a finite set of groups where each group illustrates similar CPU and I/O utilization pattern. However, through our research we also have found that different sizes of input data will affect the whole

performance pattern to some degree. Therefore, it is necessary to design a more accurate method – beyond CPU and I/O utilization – to define the performance. In our case, we redefined the performance model and developed a new solution, which can not only eliminate this effect caused by different data size, but also describe jobs' performance based on their features, for example, CPU-bound or IO-bound.

In this new model, we regarded a whole computer system over a period of time, which is also our sampling time interval, to be composed of three subsystems: CPU, memory and disk. Each of them is responsible for a time interval in this entire sampling period. For instance, if a sampling period is 2 seconds, then it is of interest to us to know the amount of time that the computer spends on CPU, memory and disk in this period. Considering that the CPU subsystem is either running in kernel mode or user mode all the time, we also divide it into two parts: kernel and user. Therefore, the performance model in a given time interval can be described as Equation III.2-1.

$$\mathbf{P[x] = Kernel[x] + User[x] + Idle[x] + IO[x] + Steal[x]} \qquad \textbf{III.2-1}$$

In Equation **III.2-1**, P is the performance model and x is used to indicate the index of the sampling interval, e.g. P [3] means the performance in the third interval. More details of each variable can be found in the Table 1.

Since equation III.2-1 is for only one time interval, the entire running performance of a Hadoop MapReduce job (more details of Hadoop are described in the Appendix), of which the running time could be divided into N intervals, is captured in Equation III.2-2 and Equation III.2-3. As an example, Figure 2 shows part of the data we collect from a previous job.

Table 1 Description of variables in performance model

Name	Description
Kernel	The amount of time CPU spends in the kernel mode
User	The amount of time CPU spends in the user mode
Steal	The amount of time CPU spends in involuntary wait
IO	The amount of time system spends on IO
Idle	The amount of time when system is idle

	%User	%Kernel	%IO	%steal	%Idle
06:16:22 PM	0.00	0.50	0.00	0.00	99.50
06:16:24 PM	0.00	0.00	0.00	0.00	100.00
06:16:26 PM	0.50	0.50	0.00	0.00	99.00
06:16:28 PM	0.00	0.00	0.00	0.00	100.00
06:16:30 PM	0.00	0.00	0.00	0.00	100.00
06:16:32 PM	4.00	2.50	1.50	1.00	91.00
06:16:34 PM	5.56	3.03	4.04	0.51	86.87
06:16:36 PM	15.26	7.37	0.00	0.00	77.37
06:16:38 PM	9.42	7.85	48.17	0.00	34.55
06:16:40 PM	0.50	0.00	3.98	0.00	95.52
06:16:42 PM	0.00	0.50	0.00	0.00	99.50
06:16:44 PM	8.42	3.96	0.50	7.43	79.70
06:16:46 PM	11.56	9.55	0.50	0.00	78.39
06:16:48 PM	27.67	11.65	0.00	24.27	36.41
06:16:50 PM	19.90	16.33	22.45	10.20	31.12
06:16:52 PM	14.21	9.14	0.00	0.00	76.65
06:16:54 PM	6.67	5.64	0.00	0.00	87.69
06:16:56 PM	0.00	0.51	77.04	0.00	22.45
06:17:00 PM	14.43	18.04	0.00	10.31	57.22
06:17:02 PM	28.02	20.77	7.25	23.67	20.29
06:17:04 PM	26.79	22.49	0.48	30.14	20.10
06:17:06 PM	17.02	19.15	5.32	0.00	58.51
06:17:08 PM	17.73	28.08	34.48	19.70	0.00
06:17:10 PM	8.87	20.69	56.65	13.79	0.00
06:17:12 PM	7.80	20.00	29.27	0.98	41.95
06:17:14 PM	4.64	8.25	0.00	0.52	86.60
06:17:16 PM	7.29	11.98	15.10	0.00	65.62

Figure 2 Performance analysis of Word count

$$\mathbf{P\,[k] = P_M\,[k] \cup \overline{P}_W\,[k]} \qquad \textbf{III.2-2}$$

The master node is responsible for collecting data and assigning tasks to the slave nodes, however, slave nodes only execute their own tasks based on the instruction from master. Therefore, on one hand, the performance of master node and the performance of slave nodes are significantly different when running a MapReduce job; on the other hand, the difference among the slave nodes is very slight. However, because the size of a Hadoop MapReduce cluster is often large, it is not feasible to use all the utilization data gathered from each slave node. To solve this problem by processing the gathered data, we came up with Equation III.2-2, in which P_M is the performance of the master node in a Hadoop MapReduce cluster, and $\overline{P}_W$ is the average performance of all worker nodes in this cluster.

$$\mathbf{P\,[Total]} = \sum_{k=0}^{n} \boldsymbol{P\,[k]}\textbf{, } \boldsymbol{P\,[k]} \textbf{ is the performance in } \boldsymbol{kth} \textbf{ interval} \qquad \textbf{III.2-3}$$

As shown above, we have gathered five-dimensional statistical data from previous jobs that have been executed. However, since our sampling technique is time-series based, it is obviously impossible to get the same running time from different jobs; for instance, it takes 5 minutes to complete job A while it could take 10 minutes or even longer to complete another. In fact, the length of data gathered always varies from job to job. In our case, a reasonable solution to solve this issue of data heterogeneity and make sure we can get same length of time series data is sampling [22]. In this stage, the collected data of each job is converted into normalized sets with the same length.

III.3 Job Clustering

Since the goal of our research is to optimize the configuration settings of MapReduce cluster for each MapReduce application, no matter if it is a totally new one or an executed one, it is very important to group all previous executed jobs based on their performance pattern we gathered from Section III.1 so that our system could recognize and classify a new unknown job with the group rules and make sure an optimal solution of this group is loaded correctly. Therefore a clustering algorithm is needed in this grouping situation.

Several clustering algorithms are frequently used in the field of Data Mining [23]. These algorithms are usually categorized as Connectivity-based Clustering, Centroid-based Clustering, Distributed-based Clustering, Density-based Clustering and others. After doing a comparison on their cluster models and use case, we believe K-Means++ [24], a popular Centroid-based Clustering algorithm, is an appropriate way for this research.

