Spring 5-4-2012

AN ENHANCED SELF-ADAPTIVE MAPREDUCE SCHEDULING ALGORITHM

Xiaoyu Sun
xsun@cse.unl.edu

AN ENHANCED SELF-ADAPTIVE MAPREDUCE SCHEDULING ALGORITHM

by

Xiaoyu Sun

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Ying Lu

Lincoln, Nebraska

May, 2012

AN ENHANCED SELF-ADAPTIVE MAPREDUCE SCHEDULING ALGORITHM

Xiaoyu Sun, M. S.

University of Nebraska, 2012

Adviser: Ying Lu

Map-Reduce [6] is "a programming model and an associated implementation for processing and generating large data sets." Hadoop is an open-source implementation of MapReduce, enjoying wide adoption, and is used not only for batch jobs but also for short jobs where low response time is critical. However, Hadoop's performance is currently limited by its default task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. Longest Approximate Time to End (LATE) [27] is a scheduling algorithm that takes a heterogeneous environment into consideration. However, its problem is that it still depends on a static method to compute the progress of tasks. As a result neither Hadoop default nor LATE schedulers perform well in a heterogeneous environment. Self-adaptive MapReduce Scheduling Algorithm (SAMR) [4] is more advantageous than LATE. It uses historical information to adjust the stage weights of map and reduce tasks when estimating task execution times. However, SAMR does not consider the fact that for different types of jobs their map and reduce stages' weights may be different. Even for the same type of jobs, different datasets may lead to different weights. To this end, we propose ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm to improve the speculative re-execution of slow tasks in MapReduce. In ESAMR, in order to identify slow tasks accurately, we differentiate historical stage weights information on

each node and divide them into K clusters using a K-means clustering algorithm; and when executing a job's tasks on a node, ESAMR classifies the tasks into one of the clusters and uses the clusters weights to estimate the execution time of the job's tasks on the node. Experimental results show that among the aforementioned algorithms, ESAMR leads to the smallest error in task execution time estimation and identifies slow tasks most accurately.

ACKNOWLEDGMENTS

The value of the master thesis goes beyond its scientific content since it summarizes the work of three years of education. In my particular case, this thesis symbolically represents all the years spent in UNL with all the gained amount of experience and personal development. I have been working on this thesis for more than one year. This document reports only the final outcome, leaving out everything that is behind it. My thesis is also made by many e-mails, meetings, hours spent in reading articles, testing and debugging. During this process, I had the honor to work with many great people, and I want to thank all of them for their support and patience.

I would like to extend my heartfelt gratitude to my advisor, Dr.Ying Lu, for guiding me throughout this thesis work and giving me invaluable advice at times when I needed it the most. I would also like to thank Chen He who was responsible for making sure that the cluster used to run the experiments worked fine.

At last, but certainly not the least, a special thanks goes to my family. They are a wellspring of motivation for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Today's world, data is growing exponentially, doubling its size every three years [23]. Huge amounts of data are being generated from digital media, web authoring, scientific instruments, physical simulations, and so on. Effectively storing, querying, analyzing, understanding, and utilizing these huge data sets presents one of the grand challenges to the computing industry and research community.

The popular solutions [6] [8] [3] are to build data-center scale computer systems to meet the high storage and processing demands of these applications. Such a system is composed of hundreds, thousands, or even millions of commodity computers connected through a local area network housed in a data center. It has a much larger scale than a traditional computer cluster, while enjoying better and more predictable network connectivity than wide area distributed computing.

One of the most popular programming paradigms on data-center scale computer systems is the MapReduce programming model [6]. MapReduce [6] is "a programming model and an associated implementation for processing and generating large data sets." It was first developed at Google by Jeffrey Dean and Sanjay Ghemawat. MapReduce was used in Cloud Computing in the beginning [7] [2] . Under this model,

an application is implemented as a sequence of MapReduce operations, each consisting of a map stage and a reduce stage that process a large number of independent data items. The system supports automatic parallelization, distribution of computations, task management, and fault tolerance in hopes that programmers can focus on application algorithms without worrying about these complex issues. MapReduce has achieved an increasing success in various applicaitons, such as [9] [21] [5] [15] [22] [28] [26] [24] and [16]

Hadoop[11], which was created by Doug Cutting[12], is the Apache Software Foundation open source and Java-based implementation of the MapReduce framework. Hadoop provides the tools for processing vast amounts of data using the MapReduce framework and, additionally, implements the Hadoop Distributed File System (HDFS)[10]. It can be used to process vast amounts of data in parallel on large clusters in a reliable and fault-tolerant fashion. Consequently, it makes the advantages of MapReduce available to users.

A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance [25] [11] [20] from the programmer. If a node crashes, MapReduce reruns its tasks on a different machine. Equally importantly, if a node is available but is performing poorly, a condition that we call a straggler, MapReduce runs a speculative copy of the straggler task on another machine to finish the computation faster. Without this mechanism of speculative execution [14], a job would be as slow as the misbehaving task. Stragglers can arise for many reasons, including faulty hardware and misconfiguration.

In this work, we address the problem of how to robustly perform speculative execution to maximize performance [1] [17]. Hadoop default scheduler starts speculative tasks based on a simple heuristic that compares each task's progress to the average task progress of a job. LATE MapReduce scheduling algorithm takes a heterogeneous

environment into consideration. However, LATE still has a poor performance due to the static method used to compute the progress of tasks. SAMR shares a similar idea with LATE scheduling algorithm. However, SAMR also uses historical information to tune weights of map and reduce stages and to get more accurate progress scores than LATE. SAMR falls short of solving one crucial problem. It fails to consider other factors such as different job types and different job sizes that can also affect stage weights.

To overcome the deficiency of SAMR, we have developed ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm. Like SAMR, ESAMR is inspired by the fact that slow tasks prolong the execution time of the whole job and different amounts of time are needed to complete the same task on different nodes due to their differences, such as computation and communication capacities and architectures. In addition, to consider other factors that affect a task's progress ESAMR incorporates historical information recorded on each node and K-means cluster identification algorithm to tune parameters dynamically and find slow tasks accurately. As a result, ESAMR significantly improves the performance of MapReduce scheduling in terms of launching the backup tasks.

Chapter 2

Background

In this Chapter, we first describe the MapReduce programming model. It is the basis of ESAMR. We also introduce Hadoop default scheduler, LATE scheduler, and SAMR scheduler. These schedulers' disadvantages have motivated us to develop ESAMR scheduler.

2.1 Basic concepts in MapReduce

MapReduce is a programming model controlling a great number of nodes to handle a huge amount of data by cooperation. A MapReduce application that needs to be run on the MapReduce system is called a job. The input file of a job, which reside on a distributed filesystem throughout the cluster, is split into even-sized chunks replicated for fault tolerance. A job can be divided into a series of tasks. Each chunk of input is first processed by a map task, which outputs a list of key-value pairs. Map outputs are split into buckets based on the key. When all map tasks have finished, reduce tasks apply a reduce function to the list of map outputs corresponding to each key.

In a cluster which runs MapReduce, there is only one NameNode also called master, which records information about the location of data chunks. There are lots of DataNodes, also called workers, which store data in individual nodes. There is only one JobTracker and a series of TaskTrackers. JobTracker is a process which manages jobs. TaskTracker is a process which manages tasks on a node. Before explaining the steps involved in a MapReduce job, let us clarify the terminology that will be used from this point on in this thesis.

- JobTracker

  -Master node controlling the distribution of a Hadoop (MapReduce) job across free nodes on the cluster. It is responsible for scheduling jobs on TaskTracker nodes. In case of a node failure, the JobTracker starts the work scheduled on the failed node on another free node. The simplicity of MapReduce tasks ensures that such restarts can be achieved easily.

- NameNode

  -Master node controlling the HDFS. It is responsible for serving any component that needs access to files on the HDFS. It is also responsible for ensuring fault tolerance on HDFS. Usually, fault tolerance is achieved by replicating data chunks over three different nodes with one of the nodes being an off-rack node.

- TaskTracker (TT)

  -Node actually running the Hadoop tasks. It requests work from the JobTracker and reports back the progress of the work allocated to it. The TaskTracker daemon does not run tasks on its own, but forks a separate daemon for each task. This ensures that if the user code is malicious, it does not bring down the TaskTracker.

- DataNode

  -This node is part of the HDFS and holds the files that are put on the HDFS. Usually, these nodes also work as TaskTrackers. The JobTracker often tries to allocate work to nodes, where file accesses can be done locally.

- ProgressScore (PS)

  -A progress score of a task in the range [0,1], based on how much of a task's key/value pairs have been finished.

- ProgressRate (PR)

  -A progress rate of a task is calculated based on how much a task's key/value pairs have been finished per second.

- TimeToEnd (TTE)

  -TimeToEnd estimates the time left for a task based on the progress rate provided by Hadoop.

- Weights of map function stage (M1) and order stage (M2) in map tasks

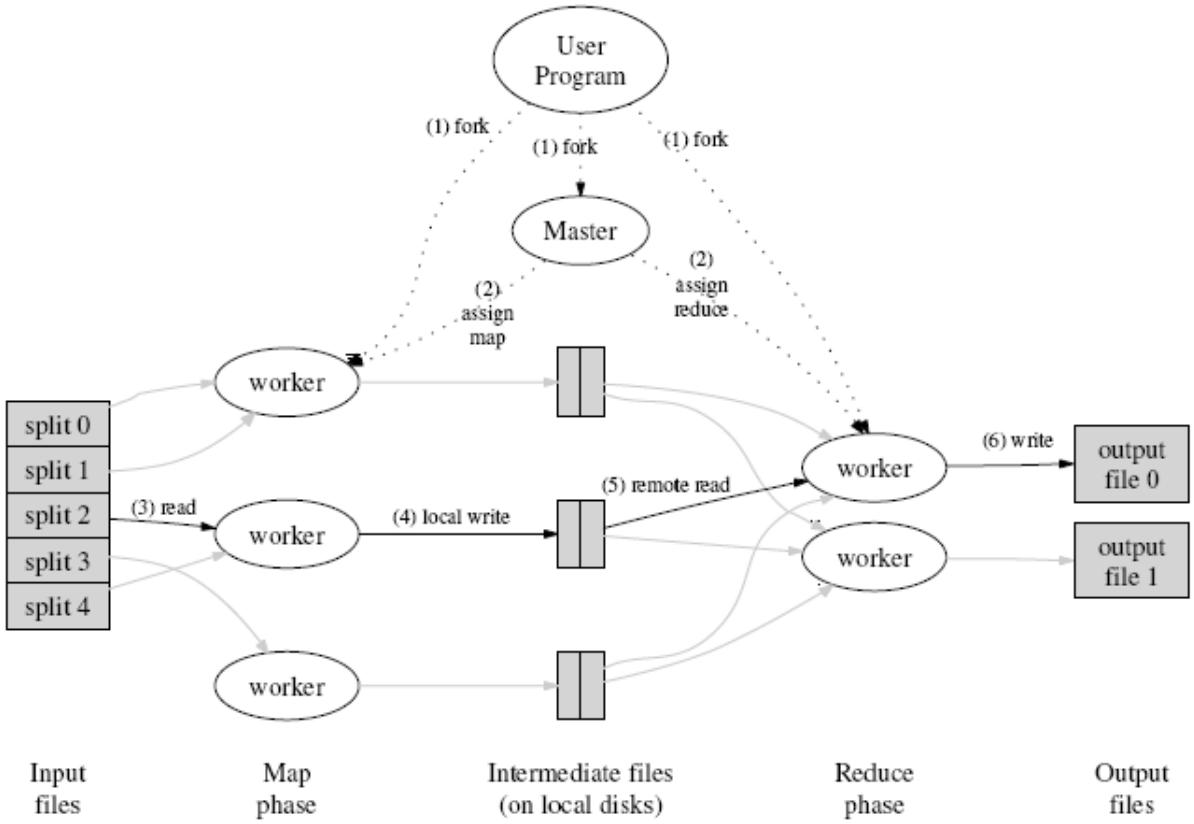
  -M1 and M2 in the range [0,1] record the stage weights in a map task. The sum of M1 and M2 is 1.

