IMPROVING PERFORMANCE IN HADOOP MAPREDUCE

By

ADEMOLA CHUKWUDI AINA

Bachelor of Science in Computer Science

Department of Computer Science / Mathematics

College of Natural and Applied Sciences

Novena University, Ogume

Delta State, Nigeria.

2011

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2014

IMPROVING PERFORMANCE IN HADOOP

MAPREDUCE

Thesis Approved:

Dr. Johnson Thomas

Thesis Adviser

Dr. David Cline

Dr. K.M. George

ACKNOWLEDGEMENTS

My deepest gratitude goes to God who granted me the grace to do this thesis. My loving appreciation to my sweet mother for her financial and mental support and encouragement.

Also, special thanks to my thesis Supervisor Dr. J. P. Thomas for his advice, support and knowledge. I also wish to express my appreciation to Dr. Michael Buser, for his Big Data support. Likewise, I acknowledge the committee members for their support

Name: ADEMOLA CHUKWUDI AINA

Date of Degree: DECEMBER, 2014

Title of Study: IMPROVING PERFORMANCE IN HADOOP MAPREDUCE

Major Field: COMPUTER SCIENCE

Abstract: Hadoop MapReduce is a parallel, distributed programming model for processing large data sets or so-called Big data, on a cluster. The basic idea of MapReduce is to split the large input data set into many small pieces and assign these pieces to different devices for processing [5]. In this thesis, we took a look at performance evaluation of the MapReduce framework. MapReduce can be improved to perform speculative execution with maximum performance. Thus, optimizing the cost of computation and cost of communication will help achieve better performance. These optimizations are done by measuring the processing power of each machine and distributing task based on the capacity of each machine. The second step, measure he communication overheads and distribute tasks in the system for a given job or workload. To this end, we represent the Hadoop MapReduce execution with a functional model, and develop an optimization model for performance improvement in the system. Our experiments show that the proposed developed optimization functional model outperforms the regular functional model of the Hadoop MapReduce system by a factor of 2.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Many programming models on large and distributed cluster systems have emerged, among which is MapReduce. MapReduce was designed for computation that involves huge amount of data. Examples include finding the common set of element in joint tables, or finding the set of most frequent queries submitted to Google's search engine on any given day, or finding the most commonly used terms in a table or gathering of commonly used data, otherwise known as mapping data. MapReduce exploits a map function and reduce function [3, 4] in a class of code to parallelize the user program automatically and it provides the states for fault tolerance during its implementation and execution.

The MapReduce System automatically takes care of managing the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, providing for redundancy and data processing features, and overall management of the whole process. There have been many custom solutions using the MapReduce for specific problems, and many publications to evaluate the performance of the Map and Reduce phases of the Hadoop MapReduce framework [2].

Hadoop is an open source software or programming framework, mostly Java-based, that supports the processing of large data sets in a distributed computing environment. It is part of the Apache project sponsored by the Apache Software Foundation.

1.1 BACKGROUND

"MapReduce is the heart of Hadoop. It is this programming paradigm that allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster. The MapReduce concept is fairly simple to understand for those who are familiar with clustered scale-out data processing solutions" [1].

MapReduce includes two major parts, the Map function and Reduce function. The input files are automatically split and copied to different computing nodes. After that the inputs will be sent to Map function in key-value pair format. The Map function will process the input pairs and generate intermediate key-value pairs as inputs for the Reduce function. The Reduce function will combine the inputs who have the same key and generate the final result. The final result will be written into the distributed file system [5]. Hadoop is an open source software project that enables the distributed processing of large data sets across clusters of commodity servers. It is designed to scale up from a single server to thousands of machines, with a very high degree of fault tolerance.

1.2 LIMITATIONS AND BASIC ASSUMPTIONS OF HADOOP MAPREDUCE

- **Batch Processing and Not Interactive:** Hadoop assumes that nodes perform work at roughly the same rate but that is not true in a virtualized environment, that is, if there is a heterogeneity in the processing elements. MapReduce is a batch

based algorithm. There are continuous optimizations on the MapReduce algorithm being proposed from different open source software providers.