K-Means++, firstly proposed by David Arthur and Sergei Vassilvitskii in 2007, is an algorithm that is regarded as one of the K-Means type algorithms for which the goal is to improve the clustering performance of K-Means, the original one for all algorithms in this type. In K-Means, a D-dimensional vector is considered as a point and the target of this method is to automatically partition N input points into K clusters in which each point belongs to a cluster with the shortest distance from the point itself to the center of this cluster. In our case, since the performance data of a MapReduce job, processed from Section III.1, is a multidimensional vector, it is also appropriate to be regarded as a point.

However, one major disadvantage is that the clusters found by the K-Means algorithm could be arbitrarily bad compared to the optimal solution, which was found by computer scientists after years of research.

Compared to the original one, the K-Means++ algorithm specifies the cluster initialization stage before proceeding with the standard K-Means clustering algorithm. This improvement not only addresses the obstacle we mentioned above but also guarantees finding a solution that is O (log k) competitive to the optimal one. However, K-Means++ algorithm could not be used without any modification in our case because the cluster centers got from it is very likely to be virtual points which cannot be mapped to real MapReduce applications and thus make no sense to the next step of our PPABS system. To this point, we have modified this well-known algorithm with a selection stage so that we can make sure the centers we finally find are real points. The details of the algorithm are described in Algorithm 1. In the modified K-Means++ algorithm, we firstly initialize the clusters by randomly selecting their centers, then we iteratively update these clusters by reassigning points to them, finally we select the real cluster centers by finding the nearest point of the virtual center for each cluster.

Each real cluster center we find from this Job Clustering step is eligible to stand for the cluster itself, which is also a group of MapReduce applications that have similar performance patterns which are also assigned to the same cluster. We will discuss how to find the optimal configuration with these clusters in Section III.4.

Algorithm 1: The modified K-Means++ Algorithm

Initialization

1: $Cluster_1$ = Random Point p in χ, $P_1, P_2, P_3 \ldots P_N \in \chi$
2: int $i = 1$
3: **for** each Point p in χ
4: update $D(p)$, $D(p)$ is the distance from p to its nearest cluster center
5: **end for**
6: **while** $i <= K$, K is the total number of clusters **do**
7: **for each** Point p in χ
8: update $Probability(p) = D(p)\,\hat{}\,2 \,/\, \sum_{x \in \chi} D(x)\hat{}\,2$
9: **end for**
10: i++
11: $Cluster_i$ = Point x with highest probability
12: **for each** Point p, $p \neq C_1, C_2, C_3 \ldots C_i$, in χ
13: update $C^{(p)}$, $C^{(p)}$ is its nearest cluster center of p
14: update $D(p)$
15: **end for**
16: **end while**

The Original K-Means algorithm

1: **for** int $j = 1$ to 100
2: **for each** Point p in χ
3: update $C^{(p)}$, $C^{(p)}$ is its nearest cluster center of p
4: update $D(p)$
5: **end for**
6: **for each** Cluster C_i in C, $C_1, C_2, C_3 \ldots C_K \in$ C
7: update $\mu(C_i)$ = average of points assigned to C_i, $\mu(C_i)$ is the position of C_i
8: **end for**
9: **end for**

Selecting the real centers

1: **for each** Cluster C_i in C, $C_1, C_2, C_3 \ldots C_K \in$ C
2: R (C_i) = the point nearest to center of C_i
3: **end for**
4: **Return** R, R (C_1), R (C_2), R (C_3) … R (C_K) $\in$ R

III.4 Optimum Searching

We have found several "center" MapReduce applications which can be mapped from cluster centers we found from Section III.3 by clustering all executed jobs. Before we describe our heuristic optimum searching algorithm by iteratively running these "center" jobs with different configuration settings, it is necessary to discuss the parameters of Hadoop MapReduce cluster first. An overview of Hadoop MapReduce parameters is presented in Section III.4.1. Besides, we also analyze and discuss how we select the most significant parameters from over 100 parameters of Hadoop MapReduce configuration in Section III.4.2. Then the details of the searching algorithm are presented in Section III.4.3.

III.4.1 Hadoop MapReduce Parameters

There are more than 100 parameters available for users to change their values in the Hadoop MapReduce framework. Based on the way they make an impact on the performance of MapReduce applications, these parameters are generally divided into three groups: *core parameters, mapreduce-relevant parameters* and *DFS-relevant parameters* (*DFS* stands for *Distributed File System*). Even though there are other methods, such as the one that categorizes the parameters by defining a parameter as job-relevant or cluster-relevant, we still believe the general method we mentioned is most helpful for our PPABS system. Actually, configuration files are provided in Hadoop for setting the values of parameters in each group.

core parameters: They are used for defining the most important features of a MapReduce cluster. The parameters in this group are only associated to the cluster itself

such as where the temp data is stored, how large the buffer size is, what the threshold of shuffle is, and el.

mapreduce-relevant parameters: The parameters in this group are closely relevant to the MapReduce procedure: some of them have direct effect only on Map phase or Reduce phase, while others may have effect on both phases.

DFS-relevant parameters: The parameters, such as the one specifying how many replicas should be stored, belonging to this group are associated with DFS.

III.4.2 Parameters Selection

More or less, each parameter may have an impact on the global performance of a MapReduce cluster. If the number of parameters is M, and the number of possible values that could be selected for i^{th} parameter is N_i, i=1, 2, 3 … M, therefore we can get the entire size of the parameters space as shown in Equation III.4-1.

$$S = \prod_{i=1}^{M} N_i \qquad \text{III.4-1}$$

Even though some parameters have Boolean values, which means $N_i = 2$, the values' types of most parameters are still Integer or Double. Therefore, assuming there are 100 parameters and the average number of their possible values is 1,000, the size of the search space becomes 100,000. In this situation, the searching time in total is 20,000,000 seconds if the average running time for a MapReduce application is 200 seconds (in most cases, the running time could be longer than 1,000 seconds).

Therefore, it is unrealistic and expensive for us to use all parameters in this research; on the contrary, our strategy to decrease the size of the parameters space consists of three steps. 1) For parameters that have binary values, we just simply either set their values as default or the values recommended by Apache Hadoop group [25]. 2) For others, following previous works about Hadoop parameters [16] [26] [27], we compared the parameters with each other based on their impact on the performance of a MapReduce cluster and we only select the most important ones. 3) For the parameters we selected from step 2), instead of selecting the possible values from the original range, we use the optimal range of their values described in [28]. Then the list of parameters being selected to build the searching space is shown in the Table III.2.