- Weight of shuffle stage (R1), order stage (R2), and merge stage (R3) in reduce tasks

  -R1, R2 and R3 in the range [0,1] record the stage weights in a reduce task. The sum of R1, R2 and R3 is 1.

MapReduce scheduling system has six steps when executing a MapReduce job, as illustrated in Figure 2.1.

Figure 2.1: A MapReduce computation. Image from [6]



1. The MapReduce framework first splits an input data file into G pieces of fixed size, typically being 16 megabytes to 64 megabytes (MB) (controllable by the user via an optional parameter). These G pieces are then passed on to the participating machines in the cluster. Usually, 3 copies of each piece are generated for fault tolerance. It then starts up the user program on the nodes of the cluster.

2. One of the nodes in the cluster is special - the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. M and R is either decided by the configuration specified by the user program, or by the cluster wide default configuration. The master

picks idle workers and assigns them map tasks. Once map tasks have generated intermediate outputs, the master then assigns reduce tasks to idle workers. Note that all map tasks have to finish before any reduce task can begin. This is because a reduce task needs to take output from every map task of the job.

3. A worker who is assigned a map task reads the content of the corresponding input split. It parses key/value pairs out of the input data chunk and passes each pair to an instance of the user defined map function. The intermediate key/value pairs produced by the map function are buffered in memory at the corresponding machines that are executing them.

4. The buffered pairs are periodically written to a local disk and partitioned into R regions by the partitioning function. The framework provides a default partitioning function but the user is allowed to override this function by a custom partitioning. The locations of these buffered pairs on the local disk are passed back to the master. The master then forwards these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate key so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys are handled by a reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used. Once again, the user is allowed to override the default sorting and grouping behaviors of the framework. Next, the reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values

to the reduce function. The output of the reduce function is appended to a final output file for this reduce partition.

6. When all map tasks and reduce tasks have completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

2.2 MapReduce scheduling algorithm in Hadoop

One problem of Hadoop default scheduler is that it can not identify tasks which need to be re-executed on fast nodes correctly. Hadoop chooses a task for it from one of three categories:

- Any failed tasks are given the highest priority.
- Non-running tasks are considered. For maps, tasks with data local to the node are chosen first.
- Slow tasks that need to be executed speculatively are considered.

To select speculative tasks, Hadoop monitors the progress of tasks using a Progress Score (PS) between 0 and 1. The average progress score of a job is denoted by PS_{avg}. The Progress Score of the i^{th} task is denoted by PS[i]. It supposes that the number of tasks which are being executed is T, the number of key/value pairs that need to be processed in a task is N, the number of key/value pairs that have been processed successfully in a task is M, the map task spends negligible time in the order stage (i.e., M1=1 and M2=0) and the reduce task has finished K stages and each stage takes the same amount of time (i.e., R1=R2=R3=1/3). Hadoop gets PS according to the Eq. (2.1) and Eq. (2.2), then launches backup tasks according to Eq. (2.3).

$$PS = \begin{cases} M/N & \text{For Map tasks} \\ 1/3 * (K + M/N) & \text{For Reduce tasks} \end{cases} \tag{2.1}$$

$$PSavg = \sum_{i=1}^{T} PS[i]/T \tag{2.2}$$

$$\text{For task } T_i\text{: } PS[i] < PSavg - 20\% \tag{2.3}$$

If Eq.(2.3) is satisfied, T_i needs a backup task. The main disadvantages of this method include:

1. In Hadoop, the values of R1, R2, R3, M1, and M2 are 0.33, 0.33. 0.34, 1 and 0 respectively. However R1, R2, R3, M1 and M2 are different when tasks are running on different nodes, especially in a heterogeneous environment.

2. In Hadoop, the scheduler uses a fixed threshold for selecting tasks to re-execute. Too many speculative tasks may be launched, taking away resources from useful tasks. Because the scheduler launches speculative tasks also by considering their data localities, the wrong tasks may be chosen for re-execution first. For example, if the average progress was 70% and there was a 2x slower task at 35% progress and a 10x slower task at 7% progress, then the 2x slower task might be chosen to run before the 10x slower task if the former's input data was available on the idle node.

3. Hadoop always launches backup tasks for those tasks that satisfy Eq. (2.3), which may not be appropriate. For example, the PS of T_i is 0.7 and needs 120 seconds to finish on a very slow node, while the PS of T_j is 0.5, but only needs 20 seconds to finish on a relatively faster node. Suppose the average progress

score PSavg is 0.8; the method will launch a backup task for T_j according to Eq.(2.3). However, if we launch a backup task for T_i instead of T_j, it will save more time. What is more, a task with PS larger than 0.8 will have no chance to have a backup task, even though its node may be very slow and needs a very long time to finish the task.

4. Hadoop may launch backup tasks for fast tasks. For example, in a typical MapReduce job, the shuffle phase of reduce tasks is the slowest, because it involves all-pairs communication over the network. Tasks quickly complete the other two phases once they have all map outputs. However, the shuffle phase counts for only 1/3 of the progress score. Thus, soon after the first few reducers of a job finish the copy phase, their progress score goes from 1/3 to 1, greatly increasing the average progress. Assuming 30% of reducers have almost finished, the average progress is roughly 0.3*1 + 0.7*1/3 = 53%, and now all reducers still at the beginning of the copy phase will be 20% behind the average, and some of them will be speculatively executed. As a result, task slots will be filled up and true stragglers may never be re-executed.

5. In Hadoop, the 20% progress difference threshold used by the default scheduler means that tasks with more than 80% progress score can never be speculatively executed, because average progress score can never exceed 100%.

2.3 Longest Approximate Time to End(LATE) MapReduce scheduling algorithm

LATE MapReduce scheduling algorithm also uses Eq.2.1 to calculate task's progress score, but launches backup tasks for those that have longer remaining execution times.

Suppose a task T has run Tr seconds. Let PR denotes the progress rate of T, and TTE denotes how much time remains until T is finished. LATE MapReduce scheduling algorithm computes $ProgressRate$ (PR) and $TimeToEnd$ (TTE) according to Eqs. (2.4) and (2.5).

$$PR = PS/Tr \tag{2.4}$$

$$TTE = (1 - PS)/PR \tag{2.5}$$

Advantage of LATE: since LATE focuses on estimating the remaining execution time rather than just the progress score, LATE speculatively executes only tasks that will improve job response time rather than any slow tasks.

Disadvantage of LATE: although LATE uses an improved strategy to launch backup tasks, it still frequently chooses wrong tasks to re-execute. This is because LATE does not approximate TTE of running tasks correctly.

Same as the Hadoop default scheduler, LATE sets the values of R1, R2, R3, M1 and M2 at 0.33, 0.33. 0.34, 1 and 0 respectively. This setting may lead to the wrong TTE estimation. Suppose reduce stage weights R1, R2 and R3 are actually 0.6, 0.2 and 0.2, respectively. When the first stage finishes in Tr seconds, the reduce task still needs $(1 - 0.6)(Tr/0.6) = 0.67Tr$ seconds to finish the whole task. However, the TTE computed by LATE scheduling algorithm is $(1 - 0.33)(Tr/0.33) = 2Tr$ seconds instead.

2.4 A Self-adaptive MapReduce Scheduling Algorithm(SAMR)

SAMR also estimates the remaining execution time to find slow tasks. However, SAMR does not use the fixed stage weights for map and reduce tasks. Unlike Hadoop default and LATE schedulers, which assume M1, M2, R1, R2, and R3 are 1, 0, 1/3, 1/3, and 1/3. SAMR recordes M1, M2, R1, R2, and R3 values on each TaskTracker node and uses these historical information to facilitate more accurate estimation of task's TTE.

Figure 2.2: Two stages of a map task. Image from [4]

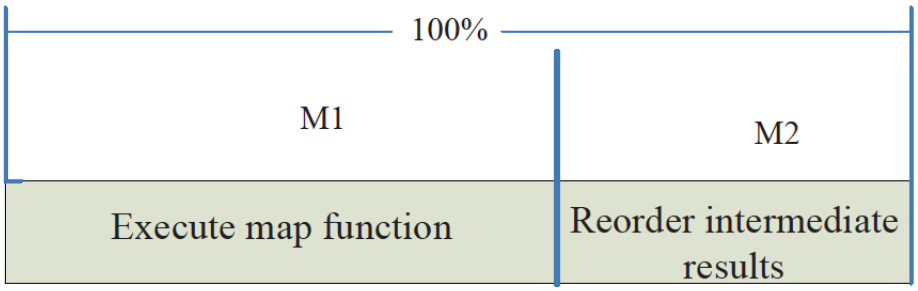
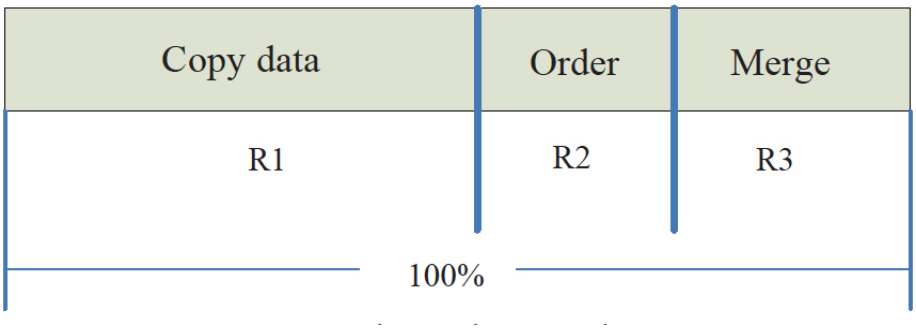


Figure 2.3: Three stages of a reduce task. Image from [4]



SAMR assumes that the number of key/value pairs which have been processed in a task is N_f, the number of overall key/value pairs in the task is N_a, the current stage of processing is S (S could be 0, 1 or 2), and the progress score in the stage is

SubPS. The SubPS in the stage can be computed according to Eq. (2.6). The PS of a task is computed according to Eq. (2.7) and Eq. (2.8). SAMR also uses Eq. 2.4 and Eq. 2.5 to calculate PR and TTE.

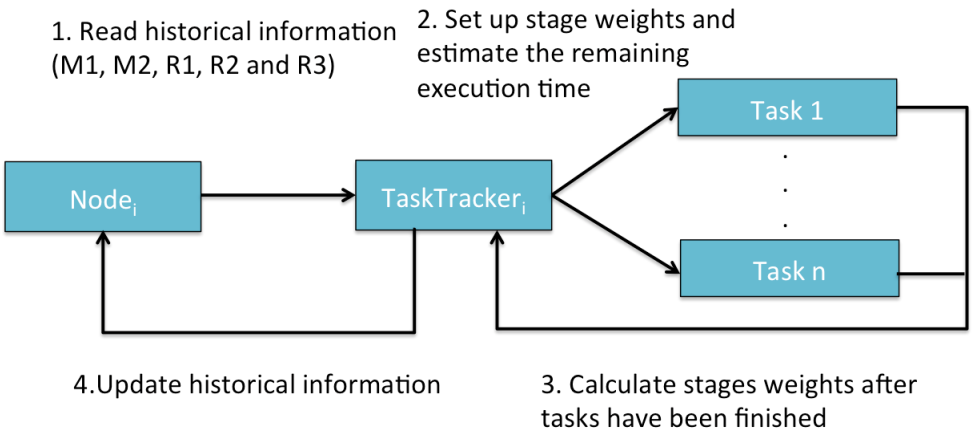
$$SubPS = N_f / N_a \tag{2.6}$$

$$For\ maptask : PS = \begin{cases} M1 * SubPS & \text{if S=0} \\ M1 + M2 * SubPS & \text{if S =1.} \end{cases} \tag{2.7}$$

$$For\ reducetask : PS = \begin{cases} R1 * SubPS & \text{if S=0} \\ R1 + R2 * SubPS & \text{if S =1} \\ R1 + R2 + R3 * SubPS & \text{if S=2.} \end{cases} \tag{2.8}$$

Figure 2.4 shows that SAMR uses historical information recorded for individual node to estimate the stage weight values (M1, M2, R1, R2, and R3) on each node.