- **Lunching Tasks:** There is no cost to launching a task on a node that has an idle slot. One problem with the Hadoop system is that by dividing the tasks across many nodes, it is possible for a few slow nodes to rate-limit the rest of the program. Tasks may be slow for various reasons, including hardware degradation, or software mis-configuration, but the causes may be hard to detect since the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another, equivalent, task as a backup. This is termed speculative execution of tasks.

Communication Costs: Communications costs in MapReduce framework are not considered or assumed to be negligible. However, when measuring the performance of the MapReduce framework communication contributes to the performance.

1.3 MOTIVATION

Given the limitations identified above, in this thesis we propose an approach to:

(a) Estimate the cost of computation for a MapReduce job

(b) Estimate the total cost for a MapReduce job that takes into consideration both the computation and communication costs

(c) Maximize performance when executing speculative tasks.

1.4 THESIS OUTLINE

Chapter 2 covers the literature review and related works on the MapReduce framework. It identifies the key features of the MapReduce. Chapter 3 describes our proposed model to improve the performance of the MapReduce framework. Chapter 4 presents the experiments to validate the model we have proposed in chapter 3. Chapter 5 gives conclusions and recommendations for future work

CHAPTER II

REVIEW OF LITERATURE

In this chapter, sections 2.1 to 2.5 give a summary of the Hadoop MapReduce framework. A review of research in performance evaluation of the Hadoop MapReduce framework is provided in sections 2.6.

The MapReduce Framework is used for computation analysis of huge data [6]. It is a programming model for data processing in large-scale, where each output depends on only two inputs (that is, the Key/Value pairs). A brief overview of the framework is discussed as follows.

2.1 HADOOP MAPREDUCE

Hadoop MapReduce [7] is a software framework for writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce *job* splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master. Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*. The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the JobTracker which then assumes the responsibility of distributing

the software configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Although the Hadoop framework is implemented in Java, MapReduce applications need not be written in Java [7].

2.2 HDFS [7]

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine projects HDFS is now an Apache Hadoop subproject. The project URL is in [8].

2.2.1 STREAMING DATA ACCESS ON HDFS

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

2.2.2 LARGE DATA SETS

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

2.2.3 SIMPLE COHERENCY MODEL

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

2.2.4 NAMENODES AND DATANODES – HDFS ARCHITECTURE

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The

DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployement has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

2.3 KEY FEAUTURES OF THE MAPREDUCE SYSTEM

There are many features of the MapReduce framework but the key features are outlined below:

- Provides a framework for MapReduce execution
- Redundancy and fault tolerance is built-in, so the programmer doesn’t have to worry about errors
- The MapReduce programming model is language independent

MapReduce facilitates automatic parallelizable and distributed execution, while enable local data processing. It also manages inter-process communication.

2.4 PHASES IN MAPREDUCE

The Major phases in the MapReduce are the "Map" Phase and the "Reduce" Phase but there are intermediate phase between the Map and Reduce phase. Basically, the "*Map*" and "*Reduce*" are like programming Functions in a class of codes. Assuming you are developing a MapReduce job, That is, a program to count the total number of words in a file and group them in MapReduce fashion. The class will be called "*WordCount*", and a "*Map*" Function will be invoked to splits the input data-set into independent chunks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the "*Reduce"* tasks. The process of sorting the output of the maps is the intermediate functions which can be called the "*Copy & Sort*" phase. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks [2].

The phases of the framework can simply be summarized as follows:

- **Map phase**: user defined function applied to input data - Is a Function that takes a key/value pair and produces an intermediate key/value pair. (k1, v1) → list (k2, v2)
- **Copy phase**: task fetches map outputs
- **Sort/Combining phase**: map outputs are sorted by key and group the output
- **Reduce phase**: user-defined function is applied to the list of map outputs with each key - Is a Function that takes a key and a list of key values and outputs the

final key/value pair. Basically, aggregate the result from map phase after the intermediate phase (copy phase and sort phase). (k2, list (v2)) → list (k3, v3)

The MapReduce System receives the result from the Reduce phase, making it accessible in the Hadoop Distributed File System (HDFS).