III.4.3 Searching Algorithm

From the analysis above, the size of the entire search space has shrunk significantly. Now our parameters space can be modeled mathematically as a two-dimensional vector **S** in which every parameter is a one-dimensional vector $\vec{S}$[i], i=1, 2, 3 … N. In addition, the length of $\vec{S}$[i] depends on how many possible values could be set for the i^{th} parameter. For example, the length of $\vec{S}$[6], *mapred.reduce.parallel.copies* is 6 while the length of $\vec{S}$[7], *dfs.block.size*, is 9 if we change it by 64 each time. Then our goal of finding the optimal configuration settings has been converted to searching for the optimal combination of these vectors so that the output, which is the execution time for a MapReduce job, of this combination is as small as possible.

Table 2 Description of variables in performance model

Name	Default Value	Optimal Range	Description
io.sort.mb	100	100 ~ 300	The size of buffer when sorting files
io.sort.factor	10	50 ~ 100	The number of streams while sorting
io.sort.spill.percent	0.8	0.5 ~ 0.8	The soft threshold to decide whether spill contents to disk
mapred.tasktracker.map.tasks.maximum	2	1 ~ 4	The maximum number of Map tasks running on a node
mapred.tasktracker.reduce.tasks.maximum	2	1 ~ 4	The maximum number of Reduce tasks running on a node
mapred.child.java.opts	200m	200~1000m	Java opts for the children processes
mapred.reduce.parallel.copies	5	6 ~ 12	The number of parallel transfers running in Reduce phase.
dfs.block.size	64m	128~640m	The size of a block in file system.

In Mathematics and Computer Science, this problem is categorized into the group of Combinatorial Optimization problems [29], a topic that is aimed at searching an optimal solution from a finite set of objects. However, it is not feasible to use exhaustive searching algorithm, for example, brute force, to solve Combinatorial Optimization problems because a lot of problems of this kind are proven to be NP-hard such as the well-known Traveling Salesman Problem. In this case, an alternative metaheuristic method to achieve our goal is Local Search.

There are several well-defined methods within the field of Local Search, for instance, Hill Climbing, Tabu Search and Simulated Annealing. The drawback with Hill Climbing, a classic Local Search algorithm, is called *premature convergence*, which means this greedy algorithm is very likely to find the nearest local optimum with low quality [30]. On the other hand, even though Tabu Search can prevent this disadvantage by maintaining a Tabu list to record the previous tries, it is not a reasonable method because it is too time consuming. In Tabu Search, every neighboring candidate should be tried so that this algorithm can choose the best one to navigate. Whereas in our case, using Tabu Search means to run a MapReduce job several times with every possible configuration settings in each step; it is definitely unrealistic and inefficient.

Compared to these two algorithms described above, we believe Simulated Annealing is more feasible for us to implement in PPABS system to search the optimal solution for MapReduce cluster configuration. The main reason is that Simulated Annealing prevents low quality local optimum by occasionally accepting a solution, even if it may be worse than the current one, with a probability-based mechanism [31]. However the shortcoming of the standard Simulated Annealing method is obvious: it could simply repeat the previous track without "remembering" that it has done the same step before. For the purpose of preventing our system from entering this situation, we have modified the original one by combining it with conception of Tabu Search. That is to say, we add a memory structure to the standard one to record the latest status in our approach. Before introducing the details of the modified Simulated Annealing algorithm in Algorithm 2, it is necessary to describe its terminology and how we combine it with our research firstly.

Energy: In PPABS the criterion to decide whether a candidate solution is good is the running time of a MapReduce application with a configuration setting.

Neighbors: They represent the states that this algorithm could "jump" from the current state. In our system, they are the nearest configuration settings we could change from the current one. For instance, if the current solution vector is [S[0][x], S[1][y], S[2][z] … S[M][α]], then its neighbors can be [S[0][x±1], S[1][y], S[2][z] … S[M][α]], [S[0][x], S[1][y±1], S[2][z] … S[M][α]] … [S[0][x], S[1][y], S[2][z] … S[M][α±1]].

Probability and Temperature: This criterion decides whether an attempt will be selected.

The PPABS system searches with the modified Simulated Annealing algorithm, shown in Algorithm 2, to find optimal parameters settings for each "center" MapReduce application. This algorithm initializes variables such as *Temperature*, *Current Energy* and *Memory* first, and then it iteratively searches the parameters space to find candidate solutions. In this case, whether a candidate solution is good or bad depends on its performance, which is the execution time of running a MapReduce application with the candidate configuration settings. When the iteration ends, this algorithm returns the best solutions for each "center" MapReduce application. The PPABS system then generates the configuration files based on the best solutions and saves these configuration files into a configuration library for future use by the Recognizer.

Algorithm 2: The modified Simulated Annealing Algorithm

1: **Input**: a MapReduce application Job_i // our goal is tuning the configuration for it
2: Set $Sulotion_{current}$ = Default Configuration // initialization
3: double $Energy_{current}$ = Running time of Job_i
4: double $Energy_{best}$ = $Energy_{current}$
5: Set $Sulotion_{best}$ = $Sulotion_1$
6: int *count* = 1
7: int *T*= 2000
8: Initialize List<State> *Memory*
9: **while** *count* < = *COUNT_MAX* and $Energy_{current}$ > *ENERGY_BOTTOM* **do**
10: *T* = *T* / log *COUNT_MAX*
11: List<Set> *Neighbors* = getNeigbors ($Sulotion_{current}$)
12: Set $Sulotion_{Attempt}$ = randomly select one solution from *Neighbors*
13: double $Energy_{Attempt}$ = Running Job_i with $Sulotion_{Attemp}$
14: double *Probability* = $e^{(Energycurrent-EnergyAttempt)/T}$
15: double *Threshold* = randomly generate a double from (0, 1)
16: **if** *Probability* >= *Threshold* **then** // check the memory
17: **if** *Memory* contains the target state **then**
18: **continue**
19: **end if**
20: $Sulotion_{current}$ = $Sulotion_{Attempt}$; $Energy_{current}$ = $Energy_{Attempt}$
21: Update *Memory* with storing the latest state and remove the oldest one
22: **if** $Energy_{current}$ < $Energy_{best}$ **then**
23: $Energy_{best}$ = $Energy_{current}$; $Sulotion_{best}$ = $Sulotion_{current}$
24: **end if**
25: **end if**
26: count ++
27: **end while**
28: Configuration $Conf_{final}$ = $Sulotion_{best}$
29: **Output**: $Conf_{final}$

III.5 Design of the Recognizer

We have discussed how we analyzed the history data and how we tune settings of a Hadoop MapReduce cluster, in which all the work is based on the profiling of executed jobs. In this section we describe how we recognize a new incoming MapReduce application and select the optimal configuration settings for it based on the learned behavior.