Figure 2.4: How to use and update historical information. Image from [4]



Advantage of SAMR: SAMR uses historical information recorded on each node to tune the weight of each stage dynamically. Instead of setting M2=0, SAMR takes the two stages of a map task into consideration for the first time.

Disadvantage of SAMR: although SAMR uses historical information stored on each node to set a more accurate estimate of PS than LATE, it does not consider that different job types can have different weights for map and reduce stages. In addition, the same type of jobs can even have different weights for map and reduce stages when handling datasets with different sizes.

Chapter 3

Empirical Study

As mentioned in Chapter 2, since there are many factors that could affect the weights: the nodes, the job types, and the dataset sizes. We believe that we could not use historical information directly without classifying the information. Therefore, in this Chapter, we investigate and prove our hypotheses by doing some controlled experiments $T - Test$ [19]. We will answer the following questions:

Q1: Does the job type affect the stage weights of map and reduce tasks?

Q2: Does the dataset size affect the stage weights of map and reduce tasks?

Q3: Does the node configuration affect the stage weights of map and reduce tasks?

3.1 Variables and measures

3.1.1 Independent variables

Our experiment manipulates three independent variables:

IV1: Type of a job executed on a cluster. We use two different job types: `Sort` and `WordCount`. These two benchmarks have been widely used in Hadoop and

researchers who developed SAMR [4] and LATE [27] algorithms have also used these two types of jobs in their experiments. We believe these two benchmarks show key characteristics of MapReduce clearly.

IV2: Dataset size for a job. We set job dataset size to be either 2.5GB or 10.0GB.

IV3: Node configuration in a cluster. We consider two kinds of nodes in a cluster. They have different configurations, including different CPUs, memory sizes and I/O capabilities.

3.1.2 Dependent variables and measures

We aim to investigate the stage weights of map and reduce tasks under different conditions. It is straightforward that we select the following five dependent variables.

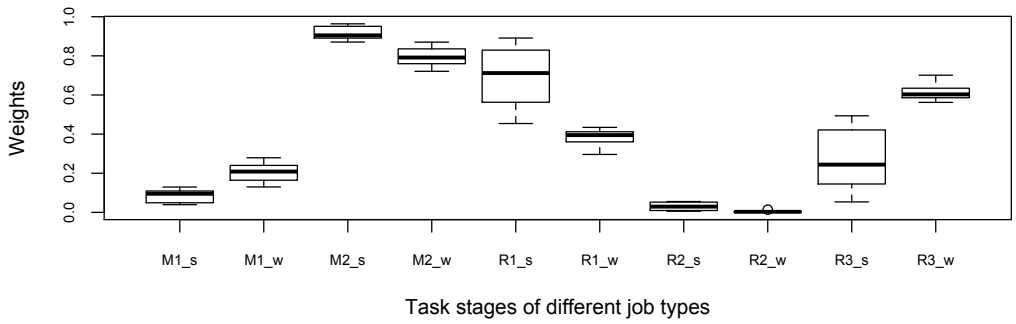
- M1: Weight of the map function stage of a map task
- M2: Weight of the intermediate results ordering stage of a map task
- R1: Weight of the data shuffle stage of a reduce task
- R2: Weight of the data ordering stage of a reduce task
- R3: Weight of the data merging stage of a reduce task

3.2 Experiment setup

Several steps have to be followed to establish the experiment setup needed to conduct our experiments.

We use six computers to compose a cluster for our experiments. This cluster contains one master node and five worker nodes. All the computer use Ubuntu operating

Figure 3.1: Impact of different job types on weights



system. The version of JDK is 1.6.0.26, and the version of Hadoop is 0.21.0. Our algorithm is implemented based on Hadoop 0.21.0. Because we cannot get the primary version of SAMR MapReduce scheduling algorithm, we implement it ourselves according to the algorithm description in [4].

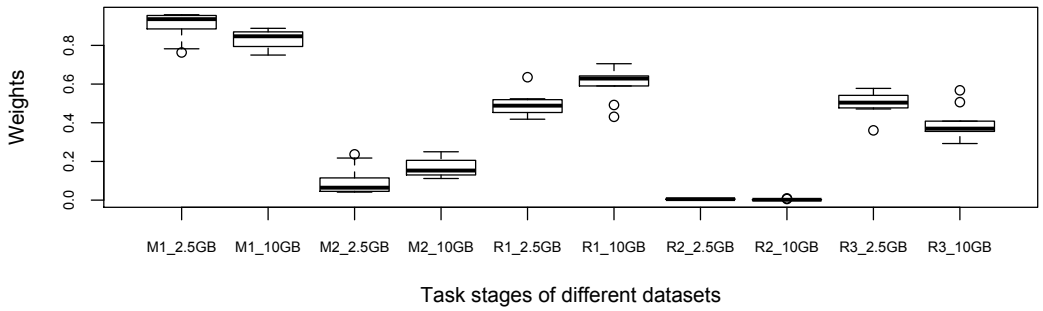
3.2.1 Experimental operations

Given our independent variables, an experiment is specified by three parameters, (N, D, T), where N is one of the node configurations (n1 or n2), D is one of the two dataset sizes(2.5GB or 10.0GB), and T is one of the two job types (Sort or WordCount). Considering randomness, we repeat each experiment 10 times. This results in 40 runs of experiments from which we collect M1, M2, R1, R2, and R3 values using SAMR algorithm.

3.3 Results and analysis

Figures 3.1, 3.2, and 3.3 present box-plots showing the data collected for our independent variables. The first figure plots the stage weights of running two types of

Figure 3.2: Impact of different dataset sizes on weights



jobs with the same size of datasets on Node 1. The second figure plots the stage weights of executing the same type of jobs with different size of datasets on Node 1. The third figure plots the stage weights of executing the same type of jobs with the same size of datasets on two different nodes. The lower bar of each box plot is the smallest data and the upper bar is the largest data. The bottom and the top of the box are the 25th and 75th percentile, and the band near the middle of the box is the 50th percentile. The circles out of the box plots are considered outliers.

3.3.1 Q1: Impact of the job type

To address Q1 (impact of job type), we compare the five weights for map and reduce tasks generated from executing two types of jobs. As the boxplots in Figure 3.1 show, the map tasks for WordCount and Sort have different M1 and M2 values. For the reduce tasks, WordCount and Sort also have very different R1 and R3 values, while R2 values are similar.

To evaluate if these two job types' data are different in statistics, we performed *t-test*s on the data. The *t-test* assesses whether the means of two data groups are statistically different from each other. This analysis is appropriate whenever you want

to compare the means of two data groups. We performed *t-test* on the data for a significant level of 0.05 to reject/validate the null hypothesis: there is no significant difference between the two job types (Sort and WordCount) in terms of task stage weights. Table 5.1 reports the results. As the p-values in the rightmost column show, since it is less than 0.05, there is enough statistical evidence to reject the null hypothesis: that is, the stage weights for map and reduce tasks are different for different job types.

3.3.2 Q2: Impact of the dataset size

To address Q2 (impact of the dataset size), we compare the five weights obtained for jobs with 2.5GB and 10GB datasets. From Figure 3.2, we can see different M1 and M2 values in map tasks for the two dataset sizes. For reduce tasks, R1 and R3 are also different for the two sizes.

We also performed *t-test*s on the data for a significant level of 0.05 to reject/validate the null hypothesis: there is no significant difference between two job dataset sizes (2.5GB and 10GB) in terms of task stage weights. Table 3.2 shows the results. As the p-values in the rightmost column indicate, there is enough statistical evidence to reject the null hypothesis: that is, the stage weights for map and reduce tasks are different for different dataset sizes.

3.3.3 Q3: Impact of node configuration

To address Q3 (impact of node configuration), we compare the five weights obtained when running jobs on two different nodes. As the boxplots in Figure 3.3 show, M1 and M2 values are very different for map tasks executing on the two nodes. For reduce tasks, we can also see different R1 and R3 values.

We also performed *t-test*s on the data for a significant level of 0.05 to reject/validate the null hypothesis: there is no significant difference between two node configurations (n1 and n2) in terms of task stage weights. Table 3.3 shows the results. As the *p*-values in the rightmost column show, there is enough statistical evidence to reject the null hypothesis: that is, the stage weights of map and reduce tasks are different for different node configurations.

3.3.4 Additional analysis

To complete our study, we performed another experiment to investigate our hypothesis: all these weights would be the same under the same conditions, where we are executing the same types of jobs with dataset of the same size on the same node. Figure 3.4 shows the boxplots, and it is obvious that these weights are similar. We also performed *t-test*s on the data. The p-values in Table 3.4 are all greater than 0.05, which proves that the weights collected under the same condition are similar.

We have identified three factors, the job types, the dataset sizes, and the node configurations that we think could affect the weights of different stages for map and reduce tasks. We did experiments to investigate them separately and the results have proved our hypotheses: they have impacts on the weights of different stages for map and reduce tasks.

Table 3.1: Results of t-test on job types

Stage	**Sort**	**WordCount**	**P-value**
m1	0.09	0.21	3.90E-06
m2	0.91	0.79	4.00E-06
r1	0.70	0.37	8.26E-06
r2	0.03	0.02	0.77E-03
r3	0.27	0.61	3.87E-06

Table 3.2: Results of t-test on job sizes

Stage	2.5GB	10GB	P-value
m1	0.90	0.83	0.02
m2	0.10	0.17	0.02
r1	0.49	0.60	0.41E-02
r2	0.02	0.01	0.02
r3	0.49	0.39	0.48E-02

Table 3.3: Results of t-test on node configurations

Stage	Node1	Node2	P-value
m1	0.75	0.19	6.56E-16
m2	0.25	0.81	6.56E-16
r1	0.19	0.41	6.90E-07
r2	0.02	0.01	0.03
r3	0.79	0.58	4.31E-07