2.5 PRACTICAL ASPECTS OF MAPREDUCE

The physically file size of an HDFS cluster 64/128 Memory – Blocks. There are different files system ranging from the Native file system, Hadoop Distributed File System (HDFS), and the Cloud. The most commonly known and used cloud at is the AWS (that is, Amazon cloud system). The Output in any Hadoop MapReduce jobs are immutable. Any Common Object Oriented Programming language such as – C++, Java, Python, Javascript, and many more others can be used for implementation.

2.6 MAPREDUCE EXAMPLE

A simple example is presented in [6]. A CSV/Text file that contains English words from a dictionary and all translations in other languages added to it, separated by a '|' symbol. The MapReduce job will read dictionaries of different languages and match each English word with a translation in another language. In this example the class code is built basically of three parts. A static class holds the mapper, the other static class holds the reducer, and the main method works as the driver to the application. More details, including implementation of this application can be found at [9, 6].

2.7 PERFORMANCE EVALUATION OF MAPREDUCE

There have been some studies into the performance of Hadoop MapReduce. According to [12], the “Hadoop MapReduce, is slower than two state-of-the-art parallel database systems in performing a variety of analytical tasks by a factor of 3.1 to 6.5. MapReduce can achieve better performance with the allocation of more compute nodes from the cloud to speed up computation; however, this approach of renting more nodes" is not cost effective in a pay-as-you-go environment. Users desire an economical elastically scalable data processing system, and therefore, are interested in whether MapReduce can offer both elastic scalability and efficiency”. Likewise, [13] states that the Hadoop’s scheduler can cause severe performance degradation in heterogeneous environments. A new algorithm called LATE (Longest Approximate time to End) was designed in [13] which was believed to be highly robust to heterogeneity and improve the Hadoop response time by a factor of 2 in a clusters of 200 virtual machines on Amazon’s Elastic Compute Cloud (EC2). The authors claim that Hadoop's performance is closely tied to its task scheduler, and this algorithm claims to improve the scheduler of the Hadoop MapReduce framework. Using Amazon's EC2 as an example, the scheduler decides when to speculatively re-execute tasks that appear to be stragglers in a homogeneous cluster where tasks progress linearly,

Some efforts have been made to improve the performance of Hadoop using job scheduling or job parameter optimization. [15] Proposes an approach to improve the performance of the Hadoop MapReduce framework by optimizing the job and task execution mechanism. This approach:

1. Reduced the cost in time during job initialization and job termination by setting up and cleaning tasks of MapReduce
2. Introduced an instant messaging communication mechanism for accelerating performance-sensitive task scheduling and execution. This is instead of using the loose heartbeat-based communication mechanism to transmit all messages between the JobTracker and TaskTrackers,
3. Implemented SHadoop, an optimized and fully compatible version of Hadoop that aims at shortening the execution time cost of MapReduce jobs, especially for short jobs. Experimental results show that compared to the standard Hadoop, SHadoop can achieve stable performance improvement by around 25% on average for comprehensive benchmarks without losing scalability and speedup. This optimization work has passed a production-level test in Intel and has been integrated into the Intel Distributed Hadoop (IDH) [15].

Estimating the completion time of MapReduce programs as a function of a new dataset and the cluster resources are given in [16]. The emphasis is on a benchmarking approach for designing a MapReduce performance Model. [14] Mentioned that the Hadoop MapReduce system works with the parameter configuration space in the Hadoop MapReduce. This parameter configuration space is a huge aspect of MapReduce whereby the job parameter can be tuned. The challenge lies in MapReduce job parameter tuning which is a daunting and time consuming task. There are more than 70 parameter configurations that impact job performance. Thus, it is a challenge to systematically explore the parameter space and select a near-optimal configuration. Hence, it was proposed in [14] that an online performance tuning system called MRONLINE would

improve performance by 30% more than the default configuration YARN. MRONLINE monitors a job's execution, tunes associated performance-tuning parameters based on collected statistics, and provides fine-grained control over parameter configuration.