III.5.1 Job Sampling

One major advantage of the Distributed File System [32], considered to be the basis for Cloud Computing technology, is it uses objects to associate logic paths with physical addresses so that it becomes possible for us to break data with very large size into a number of smaller parts and then store them distributedly while keeping a unified logic path for this data. Similarly, HDFS, the Hadoop Distributed File System, is an implementation of the Scalable Distributed File System for Hadoop MapReduce framework. Since the entire file system is broken into blocks, any very large data set submitted is also stored by blocks. This feature of HDFS makes it possible to sample a job with only using data from a few blocks instead of the entire data set.

Moreover, typically a MapReduce job consists of two parts: the job itself and data. After comparing these two parts we have found that the size of the job part is much smaller than the size of data for most MapReduce jobs. Take WordCount, a benchmark application developed by Apache Hadoop, as an example; the size of this job's codes is only a few kilobytes, whereas the size of the input data set can become as large as hundreds of terabytes. Moreover, since the performance pattern of a MapReduce job is

closely related to the job itself [14] instead of the size of data set, we believe that sampling job, which means to run a newly submitted job with only a sample part of input data, is reasonable for us to understand the performance pattern of this job so that our system could select the optimal configuration to load before running it with the entire data set.

III.5.2 Job Recognition

At the same time, when we are running a new job with part of input data, we are also collecting the performance data from this Job Sampling step as above. Similar to what we mentioned in Section III.2, the data gathered from the sampling step is a multidimensional time series. Since we used "center" to describe the performance patterns of a group of MapReduce jobs in Section III.3, the procedure of Pattern Recognition is converted to a problem of finding the nearest center of a point if we model the job we need recognize as a point which is newly added to the clustering space. Therefore, the algorithm we use is very simple in Algorithm 3.

Algorithm 3: Classify an Incoming Unknown Job

1: **Input**: Point $P_{unknown}$
2: double $D_{current}$ = the distance between P_{unknown} and C_1
3: Cluster $C_{short} = C_1$; double $D_{min} = D_{current}$
3: **for each** Cluster C_i in C, $C_1, C_2, C_3 \ldots C_K \in$ C
4: $D_{current}$ = compute the distance between P_{unknown} and C_i
5: **if** $D_{current} < D_{min}$ **then**
6: update D_{min} and C_{short}
7: **end if**
8: **end for**
9: **Output**: C_{short}

After recognizing and classifying an unknown incoming job, our system automatically loads tuned configuration files from the configuration library and runs this job again with its entire data set. One issue with this solution is that if the number of executed jobs increases with the increasing number of submitted jobs, then it is necessary to re-cluster the jobs so that we can keep the centers being updated. Nevertheless, if the clusters are recomputed, retuning our existing MapReduce configuration settings is also needed. This process, unfortunately, takes considerable time. Therefore, it becomes significantly crucial for us to maintain a balance between updating clusters and making the system stay at its current status. In this case, our approach is to set a counter which is added by one each time when a job is completed. With this mechanism, our PPABS system restarts if the counter reaches the threshold we initially set, otherwise this system just stays and uses the results it previously found.

III.5.3 Cost Model of the Recognizer

The three major steps in the Recognizer are Job Sampling, Job Recognizing and Configuration Setting. Different from the Analyzer, the Recognizer is one semi-online part of our PPABS system. Thus, the time spent on these three steps must be included in the total running time for a new job as bellow:

$$\mathbf{T_{Total} = T_{Sampling} + T_{Recognition} + T_{Setting} + T_{OptTotal}} \qquad \textbf{III.5-1}$$

If we assume the cost model of a MapReduce job as linear related to the size of data set, and the time spent on a MapReduce job running with its entire data set using the default configuration settings is $T_{Default}$, we can get the equation III.5-2:

$$T_{Sampling} = T_{Default} * (S/M) \quad \text{III.5-2}$$

In this equation M is the number of blocks used to store the total data set, while S is the number of blocks used to store the sampling part. Moreover, from the analysis we find that the time spent on the Job Recognition step and Configuration Setting is so small compared to the time spent on other steps, then our cost model is updated as III.5-3. The performance improved by our system can be modeled as III.5-4:

$$T_{Total} = T_{Default} * (S/M) + T_{OptTotal} \quad \text{III.5-3}$$

$$\Delta I = \frac{\frac{M-S}{M} T_{Default} - T_{OptTotal}}{T_{Default}} \times 100\% \quad \text{III.5-4}$$

CHAPTER IV

Experimental Evaluation

The organization of this chapter is as follows: first of all we describe the experimental settings such as how many nodes are in our Hadoop MapReduce cluster in section IV.1, then results gathered from experiments are evaluated in section IV.2.

IV.1 Experimental Settings

We implemented the PPABS system on a Hadoop MapReduce cluster built on the Amazon Elastic Cloud Computing platform and Amazon Virtual Private Network. This MapReduce cluster consists of five data nodes, considered as slave roles only, and one name node which is both a slave role and master role in our system. The details of these nodes are listed in Table 3. Besides, the version of Hadoop we used is 1.0.4 and we have set the number of replicas to be 6 since there are 6 nodes in total in this cluster.