Figure 3.3: Impact of different node configurations on weights

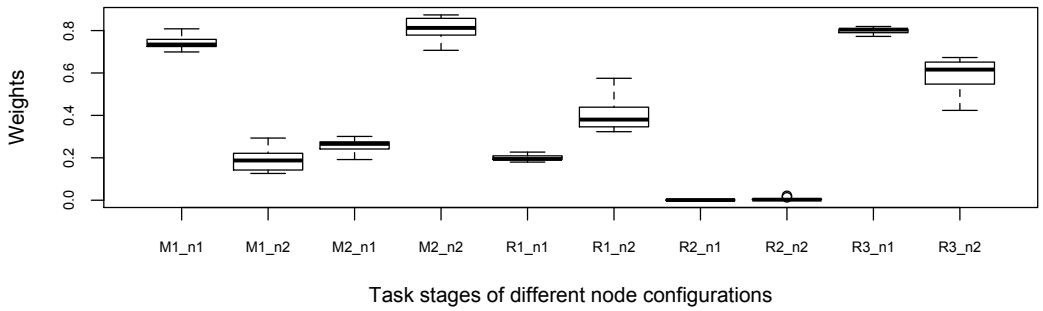
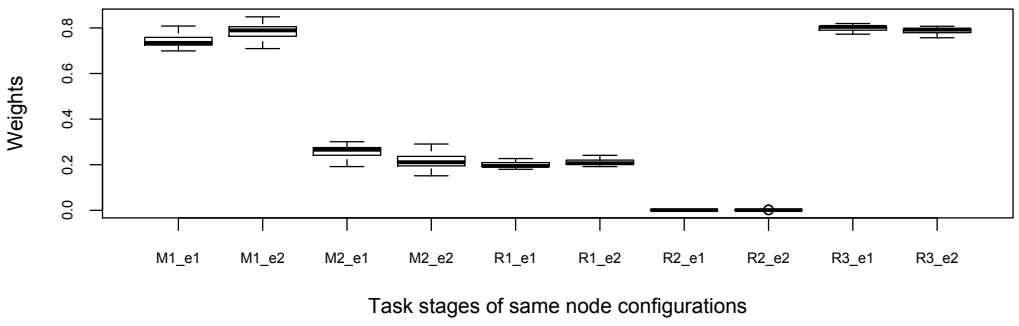


Table 3.4: Results of t-test on same condition

Stage	First execution	Second execution	P-value
m1	0.75	0.78	0.06
m2	0.25	0.22	0.06
r1	0.20	0.21	0.15
r2	0.01	0.01	0.52
r3	0.79	0.78	0.14

Figure 3.4: Weights comparison of executing WordCount 10GB jobs on a node



Chapter 4

ESAMR algorithm

In Chapter 2, we have presented the disadvantages of LATE and SAMR algorithms. In LATE, it uses fixed stage weights of map and reduce tasks to estimate TimeToEnd of each task. In SAMR, although it uses historical information to find better stage weights of map and reduce tasks to estimate TimeToEnd of each task, it does not consider the fact that the dataset sizes and the job types can also affect the stage weights of map and reduce tasks.

In this chapter, we present our new Enhanced Self-adaptive MapReduce (ESAMR) scheduling algorithm. This algorithm is designed to overcome the shortcoming of SAMR algorithm by taking into account many factors that could impact the stage weights. The main step taken by ESAMR is to classify the historical information stored on each TaskTracker node into k clusters using a machine learning technique. In the map phase, ESAMR records a job's temporary $M1$ weight based on its map tasks completed on the node and uses the temporary $M1$ weight to find the cluster whose average $M1$ weight is the closest. Then, the cluster's stage weights on the node will be used for the job to estimate its map tasks' TimeToEnd on that node. In the reduce phase, ESAMR carries out a similar procedure. it uses temporary $R1$ and $R2$

weights to find the cluster with the closest reduce stage weights. ESAMR then utilizes these stage weights to estimate TimeToEnd of the job's reduce tasks on the node and identify slow tasks. After a job has finished, ESAMR calculates the stage weights of map and reduce tasks on each TaskTracker node, and saves these new weighs as a part of the historical information. Finally, ESAMR applies K-means, a machine learning algorithm, to re-classify the historical information stored on each TaskTracker node into k clusters and saves the average stage weights of each of the k clusters. By utilizing more accurate stage weights to estimate TimeToEnd of each task, ESAMR can identify slow tasks more accurately than SAMR and LATE algorithms.

Table 1 gives the pseudo code of ESAMR algorithm. Section 4.1 presents the definition of two important parameters: Percentage of Finished Map tasks (PFM) and Percentage of Finished Reduce tasks (PFR). Section 4.2 presents how to use historical information to find the closest combination of stage weights for a running job on each TaskTracker node. Section 4.3 describes how to find slow tasks and section 4.4 discusses how to find slow TaskTracker nodes. Section 4.5 describes the K-means algorithm used in ESAMR.

Algorithm 1 ESAMR

Require:

PFM (Percentage of Finished Map Tasks), a threshold used to control when to begin the slow map task identification

PFR (Percentage of Finished Reduce Tasks), a threshold used to control when to begin the slow reduce task identification

history, historical information of the K clusters, where each record of a cluster contains 5 values, $M1$, $M2$, $R1$, $R2$ and $R3$

threshold, a variable for selecting slow tasks

Main Procedure

if a job has completed PFM of its map tasks **then**

  $M1$= **CalculateWeightsMapTasks**

  $M2$=1-$M1$

end if

if a job has completed PFR of its reduce tasks **then**

  $< R1, R2 >$= **CalculateWeightsReduceTasks**

  $R3$=1-$R1$-$R2$

end if

$slowTasks$= **FindSlowTask**

run backup tasks for $slowTasks$

if a job has finished **then**

  run $K-means$ algorithm to re-classify historical information into k clusters

end if

Procedure CalculateWeightsMapTasks

if a node has finished map tasks for the job **then**

  calculate $tempM1$ based on the job's map tasks completed on the node

  $M1$=randomly chosen first stage weight $M1$ from the corresponding node's *history*

  $beta$=abs($tempM1$-$M1$)

  for each $M1[i] \in$ *the node's history*, i=1.2,...,K **do**

    if $abs(M1[i]$-$tempM1) < beta$ **then**

      $M1=M1[i]$

      $beta=abs(tempM1$-$M1[i])$

    end if

  end for

  return $M1$

else

  $M1=\sum_{i=1}^{K} M1[i]$/K

  return $M1$

end if

35: **Procedure CalculateWeightsReduceTasks**
36: **if** a node has finished reduce tasks for the job **then**
37: calculate $tempR1$ based on the job's reduce tasks completed on the node
38: calculate $tempR2$ based on the job's reduce tasks completed on the node
39: $< R1, R2 >$=a randomly chosen $R1$ and $R2$ pair from the node's $history$
40: $beta = abs(tempR1 - r1) + abs(tempR2 - r2)$
41: **for** each $R1[i]$ and $R2[i]$ pair in the node's $history$, i=1,2,..,K **do**
42: **if** $abs(R1[i]\text{-}tempR1)+abs(R2[i]\text{-}tempR2)<beta$ **then**
43: $R1=R1[i]$
44: $R2=R2[i]$
45: $beta=abs(R1[i]\text{-}R1)+abs(R2[i]\text{-}R2)$
46: **end if**
47: **end for**
48: return$< R1, R2 >$
49: **else**
50: $R1=\sum_{i=1}^{K} R1[i]$/K
51: $R2=\sum_{i=1}^{K} R2[i]$/K
52: return$< R1, R2 >$
53: **end if**
54:
55: **Procedure FindSlowTasks**
56: set $SlowTasks$ //a temp list to save all slow tasks
57: **for** each job that has completed PFM (or PFR) of its map (or reduce) tasks **do**
58: **for** each running task i of the job **do**
59: $ProgressScore_i$=**CalculateProgressScore**
60: $ProgressRate_i=ProgressScore_i/Tr_i$, where Tr_i is the time that has been used by the task
61: TTE_i= (1-$ProgressScore_i$)/$ProgressRate_i$
62: **end for**
63: $ATTE=\sum_{i=1}^{N} TTE_i$/N, where N is the total number of running tasks of the job
64: **for** each running task i of the job **do**
65: **if** $TTE_i\text{-}ATTE > ATTE * threshold$ **then**
66: $slowTasks$.add(ith task)
67: **end if**
68: **end for**
69: **end for**
70: return $SlowTasks$

71: **Procedure CalculateProgressScore**
72: $SubPS$=N_f/N_a, where N_f is the number of key/value pairs which have been processed in a sub-stage of a task and N_a is the total number of key/value pairs to be processed in a sub-stage of the task
73: **if** the task is a map task **then**
74: **if** the map task is on the first sub-stage **then**
75: $PS = M_1 * SubPS$
76: **else**
77: $PS = M_1+M_2 * SubPS$
78: **end if**
79: **end if**
80: **if** the task is a reduce task **then**
81: **if** the reduce task is on the first sub-stage **then**
82: $PS = R_1 * SubPS$
83: **else if** the reduce task is on the second sub-stage **then**
84: $PS = R_1+R_2 * SubPS$
85: **else**
86: $PS = R_1+R_2+R_3 * SubPS$
87: **end if**
88: **end if**
89: return PS

4.1 Percentage of Finished Map tasks (PFM) and Percentage of Finished Reduce tasks (PFR)

ESAMR sets and uses PFM and PFR to decide when to calculate temporary stage weights based on completed map and reduce tasks of a job. Eq. 4.1 shows how to use PFM in ESAMR. In Eq. 4.1, N stands for the total number of TaskTrackers in a cluster. TM stands for the total number of map tasks in a job. FM_i stands for the number of map tasks finished on each TaskTracker. Eq. 4.2 shows how to use PFR in ESAMR. In Eq. 4.2, N also stands for the total number of TaskTrackers in

a cluster. TR stands for the total number of reduce tasks in a job. FR_i stands for the number of reduce tasks finished on each TaskTracker.

$$PFM \leq \sum_{i=1}^{N} FM[i]/TM \tag{4.1}$$

$$PFR \leq \sum_{i=1}^{N} FR[i]/TR \tag{4.2}$$

Only when Eq. 4.1 or Eq. 4.2 is satisfied will ESAMR calculate the temporary map or reduce stage weights for a job.

4.2 How to use historical information and temporary information

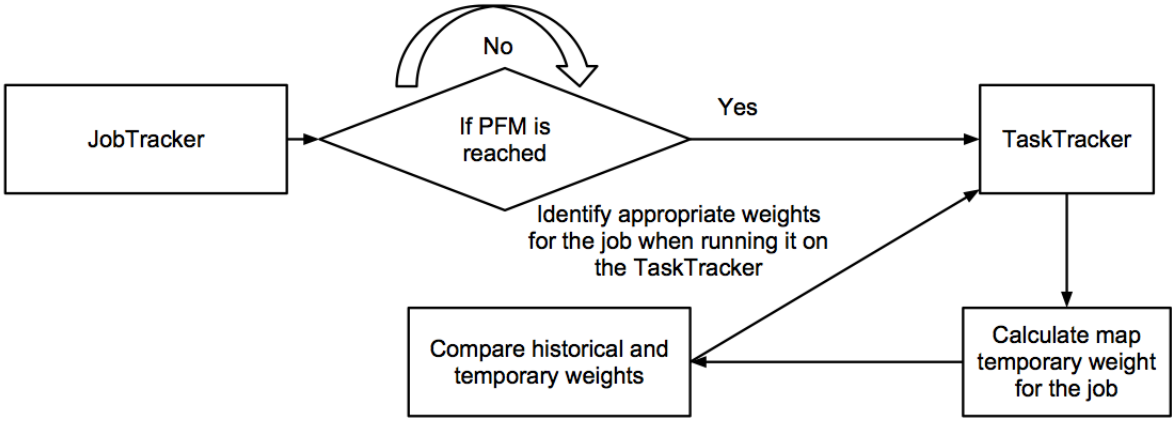
ESAMR algorithm calculates temporary stage weights, and then compares those weights with historical information to find the best combination of stage weights appropriate for the job. Figure 4.1 shows the way to use map temporary information and historical information in ESAMR. Figure 4.2 shows the way to use reduce temporary information and historical information in ESAMR.

1. JobTracker checks to see if PFM is reached. If so, ESAMR calculates the weights of the finished map tasks on each node and generates a temporary MapWeight file on each node to record the temporary $M1$.

2. Each TaskTracker reads the historical information ($M1$, $M2$, $R1$, $R2$, $R3$) that are recorded on the node.

3. ESAMR compares the information stored in the temporary MapWeight file with the historical information. As mentioned above, in the map phase, ESAMR

Figure 4.1: The way to use map temporary information and historical information



compares temporary $M1$ with the k average results of classified groups in historical information and finds the group with the closest weight and set $M1$ as the average result of that group. In addition, $M2 = 1 - M1$.