Somewhat similar to our model is the data fetching mechanism proposed in [17], on improving MapReduce performance by data fetching in heterogeneous or shared environments. This mechanism fetches data to corresponding compute nodes in advance. The mechanism is implemented and evaluated on Hadoop-1.0.4. According to [17], results on real applications show that the data prefetching mechanism can reduce data transmission time by up to 94%. The load balancing for data placement as proposed in [17], addresses the problem of how to place data across nodes in a way that each node has a balanced data processing load. Given a data-intensive application running on a Hadoop MapReduce cluster, this data placement scheme adaptively balances the amount of data stored in each node to achieve improved data-processing performance. Using experimental results on two real data-intensive applications show that the proposed data placement strategy improves the MapReduce performance by rebalancing data across nodes before performing a data-intensive application in a heterogeneous Hadoop cluster.

2.8 OPTIMIZATION TECHINIQUES

There are many different ways to optimize MapReduce. It is critically important because of the huge volumes of data and we want to get an optimal performance in the system. There may sometimes be resource constraints. There may also be time constraints In the Hadoop cluster, there are many configuration settings that can be adjusted.

The focus in our work on MapReduce Jobs. Optimization can be achieved at different stages of the MapReduce job. We can do optimization before a job runs, optimization while loading the data, optimization during the Map phase of the job, which often includes breaking a complex mapping task into multiple jobs, optimization at the shuffle phase of the job, Optimization at the Reduce phase of the Job, and post processing or optimization of the job after the job completes. There may be adjustments or preprocessing of the incoming file to filter out most commonly unvalued or junk data. In addition incoming files may be compressed.

CHAPTER III

FUNCTIONAL MODEL

In this chapter, we propose a functional model to represent a regular MapReduce job execution. We also propose an optimization of the functional model. The optimization of the functional model is achieved by optimizing the cost of computation and cost of communication to improve the performance of executing a task or a job. First, these optimizations are done by measuring the processing power of each machine and thereby distribute tasks based on the capacity of each machine. Second, measure and optimize the communication in the system for a given job or workload. Then, we represent the Hadoop MapReduce execution with a functional model, and develop an optimization model for performance improvement in the system. Our experiments as shown in chapter 4, using the proposed developed optimization functional model show that our approach outperforms the regular functional model of the Hadoop MapReduce system by a factor of 2.

3.1 FUNCTIONAL MODEL FOR REGULAR MAPREDUCE

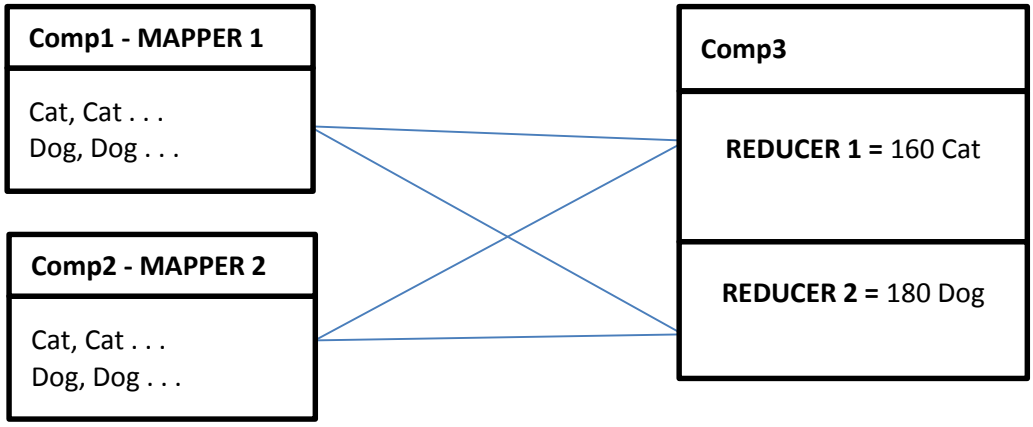

Figure 3.1 – MapReduce framework without considering communication or processing power

In figure 3.1, 'Comp' stands for Computer System, 'Cat' and 'Dog' stands for different words, in a word count example.