Table 3 Description of cluster composition

Node	Instances Type	CPU	Memory	Storage	Private IP
NameNode	M1 medium	2 EC2 Compute Units	3.75 GiB	300 GB	10.0.0.10
DataNode 1	M1 small	1 EC2 Compute Unit	1.7 GiB	160 GB	10.0.0.5
DataNode 2	M1 small	1 EC2 Compute Unit	1.7 GiB	200 GB	10.0.0.6
DataNode 3	M1 small	1 EC2 Compute Unit	1.7 GiB	250 GB	10.0.0.7
DataNode 4	M1 small	1 EC2 Compute Unit	1.7 GiB	180 GB	10.0.0.8
DataNode 5	M1 small	1 EC2 Compute Unit	1.7 GiB	180 GB	10.0.0.9

Since the Data Mining technique we used to profile and analyze the performance of MapReduce applications is K-Means++, this clustering analysis algorithm needs to be trained first. Thus, it is necessary for us to decide which applications should be included in the training set of our PPABS system. In this experiment, the training set of MapReduce applications consists of three well-known sets that have predominantly being used in research: Hadoop Examples set, Hadoop Benchmarks set and HiBench, which is another benchmark set implemented by Intel. In total there are 48 applications in these three sets, however, we decided to select three most popular applications − WordCount, TeraSort and Grep − to test the performance of our system. Therefore, the size of training set is 45. The training set may not be as large as some experiments in the field of Data Mining, however, in the field of MapReduce, we have plenty of reasons to believe that this set in our research is solid and large enough, especially compared to the related works we mentioned in Chapter II, of which the number of applications used is usually less than 5. Moreover, no matter in the step of Data Collection and Performance Analysis, or in the tuning step, we set the size of input data to be 1GB for each application.

Table 4 Results of Job Clustering

Number of clusters	Average Distance	Sizes of clusters
K = 3	21.3/100	[19, 12, 14]
K = 4	15.7/100	[11, 15, 11, 8]
K = 5	13.5/100	[12, 7, 9, 8, 9]

IV.2 Experimental Results

First of all, we list the intermediate results generated by the Analyzer, which is the offline part of our PPABS system, in Table IV.2. We have tested the clustering algorithm in situations when we set K=3, 4 and 5, and then the size of each cluster and the average distance (normalized by percentage) between each point and the cluster center this point belongs to are also listed in the table above. It can be seen from this table that the accuracy of the modified K-Means++ algorithm we used to cluster MapReduce jobs increases with the growing of K which describes the number of clusters. Whereas, the reason why we didn't set K>5 in this experiment is because a large K will make the tuning step more complicated. Based on the observation from the Analyzer, we decided to set K = 4 for the next step in this evaluation.

Next, to evaluate the performance of the Recognizer in our PPABS system, we provide a comparison between the tuned configuration settings and the default cluster configuration settings for WordCount, TeraSort and Grep in Table 5 after they are submitted to our system. From Table 5, we can find that there is significant difference among the tuned configurations and the default one, for instance, the value of parameter *io.sort.mb* in each of the tuned configuration is twice as large as the default value. Moreover, from the output of the Recognizer, we also notice that the configuration files loaded for these three MapReduce applications we have submitted are also different in some way. Take the parameter *io.sort.factor* as an example, the optimal value our PPABS found for TeraSort is larger than the one for WordCount, while the value of this parameter for Grep is only 30, which is obviously smaller than the others.

Table 5 Comparison of MapReduce Configuration Settings

	Default	WordCount	TeraSort	Grep
io.sort.mb	100	240	220	280
io.sort.factor	10	50	80	30
io.sort.spill.percent	0.8	0.7	0.6	0.8
mapred.tasktracker.map.tasks.maximum	2	4	4	3
mapred.tasktracker.reduce.tasks.maximum	2	2	3	2
mapred.child.java.opts	200M	500M	800M	800M
mapred.reduce.parallel.copies	5	8	10	8
dfs.block.size	64M	256M	256M	374M
mapreduce.map.output.compress	False	True	True	True

Finally, the performance of PPABS is evaluated as follows. We compared the execution time of WordCount with tuned configuration to the execution time with the default settings in Figure 3. Subsequently, we compared the execution time of the other two applications with the tuned configuration to execution time with the default one in Figure 4 and Figure 5. Moreover, in order to evaluate whether our system can improve the performance of MapReduce jobs when the input data is very large, as our expectation, the size of input data is set as 1GB, 5GB, and 10GB. And we also repeat this evaluation several times to prevent any occasional results.