4. ESAMR utilizes these new weights of the map phase to estimate the TimeToEnd of the job's map tasks currently running on the node and find which tasks of the job are slow.

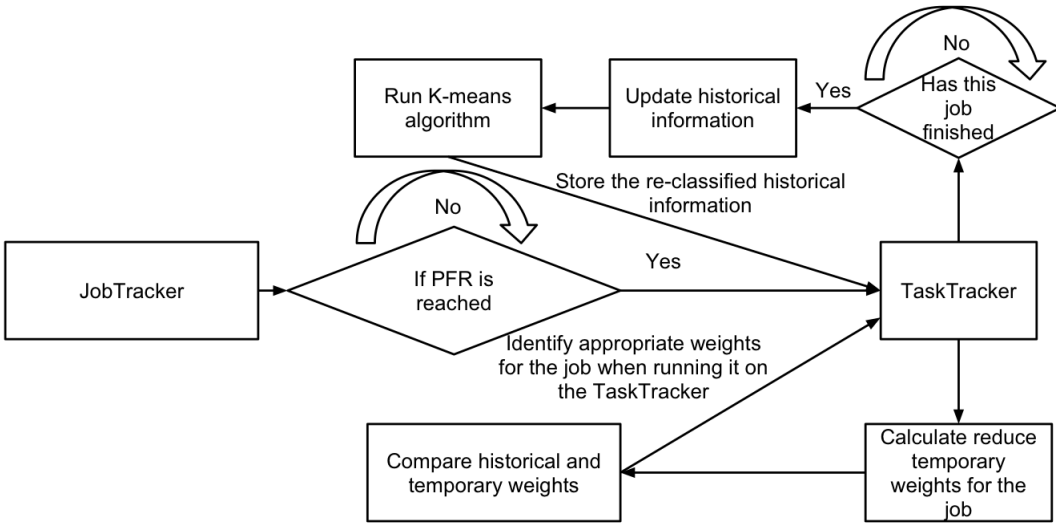
5. In the reduce phase, ESAMR checks to see if PFR is reached. If so, ESAMR calculates the stage weights of finished reduce tasks on each node and generates a temporary ReduceWeights file on each node. ESAMR compares these temporary $R1$ and $R2$ weights with k average results of classified groups, finds the group with the closest weights and sets $R1$ and $R2$ as the average results of that group. In addition, $R3 = 1 - R1 - R2$.

6. ESAMR utilizes these new stage weights of the reduce phase to estimate the TimeToEnd of the job's reduce tasks currently running on the node and find which tasks of the job are slow.

7. After a job is finished, ESAMR calculates the average stage weights of all map or reduce tasks of the job that were completed on the node, and generates a new combination of stage weights as a part of the historical information.

8. In the end, ESAMR runs the K-means algorithm to re-classify all combinations of stage weights into k clusters and store the re-classified historical information. (See Figure 4.2).

Figure 4.2: The way to use reduce temporary information and historical information



4.3 Find slow tasks

SlowTaskThreshold (STT) in the range [0,1] is used to classify tasks into fast and slow tasks. If the TimeToEnd of the ith task (TTE_i) (see Eq. 2.5) and the average TimeToEnd of all running tasks ($ATTE$) fulfill Eq. 4.3, the ith task is judged to be a slow task. Suppose for a job, the number of currently running map or reduce tasks is N. $ATTE$ is computed according to Eq. 4.4.

$$TTE_i - ATTE > ATTE * STT \tag{4.3}$$

$$ATTE = \sum_{j=1}^{N} TTEj/N \tag{4.4}$$

According to Eq. 4.3, if STT is too small (close to 0), ESAMR will classify some fast tasks to be slow tasks. If STT is too large (close to 1), ESAMR will classify some slow tasks to be fast tasks. Therefore, we need to choose an appropriate value for STT.

ESAMR schedules tasks as follows. First, all the TaskTrackers obtain tasks from a queue of new MapReduce jobs. Then, a TaskTracker computes $TimeToEnd$ for all tasks running on it. Next, ESAMR finds which map or reduce tasks are slow and put them in a slow task queue. When the queue of new jobs is empty, a TaskTracker tries to fetch tasks from the slow task queue and launch backup tasks for them. However, Only when the TaskTracker is not a slow node, can it launch backup tasks.

4.4 Find slow tasktracker nodes

SlowNodeThreshold (SNT) in the range [0,1], is used to classify TaskTrackers into fast TaskTrackers and slow TaskTrackers. This value is important because it protects ESAMR against launching a speculative task on a node that is slow but happens to have a free slot when ESAMR needs to make a scheduling decision. Launching a task on a slow node does not help and also means that we cannot re-execute the task on any other node because ESAMR allows only one speculative copy of each task to run at any time.

Suppose there are N TaskTrackers in the system. The progress rate of the ith TaskTracker node is TR_{m_i} for map task and TR_{r_i} for reduce task, and the average progress rate of all TaskTrackers for map task is ATR_m, and for reduce task is ATR_r. If there are M map tasks and R reduce tasks running on the ith TaskTracker node, TR_{m_i}, TR_{r_i}, ATR_m and ATR_r can be computed according to Eqs.(4.5), (4.6), (4.7), and (4.8). Progress rate PR_k in Eqs. (4.5) and (4.6) is calculated according to Eq. (2.4).

$$TR_{m_i} = \sum_{k=1}^{M} PR_k/M \tag{4.5}$$

$$TR_{r_i} = \sum_{k=1}^{R} PR_k/R \tag{4.6}$$

$$ATR_m = \sum_{i=1}^{N} TR_{m_i}/N \tag{4.7}$$

$$ATR_r = \sum_{i=1}^{N} TR_{r_i}/N \tag{4.8}$$

For the ith TaskTracker, if it fulfills Eq.(4.9), it is a slow map TaskTracker. If it fulfills Eq.(4.10), it is a slow reduce TaskTracker.

$$TR_{m_i} - ATR_m > ATR_m * SNT \tag{4.9}$$

$$TR_{r_i} - ATR_r > ATR_r * SNT \tag{4.10}$$

According to Eq.(4.9) and Eq.(4.10), if SNT is too small, ESAMR will classify some fast TaskTrackers to be slow TaskTrackers. If SNT is too large, ESAMR will

classify some slow TaskTrackers to be fast TaskTrackers. Therefore, we need to choose an appropriate value for SNT.

4.5 K-means algorithm in ESAMR

In statistics and data mining, K-means [18] [13] clustering is a method of cluster analysis. The main purpose of K-means clustering is to partition a set of entities into different clusters in which each observation belongs to a cluster with the nearest mean value.

In our K-means algorithm, ESAMR first assigns random values for the centroids (i.e., mean values) of K groups. Second, ESAMR assigns each entity to a cluster that has the closest centroid. Third, ESAMR recalculates the centroids and repeats the second and third steps until entities can no longer change groups. Table 2 gives the pseudo code of the K-means algorithm used in ESAMR.

Each TaskTracker runs K-means algorithm to classify the historical information for the node on which it is running. No additional communication between nodes is needed when reading and updating historical information and the running time of K-means algorithm is around 80 milliseconds on each TaskTracker node. So, ESAMR is scalable.

Algorithm 2 K-means

Require: E=e_1,e_2,...,e_n (set of entities to be clustered)

1: k (number of clusters)

2: $MaxIters$(Maximum number of iterations)

3:

Ensure: $C = \{c_1,c_2,...,c_k\}$ (set of cluster centroids)

4: $L = \{l(e)|e = 1, 2, ..., n\}$ (set of cluster labels of E)

5:

6: **for** $i = 1$ *to* k **do**

7: c_i=e_j (randomly select an e_j from E)

8: **end for**

9: **for** $e_i \in$ E **do**

10: $l(e_i) = argminDistance(e_i, c_j), j \in \{1...k\}$// find the cluster j whose center is nearest to an entity

11: **end for**

12: $iter = 0$

13: **repeat**

14: **for** $c_i \in$ C **do**

15: c_i= avg(e_k), for all $l(e_k) = i$

16: **end for**

17: $changed = false$

18: **for** $e_i \in$ E **do**

19: $clusterID$= $argminDistance(e_i, c_j)$, $j \in \{1...k\}$

20: **if** $clusterID \neq$ l(e_i) **then**

21: l(e_i) = $clusterID$

22: $changed = true$

23: **end if**

24: **end for**

25: $iter{+}{+}$

26: **until** $changed = false$ or $iter > MaxIters$

Chapter 5

Evaluation

To evaluate our ESAMR algorithm, we compare it with SAMR and LATE algorithms. Three metrics, weight estimation error, TimeToEnd estimation error, and identified slow tasks, are used for evaluation.

We run experiments in a cluster of 1 JobTracker and 5 TaskTrackers that are configured as a rack. The version of JDK is 1.6.0.26, and the version of Hadoop is 0.21.0. Because we cannot get the primary version of SAMR MapReduce scheduling algorithm, we implement it ourselves according to the algorithm description in [4]. Table 5.1 lists our Hadoop cluster hardware environment and configuration.

The rest of this section is organized as follows. Section 5.1 presents the best parameters of ESAMR. Section 5.2 shows the correctness of stage weights estimation and section 5.3 shows the TimeToEnd estimation error and slow tasks identified by the algorithms.

Table 5.1: Evaluation Environment

Nodes	Quantity	Hardware and Hadoop Configuration
Master node	1	2 single-core 2.2GHz Optron-64 CPUs, 6GB RAM, 1Gbps Ethernet
Data nodes A	3	2 single-core 2.2GHz Optron-64 CPUs, 4GB RAM, 1Gbps Ethernet, 2 map and 1 reduce slots per node
Data nodes B	2	2 single-core 2.3GHz Optron-64 CPUs, 2GB RAM, 100Mbps Ethernet, 2 map and 1 reduce slots per node

5.1 Best parameters of ESAMR

Before evaluating the performance of ESAMR, we should select the proper combination of parameters in ESAMR. The parameters include the Percentage of Finished Map tasks (PFM) and the Percentage of finished Reduce tasks (PFR) mentioned in section 4.1, SlowTaskThreshhold (STT) mentioned in section 4.3, SlowNodeThreshhold (SNT) mentioned in section 4.4 and the values of K in K-means algorithm mentioned in section 4.5. In order to select the proper parameters in ESAMR, we run experiments where we change one parameter while keeping all other parameters constant. In the experiments, we run Sort and WordCount benchmarks ten times each for a setting.

1. PFM: PFM represents the Percentage of Finished Map tasks for a job. ESAMR uses PFM to decide when ESAMR calculates temporary M1. We show the difference between real and estimated M1 weights when PFM is set at 5%, 10%, 20%, and 50% respectively. As shown in Figure 5.1 and Figure 5.2, setting PFM at 20% leads to a similar weight estimation error as setting PFM at 50% in WordCount. The difference between setting PFM at 20% and 50% in Sort is a little bigger than that in WordCount.