In our model, our goal is to make results available at a single site. However, the data sources may be at different machines. Hence, map tasks are running on different machines, and all reduce tasks are running on one machine. The results from the map functions running on multiple machines in parallel are sent to the reducer function which is running on one machine.

From Figure 3.1, the performance of any Mapper M_i *and* Reducer R_i on a machine is:

$$perf(M_i) = CPU\ Time$$

This is the CPU time for the Map phase of a task on a mapper M_i. The result of the mapper are sent through the network to the Reducer.

$perf(R) = Max(CPU\ Time\ (R_i, \dots, R_j))$ is the CPU time of Reducer nodes used during the reducer phase. The worst case performance is considered.

The CPU Time is measured in units such as milliseconds (ms)

The performance measure of task *i* that takes into account the mapper task on a machine M_i and the reducer tasks is given as:

$$perf(P_k) = \frac{(D_i)}{perf(M_i) + perf(R)} \quad (1)$$

Where P_k is the performance of task *k* measured in kb/sec

For any number of reducers and mappers in our model, the total time to execute a task is the workload (D_i) divided by the multiplication of both performance and the assigned workload (D_i) of the given machine. In other words, it is the inverse of the performance.

That is, response time

$$Rt(P_k) = \frac{(D_i)}{perf(P_i)(D_i)} = \frac{1}{perf(P_i)} \quad (2)$$

This model is based on the Hadoop MapReduce assumptions, that is, tasks in the same category (map or reduce) require roughly the same amount of work. For example in figure 3.1, consider a pre-defined input of a workload text file containing the words 'cat'

and ‘dog’ where the goal of the program is a simple *Word Count*. There are 2 Mapper nodes and 2 Reducers on a single computing Machine (total computers = 3). The mappers and reducers are working in parallel.

Thus, for optimum performance the following constraint must be true for both the Mappers and Reducers:

$$\frac{1}{perf(P_1)} \approx \frac{1}{perf(P_2)} \approx \ldots \approx \frac{1}{perf(P_i)} \quad (3)$$

From equation (1), the lower the workload, the higher the performance of the Machine.

This model has the following deficiencies as it does not consider the impact on performance caused by straggler machines. There are two reasons why a machine becomes a straggler machine. Firstly, because it has limited processing power. Secondly, because it spends too much time in communications. This leads to unbalanced processing which results in reduced performance.

Our goal is to realize optimal performance that considers straggler machines and communication costs by adjusting the workload.

3.2 FUNCTIONAL MODEL WITH COMMUNICATION AND PROCESSING POWER

To reduce the effect of straggler machines and thereby the consequent impact on load balancing, we take into consideration the processing power and limit the communication taking place in the system. This will improve the performance of the system.

First, we measure the processing power of each computer machine assigned to the Mapper and distribute tasks based on the capacity of that machine, in other to avoid any process becoming a straggler. Hence, we assign task based on the processor power of a given machine. This is represented as:

$$Po_1 < Po_2 < Po_3 < \ldots . Po_i$$

The processing power of Machine 1 (Po_1) is less than that of Machine 2 (Po_2) and Machine 2 (Po_2) is less than Machine 3 (Po_3).

The processor power is measured as the CPU speed of that machine. This can be obtained from the system properties. Therefore, total processing power Pr_T for all machines is:

$$Pr_T = \sum_{i=1}^{n} CPU_i = Cpu_1 + Cpu_2 + Cpu_3 + \ldots + Cpu_n$$

Where Cpu_i is the system processor speed measured in megahertz (MHz) or gigahertz (GHz).

For each computer with a Mapper node M_i, the workload D_i assigned to it is:

$$D_i = \frac{Cpu_i}{\sum_{i=1}^{n} CPU_i} \; x \; D \tag{4}$$

Where D is the total workload to be distributed among the Mapper.