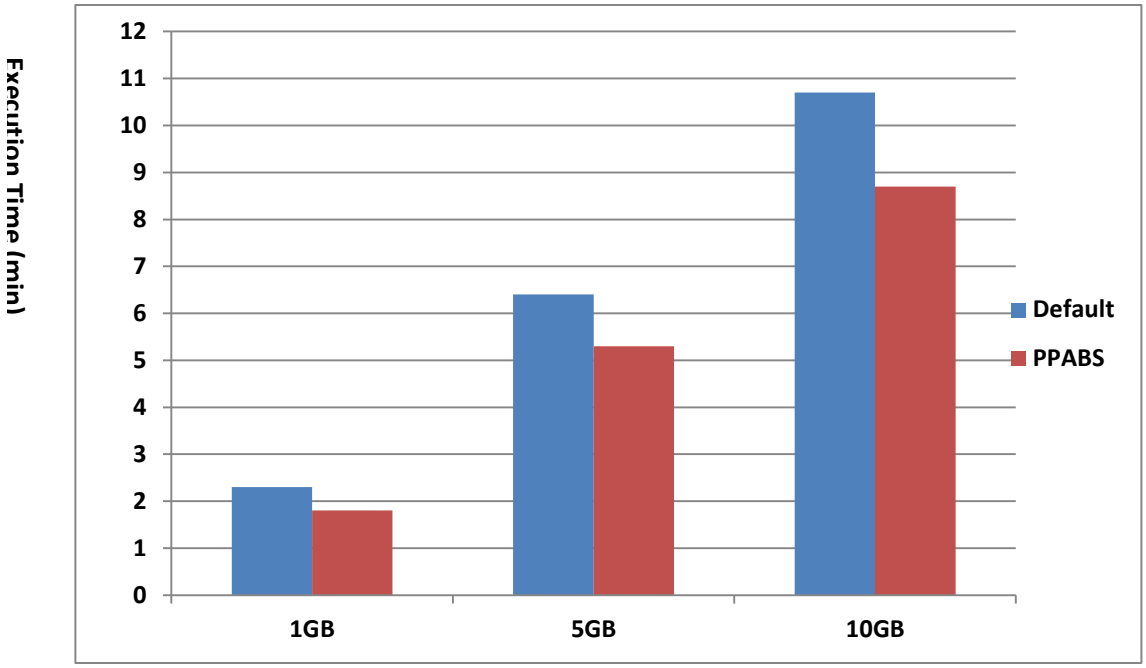

Figure 3 Performance of WordCount

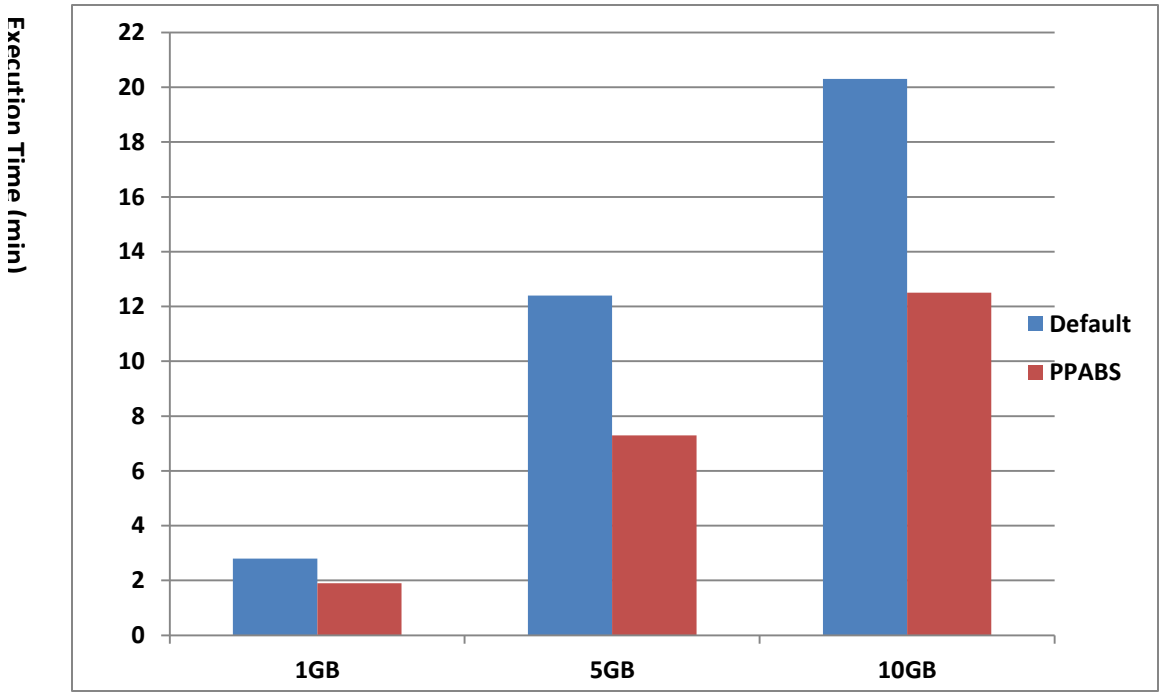

Figure 4 Performance of TeraSort

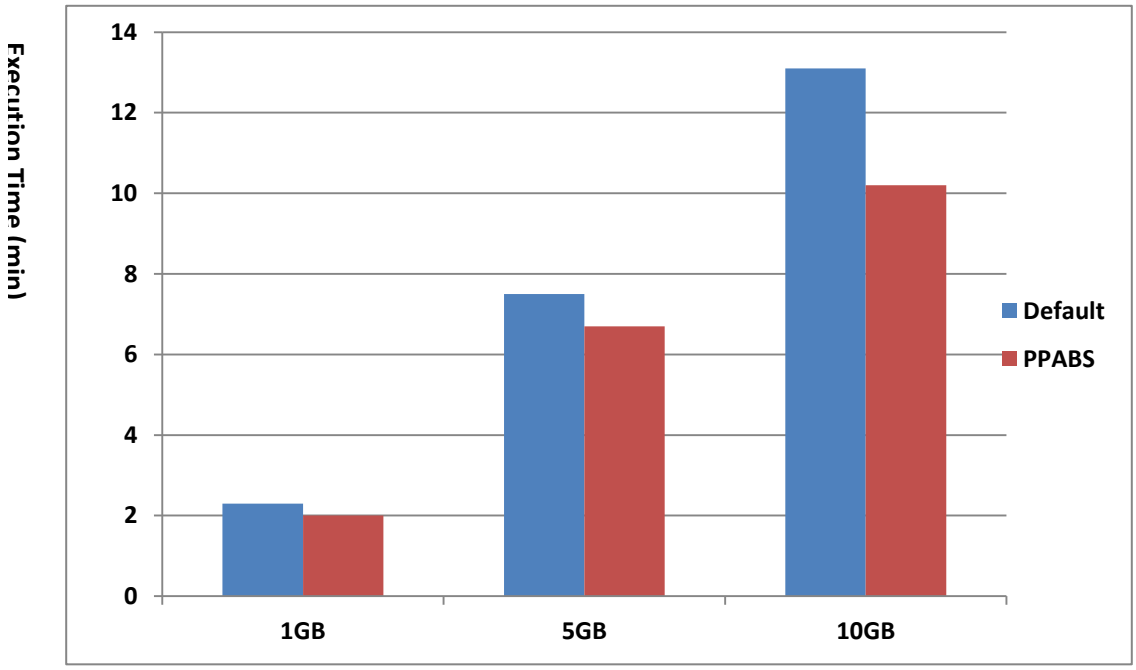

Figure 5 Performance of Grep

The figures above demonstrate that our PPABS system improves the performance of MapReduce cluster. Besides, when the size of input data set is relatively small such as only 1GB, this improvement is not very obvious. While when the size of input data becomes larger, for example, more than 5GB, the performance of all three Hadoop applications is significantly improved by our system. How much improvement in the performance also varies by the different jobs: when the size of input data set is 10 GB, the average execution time for TeraSort decreases by 38.4%, however, in the same situation, the average execution time for WordCount decreases only by 18.7%. The reason of this difference is because our PPABS recognizes the incoming unknown jobs by classifying and assigning each of them to a cluster, but the distance between each job, which can be modeled as a point, and its cluster center varies by the job itself. Therefore,

if a job is far from its cluster center, it is possible that its performance is not optimized as well as the performance of another job which is very close to its cluster center.