Figure 5.1: Weight Estimation Error with different PFM (WordCount)

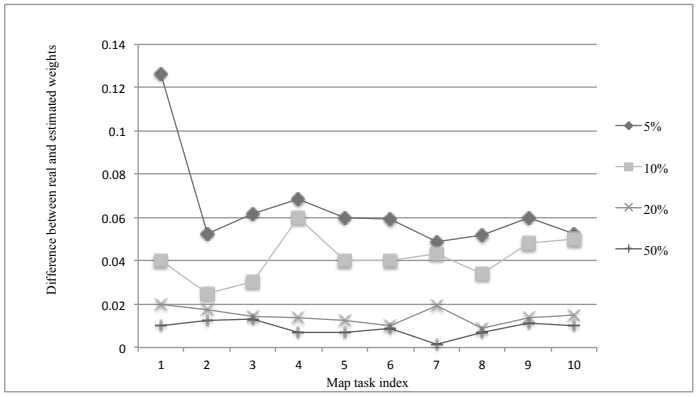
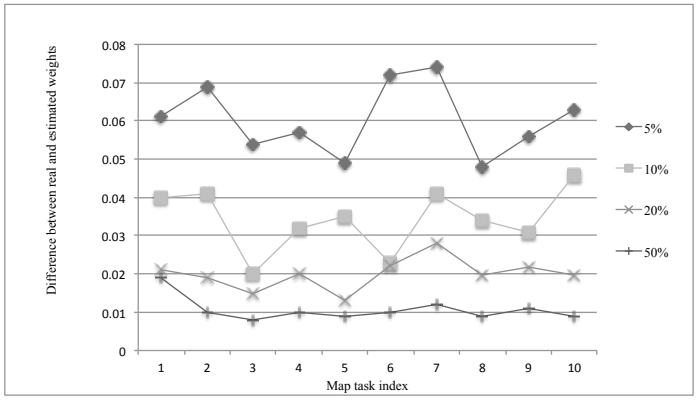


Figure 5.2: Weight Estimation Error with different PFM (Sort)



2. PFR: PFR represents the Percentage of Finished Reduce tasks for a job. ESAMR uses PFR to decide when ESAMR calculates temporary weights of the reduce phase. We show the difference between real and estimated R1 weights when PFR is set at 5%, 10%, 20%, and 50% respectively. As shown in Figure 5.3 and Figure 5.4, we can see that setting PFR at 10% leads to a similar estimation error as PFR at 20% in WordCount and Sort. The difference between setting PFR at 20% and 50% is not significant either.

3. The value of K in the K-means algorithm: ESAMR uses the parameter K to decide how many clusters are partitioned by the K-means algorithm. We show the differences between real and the estimated M1 weights when K is at 2, 4,

Figure 5.3: Weight Estimation Error with different PFR (WordCount)

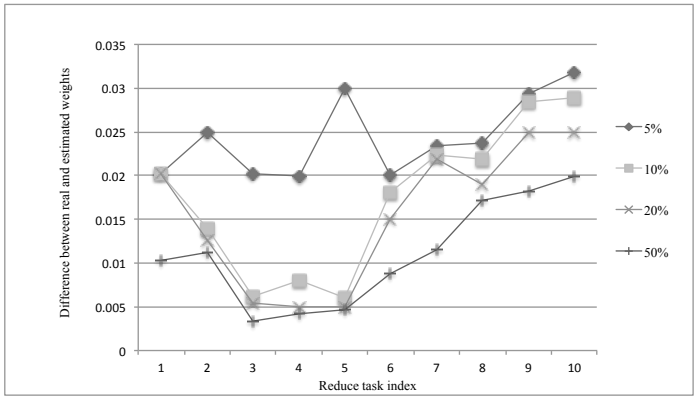
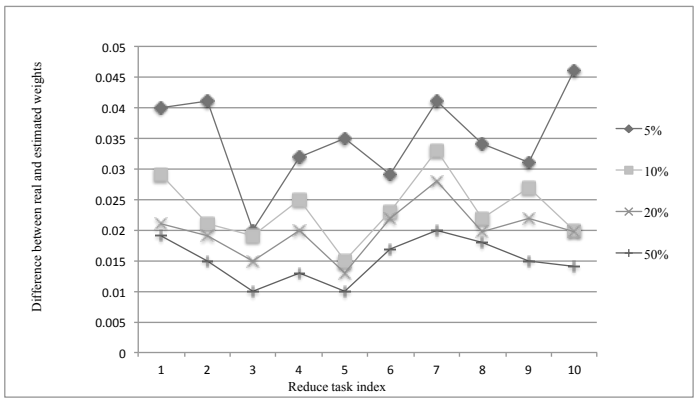


Figure 5.4: Weight Estimation Error with different PFR (Sort)



10, and 20 respectively. As shown in Figure 5.5 and Figure 5.6, the difference between the real M1 and the estimated M1 is smaller than 0.1 when K is set at 10 in WordCount and Sort.

4. STT: STT is a parameter used to find slow tasks according to Eq. (4.3). From Eq. (4.3), we know if we set STT too large, ESAMR classifies some slow tasks to be fast tasks and vice versa. In Figure 5.7 , we show the resultant MapReduce job execution time when setting STT at 10%, 20%, 30%, 40%, 50%, and 60% respectively. From Figure 5.7, we see that the job execution time first decreases, then increases, with increasing STT. This is because ESAMR considers a fewer number of tasks to be slow tasks with the increase of STT according to Eq.(4.3).

Figure 5.5: Weight Estimation Error with different K (WordCount)

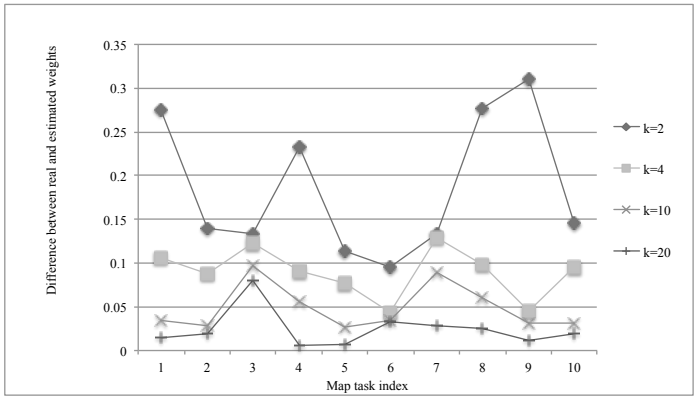
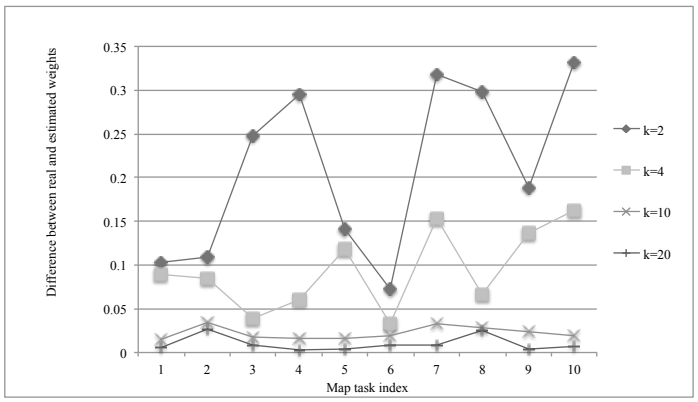


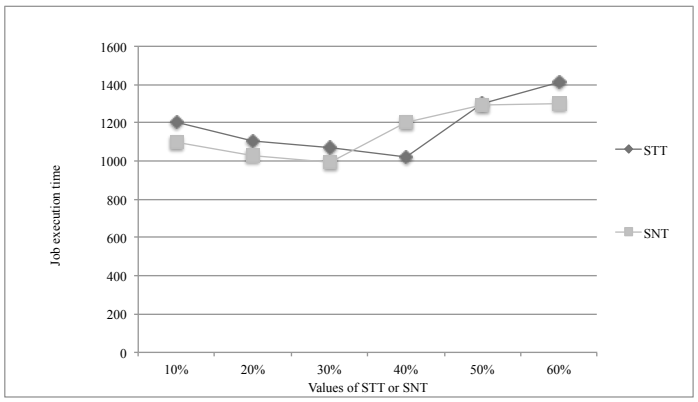
Figure 5.6: Weight Estimation Error with different K (Sort)



When STT is smaller than 0.4, ESAMR considers several fast tasks to be slow. Backup tasks of these fast tasks consume a great deal of system resources, so the job execution time is prolonged. On the other hand, when STT is larger than 0.4, some very slow tasks are considered to be fast. These slow tasks will prolong the job execution time as well. We thus set STT to be 0.4 in the following experiments.

5. SNT: SNT is a parameter used to find slow TaskTracker nodes according to Eqs. (4.9) and (4.10). As shown in Figure 5.7 , the job execution time is the shortest when SNT is set at 0.3. This is because ESAMR considers a fewer number of TaskTrackers to be slow TaskTrackers with the increase of SNT

Figure 5.7: Algorithm effectiveness with different STT and SNT



according to Eqs. (4.9) and (4.10). When SNT is smaller than 0.3, ESAMR considers several fast TaskTrackers to be slow. As a result, the system resources that can be used are limited. When the SNT is larger than 0.3, some slow TaskTrackers are considered to be fast. Consequently, backup tasks may run on these slow TaskTrackers, so the execution time cannot be shortened. We thus set SNT to be 0.3 in the following experiments.

Since, the same equations are used to identify slow tasks and slow TaskTrackers in ESAMR, SAMR and LATE algorithms, we choose the same parameter values for all three algorithms. HP is an important parameter for SAMR [4]. We set HP at 0.2, since experiments show setting HP at 0.2 can achieve the best performance [4] for SAMR.

5.2 Correctness of weights estimation

In order to verify the correctness of the weights of the map and reduce phases estimated by ESAMR, we list the weights of map and reduce phases estimated by ESAMR and the actual weights of map and reduce phases collected from the system in Table 5.2 . Because M1+M2 = 1 and R1+R2+R3 =1, we chose M1, R1 and

R3 to show the result. As shown in Table 5.2, for either map task or reduce task, the weights of the map and reduce stage estimated by ESAMR are not far from the real weights collected from the system. However, all stage weights are far from the constant weights(1,0,1/3,1/3,1/3) used in LATE algorithm. In Table 5.3, we can see that the differences between the weights of the map and reduce stages estimated by SAMR and the real weights are bigger than the differences between the weights estimated by ESAMR and the real weights. The reason for this is that SAMR does not differentiate different types of jobs and jobs with different sizes of datasets.

Table 5.2: Weights estimated by ESAMR vs Real Weights of a WordCount 10GB job

Node Name	**M1**	**R1**	**R3**
Node 1	0.7261/0.7198	0.1926/0.1901	0.8062/0.8078
Node 2	0.7633/0.7502	0.1917/0.1899	0.8072/0.8090
Node 3	0.6200/0.6109	0.2060/0.2079	0.7920/0.7814
Node 4	0.2142/0.2078	0.3647/0.3699	0.6327/0.6284
Node 5	0.2062/0.2012	0.3954/0.3894	0.6028/0.6012

Table 5.3: Weights estimated by SAMR vs Real Weights of a WordCount 10GB job

Node Name	**M1**	**R1**	**R3**
Node 1	0.9563/0.7902	0.5717/0.2247	0.4248/0.7747
Node 2	0.2942/0.8074	0.5839/0.2332	0.4116/0.7661
Node 3	0.9487/0.7836	0.5683/0.1794	0.4276/0.8197
Node 4	0.8241/0.5922	0.4990/0.3395	0.4513/0.5549
Node 5	0.8164/0.4071	0.6949/0.2960	0.2916/0.6948

5.3 TimeToEnd Estimation and Slow Task Identification

In order to evaluate the performance of ESAMR, we compare the TimeToEnd estimation of the three MapReduce scheduling algorithms by running Sort and WordCount applications ten times each. The three algorithms are LATE, SAMR and ESAMR. We set PFM at 20%, PFR at 20%, K at 10, STT at 40%, and SNT at 30% respectively.