CHAPTER V

Conclusion and Future Work

In this thesis, we proposed PPABS, a profiling and performance analysis based self-tuning system that is aimed to optimize the configuration of the Hadoop MapReduce cluster. This system consists of two major parts: the Analyzer and the Recognizer. The former is called by PPABS before a new job is submitted. It analyzes and processes data gathered from executed jobs and then uses a modified K-Means++ clustering algorithm to group these jobs based on their performance pattern. In addition, the Analyzer also uses a modified Simulated Annealing algorithm to search for the optimal solutions for each "center" found from the Job Clustering step. On the other hand, the latter is called when a new job is incoming. It samples the new job by running it only with a small part of its entire data set at first. Then the Recognizer compares the new job's profile with profiles of "centers" and classifies this new-incoming job into one group we previously found. The last step for the Recognizer to do is selecting the tuned configuration files to load and running the new job with updated configuration settings.

We have implemented this system on Amazon EC2, and then we evaluated the performance of PPABS system by running a bunch of real MapReduce applications. The experimental results are promising and they have showed the effectiveness of our approach in improving the performance of several MapReduce applications. Moreover, we have compared the execution time of jobs running with the tuned configuration to that of the execution time of the same jobs running with the default configuration, and then

we observed that the improvement by our PPABS system is significant when the size of input data set is large.

Our future work in this area involves on two enhancements. One is that we plan to add more MapReduce applications into our training set so that we can let our system "remember" more executed jobs as history data. The other enhancement we plan to research is to optimize Job Sampling step with appropriate tradeoffs: on one hand, the time of sampling should be as short as possible; on the other hand, we have to make sure the sampled performance pattern can describe the features of each job as well.

REFERENCES

1. Provost, F. and T. Fawcett, *Data Science and its Relationship to Big Data and Data-Driven Decision Making.* Big Data, 2013. **1**(1): p. 51-59.
2. Ibrahim, S., et al., *Evaluating mapreduce on virtual machines: The hadoop case.* Cloud Computing, 2009: p. 519-528.
3. Ordonez, C., I.-Y. Song, and C. Garcia-Alvarado. *Relational versus non-relational database systems for data warehousing*. in *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*. 2010. ACM.
4. Ranger, C., et al. *Evaluating mapreduce for multi-core and multiprocessor systems*. in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. 2007. IEEE.
5. Dean, J. and S. Ghemawat, *MapReduce: simplified data processing on large clusters.* Communications of the ACM, 2008. **51**(1): p. 107-113.
6. Zaharia, M., et al. *Improving mapreduce performance in heterogeneous environments*. in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. 2008.
7. Dean, J. and S. Ghemawat, *MapReduce: a flexible data processing tool.* Communications of the ACM, 2010. **53**(1): p. 72-77.
8. Dean, J. *Experiences with MapReduce, an abstraction for large-scale computation*. in *PACT: Proceedings of the 15 th international conference on Parallel architectures and compilation techniques*. 2006.
9. Ma, Z. and L. Gu. *The limitation of MapReduce: A probing case and a lightweight solution*. in *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*. 2010.
10. Lämmel, R., *Google's MapReduce programming model—Revisited.* Science of Computer Programming, 2008. **70**(1): p. 1-30.
11. Condie, T., et al. *MapReduce online*. in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. 2010.
12. Tseng, F.-H., et al. *Implement A Reliable and Secure Cloud Distributed File System*. in *IEEE International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS 2012)*. 2012.
13. Wang, K., X. Lin, and W. Tang. *Predator—An experience guided configuration optimizer for Hadoop MapReduce*. in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. 2012. IEEE.
14. Babu, S. *Towards automatic optimization of MapReduce programs*. in *Proceedings of the 1st ACM symposium on Cloud computing*. 2010. ACM.
15. SUN, G.-z., F. XIAO, and X. XIONG, *Study on Scheduling and Fault Tolerance Strategy of MapReduce [J].* Microelectronics & Computer, 2007. **9**: p. 053.
16. Kambatla, K., A. Pathak, and H. Pucha. *Towards optimizing hadoop provisioning in the cloud*. in *Proc. of the First Workshop on Hot Topics in Cloud Computing*. 2009.
17. Tian, C., et al. *A dynamic mapreduce scheduler for heterogeneous workloads*. in *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*. 2009. IEEE.
18. Kavulya, S., et al. *An analysis of traces from a production mapreduce cluster*. in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. 2010. IEEE.

19. Rizvandi, N., et al., *On Modelling and Prediction of Total CPU Usage for Applications in MapReduce Environments.* Algorithms and Architectures for Parallel Processing, 2012: p. 414-427.
20. Lama, P. and X. Zhou. *AROMA: automated resource allocation and configuration of mapreduce environment in the cloud*. in *Proceedings of the 9th international conference on Autonomic computing*. 2012. ACM.
21. Herodotou, H., et al. *Starfish: A self-tuning system for big data analytics*. in *Proc. of the Fifth CIDR Conf*. 2011.
22. Faloutsos, C., M. Ranganathan, and Y. Manolopoulos, *Fast subsequence matching in time-series databases*. Vol. 23. 1994: ACM.
23. Jain, A.K., M.N. Murty, and P.J. Flynn, *Data clustering: a review.* ACM computing surveys (CSUR), 1999. **31**(3): p. 264-323.
24. Arthur, D. and S. Vassilvitskii. *k-means++: The advantages of careful seeding*. in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. 2007. Society for Industrial and Applied Mathematics.
25. Hadoop, A. *Core-Default-site.xml*. Available from: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml.
26. Herodotou, H., *Hadoop performance models.* arXiv preprint arXiv:1106.0940, 2011.
27. White, T., *Hadoop: The definitive guide*2012: O'Reilly Media.
28. Advanced Micro Devices, I. *Hadoop_Tuning_Guide-Version*. Available from: http://developer.amd.com.php53-23.ord1-1.websitetestlink.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.
29. Papadimitriou, C.H. and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*1998: Dover publications.
30. Hinson, J.M. and J. Staddon, *Matching, maximizing, and hill-climbing.* Journal of the Experimental Analysis of Behavior, 1983. **40**(3): p. 321.
31. Aarts, E.H., J.H. Korst, and P.J. Van Laarhoven, *Simulated annealing.* Local search in combinatorial optimization, 1997: p. 91-120.
32. Ghemawat, S., H. Gobioff, and S.-T. Leung. *The Google file system*. in *ACM SIGOPS Operating Systems Review*. 2003. ACM.
33. *Apache Hadoop: What is Apache Hadoop?* ; Available from: http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F.
34. *Apache Hadoop; HDFS Architecture Guide*. Available from: http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html.

APPENDIX

Architecture of Apache Hadoop

As an implementation of MapReduce technology, Apache Hadoop is an open-source software framework that supports reliable and scalable Big Data computing. This framework is written in Java language and it consists of four major modules: Hadoop Common module, Hadoop Distributed File System module, Hadoop YARN module and Hadoop MapReduce module [33].

Hadoop Common module contains the common utilities that support the other Hadoop modules. In this module, scripts and Java ARchive files are provided to support the basic operations of Hadoop such as starting and stopping Hadoop, formatting the NameNode and so on. In addition, it also provides source code, documentation and other necessary files of Hadoop for developers.

Hadoop Distributed File System (HDFS) module is a distributed file system that provides high throughput to access data. Compared to other existed distributed file system, it is designed to work on low-cost hardware and keep high quality fault-tolerance at the same time. HDFS module has a master/slave architecture in which a NameNode is the master that manages the file system namespace and several DataNodes are served as slaves that only manage the nodes they are running on. The detailed architecture of HDFS is shown in Figure 6 [34].

Hadoop Yarn module is a framework for job scheduling and resource management. The fundamental idea of this module is to split the JobTracker, which is responsible to

supervise the running process of MapReduce applications, into a Resource Manager and an Application Master. The former is the ultimate authority that arbitrates the resource of the entire Hadoop system; while the latter is responsible to track and monitor the status of Resource Containers from the Scheduler, one important component of Resource Manager.

Hadoop MapReduce module is a Hadoop YARN-based system for parallel data processing. Similar to the HDFS module, it has a master/slave architecture. For each MapReduce job, a JobTracker is served as a master which can be regarded as a interaction point between the client and the framework. However when Hadoop is running a job, a lot of TaskTrackers, considered as servers, not only execute tasks based on the instruction from the JobTracker, but also handle data motion between the two phases of MapReduce. The architecture of this modules is described in Figure 7.

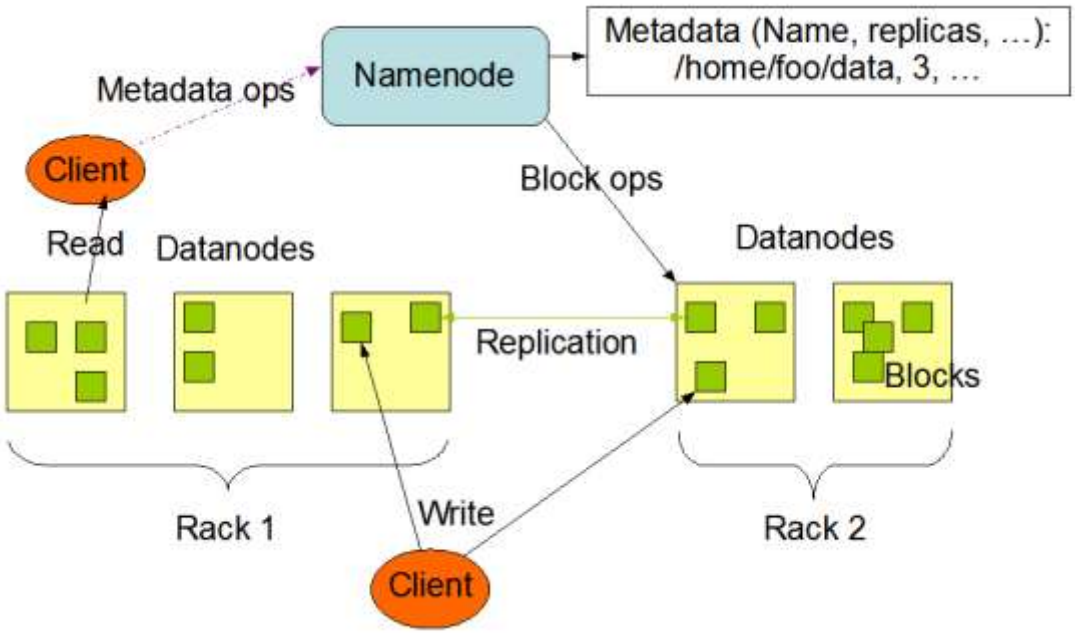

Figure 6 Architecture of Hadoop Distributed File System

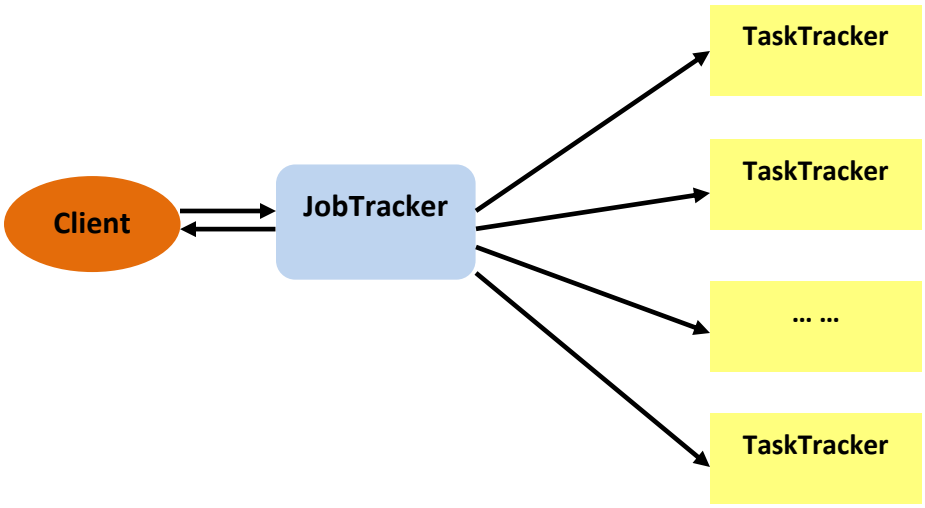

Figure 7 Architecture of Hadoop MapReduce Module