Figures 5.8 and 5.10 show the the TimeToEnd estimation error of map and reduce tasks by ESAMR, SAMR and LATE on a WordCount 10GB job (Figures 5.9 and 5.11 show the ratios of Figures 5.8 and 5.10 respectively). From Figures 5.8 and 5.10 we can see that the effectiveness of ESAMR. Among the three algorithms ESAMR leads to the smallest prediction error. With ESAMR, the differences between estimated and actual TimeToEnd of map and reduce tasks are less than 4 and 5 seconds respectively. With SAMR, the differences between estimated and actual TimeToEnd of map and reduce tasks are less than 38 and 27 seconds respectively. With LATE, the differences between estimated and actual TimeToEnd of map and reduce tasks are less than 64 and 129 seconds respectively.

Figures 5.12 and 5.14 show the TimeToEnd estimation error of map and reduce tasks by ESAMR, SAMR and LATE on a Sort 10GB job (Figures 5.13 and 5.15 show the ratios of Figures 5.12 and 5.14 respectively). From Figures 5.12 and 5.14, we can see that ESAMR still has the smallest error, but LATE has a better performance than SAMR from the first task to the seventh task on the map and reduce phases. The reason is that the default weights of map and reduce phases used in LATE is close to the real weights of the map and reduce phases than SAMR which still uses historical information from running the WordCount job to estimate the weights of the map and reduce phases. From the eighth task to the twentieth task, SAMR has

Figure 5.8: Map tasks TimeToEnd estimation error (WordCount 10GB)

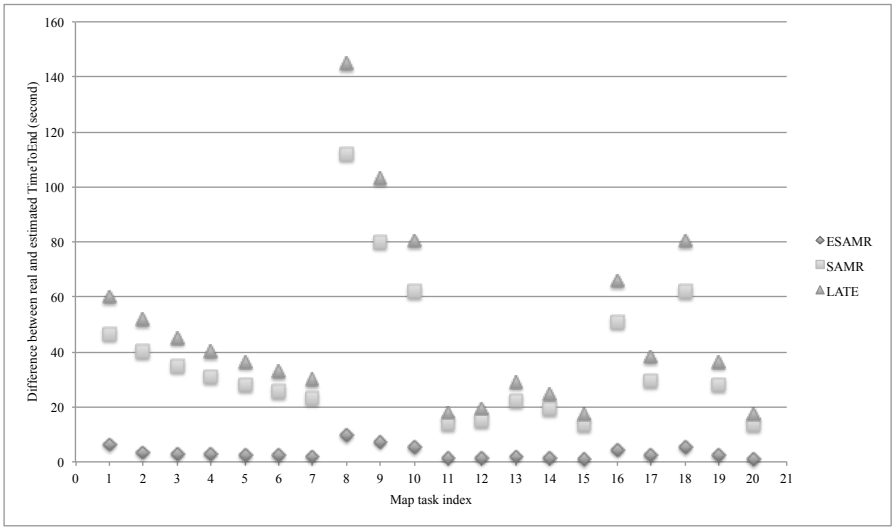
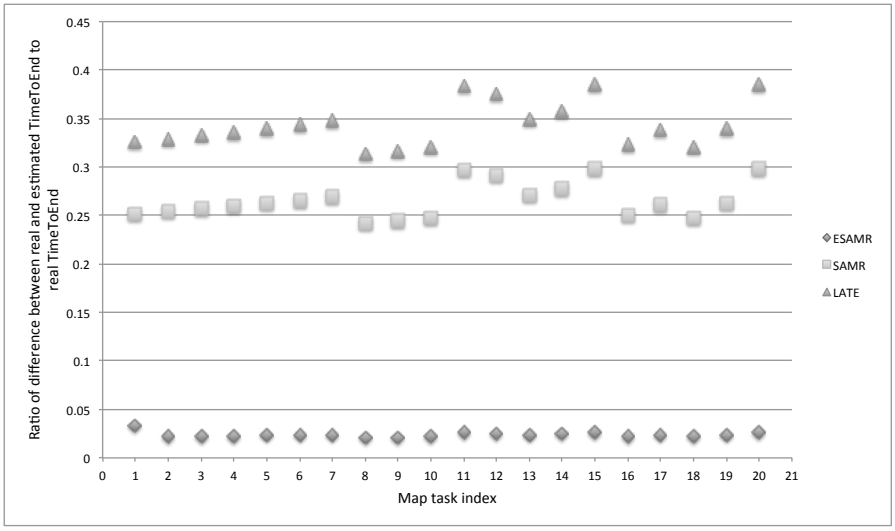


Figure 5.9: Map tasks TimeToEnd estimation error ratio (WordCount 10GB)



a better performance than LATE. The reason is that SAMR begins to use historical information from running the Sort job to estimate the weights of map and reduce phases, but LATE still uses fixed weights for the map and reduce phases. So SAMR's estimation of TimeToEnd become more accurate than LATE's. With ESAMR, the differences between estimated and actual TimeToEnd of map and reduce tasks are less than 0.77 and 3 seconds respectively. With SAMR, the differences between estimated

Figure 5.10: Reduce tasks TimeToEnd estimation error (WordCount 10GB)

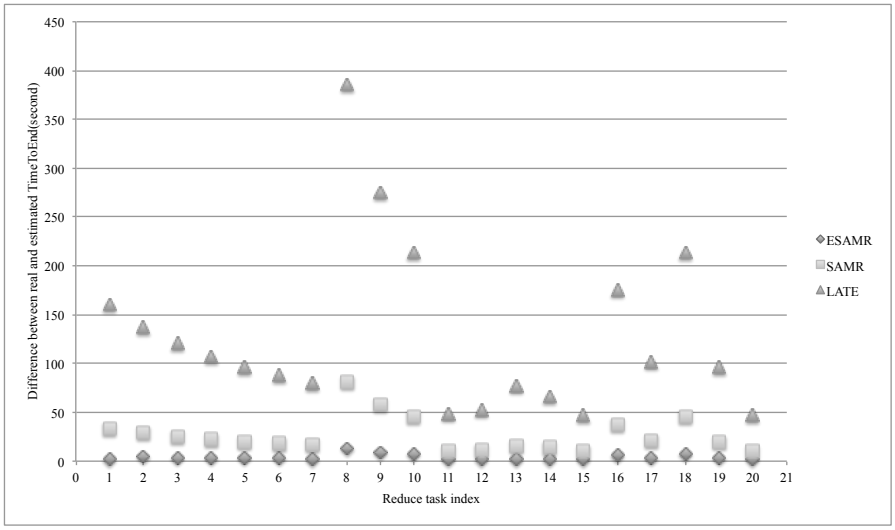
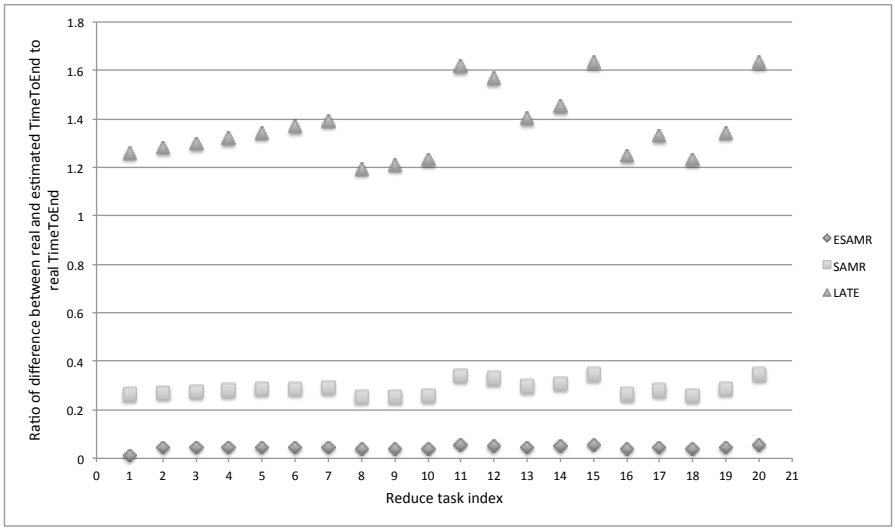


Figure 5.11: Reduce tasks TimeToEnd estimation error ratio (WordCount 10GB)



and actual TimeToEnd of map and reduce tasks are less than 14 and 83 seconds respectively. With LATE, the differences between estimated and actual TimeToEnd of map and reduce tasks are less than 27 and 139 seconds respectively.

To evaluate if these algorithms differ statistically, we performed ANOVA analysis on the data sets of the three algorithms for a significant level of 0.05. Table 5.4 reports the results. As the p-values in the rightmost column show all p-values are less than

Figure 5.12: Map tasks TimeToEnd estimation error (Sort 10GB)

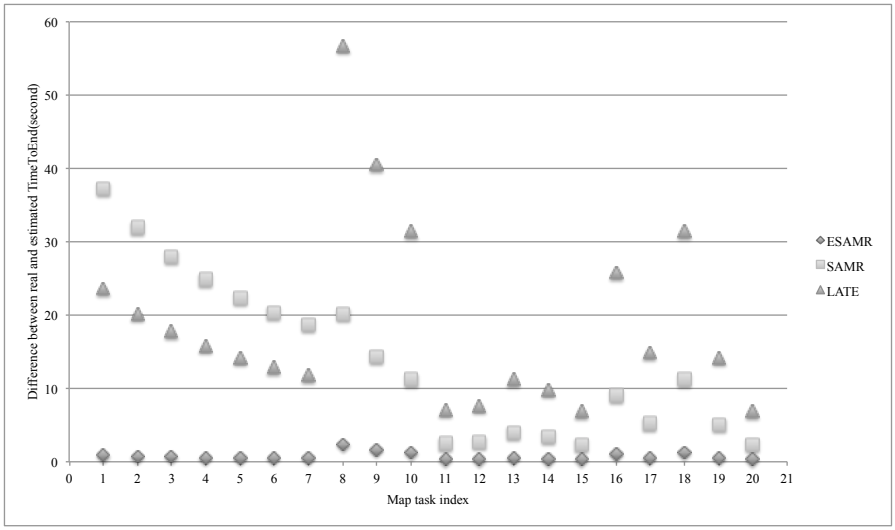
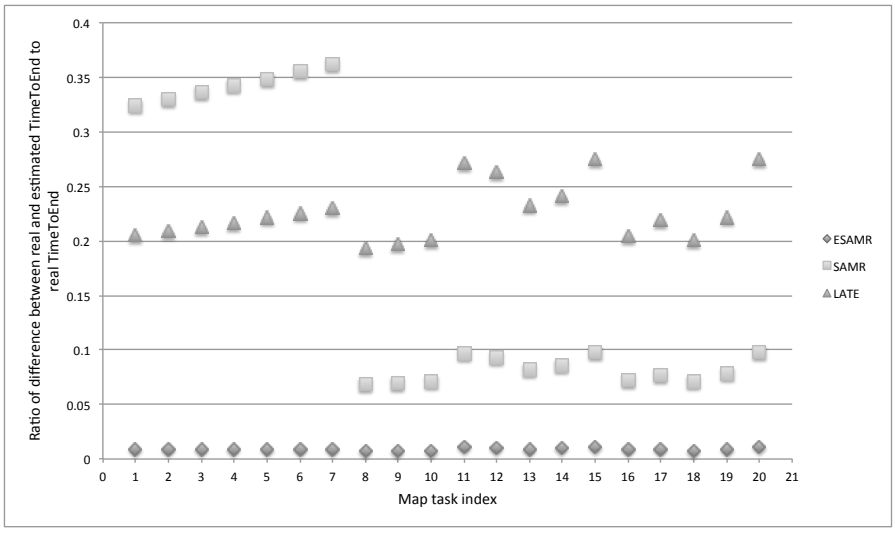


Figure 5.13: Map tasks TimeToEnd estimation error ratio (Sort 10GB)



0.05, there is enough statistical evidence to reject the null hypothesis on all cases, indicating that the mean of the differences between real TimeToEnd and estimated TimeToEnd achieved by the three algorithms are significantly different in statistics.

The ANOVA analysis is used to evaluate whether these algorithms perform differently, and a multiple comparison procedure using Bonferroni analysis quantifies how the datasets differ from each other. Table 5.5 presents the results of this analy-

Figure 5.14: Reduce tasks TimeToEnd estimation error (Sort 10GB)

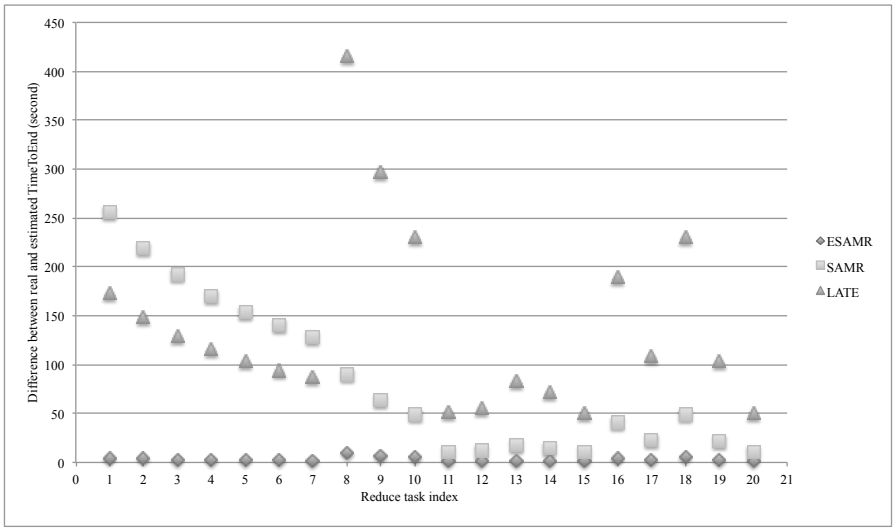
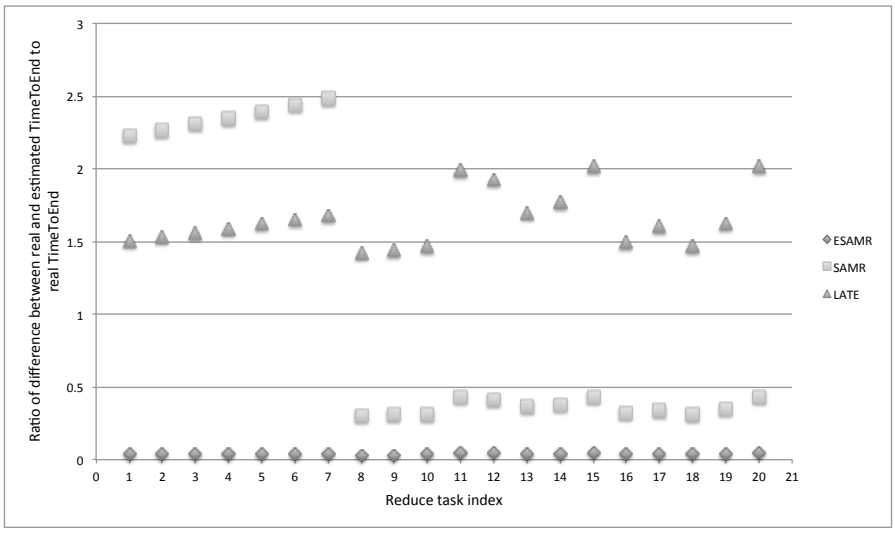


Figure 5.15: Reduce tasks TimeToEnd estimation error ratio (Sort 10GB)



sis, ranking the datasets by their means in an ascending order. Grouping letters (in columns with the header "Gr") indicate the degree of differences: datasets with the same grouping letter were not significantly different in statistics. For all the data, ESAMR has the datasets with the smallest means while LATE and SAMR are in a single group, which means ESAMR estimates significantly more accurately than LATE and SAMR .

Table 5.4: Results of ANOVA Analysis

	Df	Sum Sq	Mean Sq	F value	Pr
Map WordCount	2	22259	11129.3	19.341	3.879e-07
Reduce WordCount	2	178247	89124	33.434	2.472e-10
Map Sort	2	4464.1	2232.1	40.364	1.203e-11
Reduce Sort	2	188394	94197	18.605	6.034e-07

Table 5.5: Results of Bonferroni Means Test

WordCount					
	Mean Map	Gr		Mean Reduce	Gr
ESAMR	3.41	A	ESAMR	4.17	A
SAMR	37.64	B	SAMR	27.43	B
LATE	64.64	B	LATE	129.65	B
Sort					
ESAMR	0.77	A	ESAMR	3.26	A
SAMR	13.87	B	SAMR	83.95	B
LATE	27.06	B	LATE	139.77	B

Figure 5.16 shows the execution time estimated by ESAMR, SAMR and LATE. There are 100 map tasks in each job. For convenience, We chose the first 20 map tasks to show the performance of ESAMR, SAMR and LATE in selecting speculative tasks. 20 map tasks is enough to show the difference and performance of the three algorithms. From Figure 5.16, we see that ESAMR considered the 8th and 9th map tasks as the slowest tasks. SAMR chose the 10th map task as the slowest task. LATE chose the 1st map tasks as the slowest tasks. The real slowest tasks were the 8th and 9th map tasks. Only ESAMR identified the slow tasks correctly.

Figure 5.17 shows the execution time estimated by ESAMR, SAMR and LATE. There are 20 reduce tasks in each job. We use all 20 reduce tasks to show the performance of ESAMR, SAMR and LATE in selecting speculative tasks. From Figure 5.17, we see that ESAMR estimated the 1st reduce task as the slowest task. SAMR estimated the 8th reduce task as the slowest task. LATE estimated the 8th and 9th reduce tasks as the slowest tasks. The real slow tasks was the 1st reduce task. ESAMR is the only algorithm to find the correct slow reduce task.

Figure 5.16: Real and estimated execution time of map tasks for a WordCount 10GB job

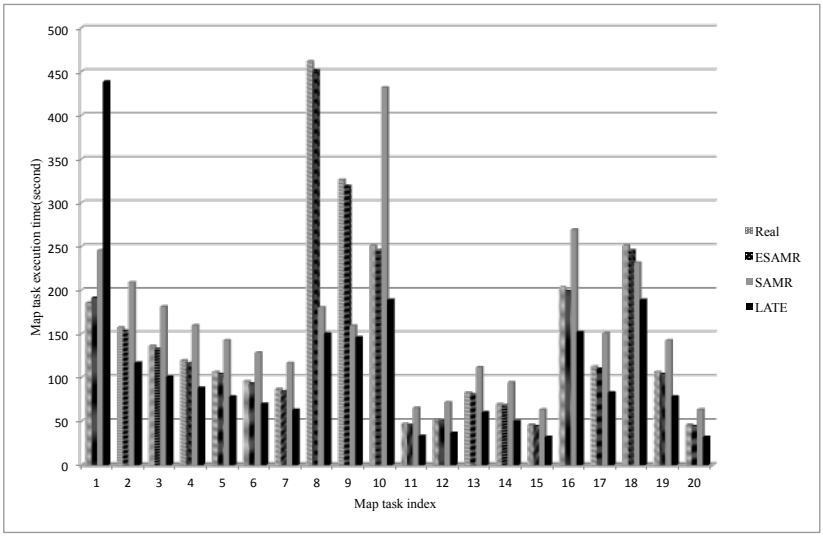
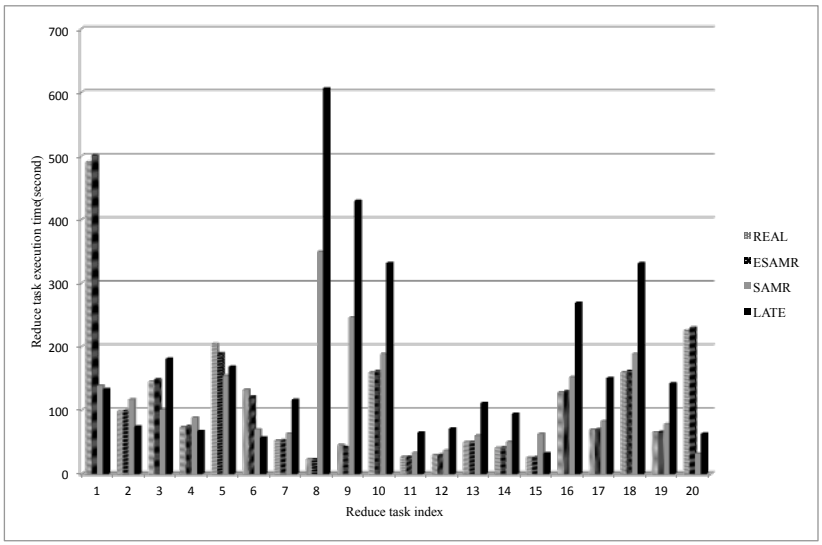


Figure 5.17: Real and estimated execution time of reduce tasks for a WordCount 10GB job



Chapter 6

Conclusion

To overcome the limitations of existing MapReduce scheduling algorithms, we have proposed ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm in this master thesis, which uses K-means clustering algorithm to classify historical information into K clusters and thus generates more accurate estimation of task's stage weights especially in heterogeneous environments to correctly identify slow tasks and re-execute them. Experimental results have shown the effectiveness of ESAMR.

Bibliography

[1] Greg Barish. Speculative plan execution for information agents. Technical report, 2003.

[2] Luiz Andr 233; Barroso, Jeffrey Dean, and Urs H246;lzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7*, pages 205–218, 2006.

[4] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. *Computer and Information Technology, International Conference on*, 0:2736–2743, 2010.

[5] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53:72–77, January 2010.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazons highly available key-value store. In *IN PROC. SOSP*, pages 205–220, 2007.

[9] Peter R. Elespuru, Sagun Shakya, and Shivakant Mishra. Mapreduce system over heterogeneous mobile devices. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 168–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[11] Hadoop. `http://hadoop.apache.org`.

[12] Apache hadoop. `http://en.wikipedia.org/wiki/Apache_Hadoop`.

[13] Greg Hamerly and Charles Elkan. Alternatives to the k-means algorithm that find better clusterings.

[14] Hbase. `http://hbase.apache.org/book/book.html`.

[15] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang. Mars: A mapreduce framework on graphics processors.

[16] Chao Jin and Rajkumar Buyya. Mapreduce programming model for.net-based distributed computing.

[17] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous computing systems. In *6th Heterogeneous Computing Workshop (HCW '97*, pages 135–146, 1997.

[18] K-means. `http://en.wikipedia.org/wiki/K-means_clustering`.

[19] Robert Lupton. Statistics in theory and practice.

[20] Sathiamoorthy Manoharan. Effect of task duplication on the assignment of dependency graphs. *Parallel Comput.*, 27:257–268, February 2001.

[21] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. Supporting mapreduce on large-scale asymmetric multi-core clusters. *SIGOPS Oper. Syst. Rev.*, 43:25–34, April 2009.

[22] Michael C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[23] Executive Summary. A platform 2015 workload model recognition , mining and synthesis moves computers to the era of tera introduction : The coming era of tera the first step : Computers need. *Synthesis*, pages 1–8, 2005.

[24] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A dynamic mapreduce scheduler for heterogeneous workloads. *Grid and Cloud Computing, International Conference on*, 0:218–224, 2009.

[25] Yahoo hadoop tutorial. `http://public.yahoo.com/gogate/hadoop-tutorial/starttutorial.html`.

[26] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[27] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[28] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial queries evaluation with mapreduce. *Grid and Cloud Computing, International Conference on*, 0:287–292, 2009.