$\sum_{i-1}^{n} CPU_i$ is the total processing power of mapper machines

Another reason why a machine could become a straggler is if the machine is spending too much time in communication rather than actually processing the task. We measure communication time with a timer variable called Ground Communication (GC) which is

the elapsed time of the process. The GC elapsed time is the monitor timer set and configured in the Map-Reduce program segment. The GC elapsed time counter increments only when the Reducer triggers a call to the Mapper and vice versa. As shown in section 4.4 later, the last communication takes place between the Mapper and Reduce right after the "Merged Map outputs", before the GC elapsed time is recorded. This shows that the GC elapsed time counter stops when there is no more communication between the Mapper and the Reducers. Thus, we measure Communication as the maximum of the GC elapsed time for a particular task assigned to a particular machine.

C_i = *GC Elapsed Time (ms)*

Where C_i denotes the communication of a task and its unit of measurement is *ms*

We modify the first model to generate the new model.

- Let $D_1,..., D_i$, $M_1,..., M_i$, $R_1,..., R_j$, be as defined earlier in section 3.1
- Let C_{ij} be the communication time between Mapper i and Reducer j. This is measured as kb/sec

Recall that the Mapper is on a machine that is separate from the reducers. The Reducer machine receives the result from the Mapper node on another machine and generates the output. Thus, the function performance model with communication is the CPU time of the Mapper machine and the time spent by the Reducer machine. Let Pt_i denote the processing time of the CPU of the Mapper i

$perf(M_i) = CPU\ Time\ (Pt_i)$ where $perf(P_i)$ be as defined earlier in section 3.1

From (3), for optimum performance the following constraint must be true for both the Mapper and Reducer, considering communication:

$$\frac{(D_i)}{perf(P_i)(D_i)} \text{ x Max } \left(C_{1ij}, C_{2ij} \ldots C_{Nij}\right)$$

$$= \frac{1}{perf(P_i)} \text{ x Max } \left(C_{1ij}, C_{2ij} \ldots C_{Nij}\right) \quad (5)$$

Note that the model is almost the same as our previous model where the results of P_i differ due to the distribution of workload using equation (4). Eq (5) considers the computational cost as well as the added overhead of communication cost. The added overhead of communication cost is not considered in eq. (3).

Continuing with our example in figure 3.1, the extended model with communication is shown below:

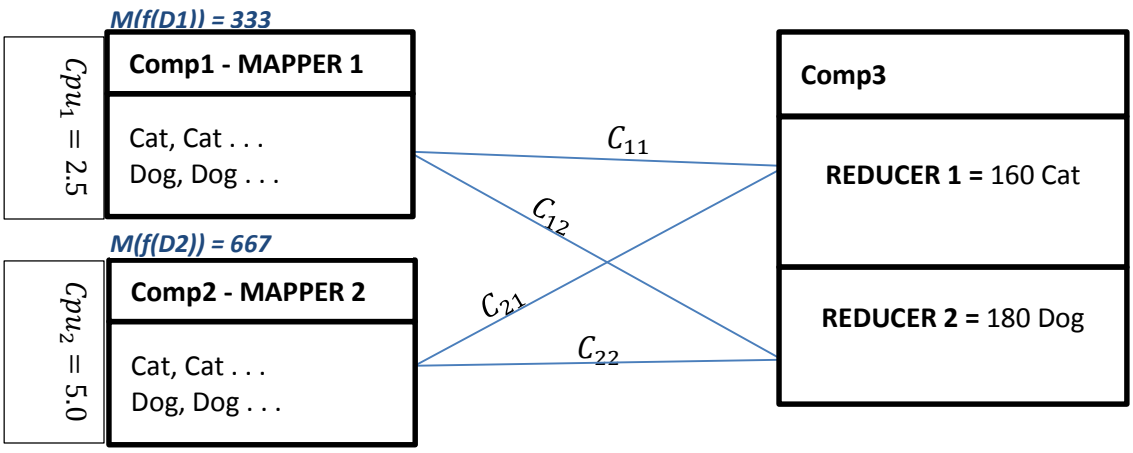

Figure 3.2 –Proposed framework communication and processing power of each machine

The CPU capacity of each machine differs. Thus, for optimum performance the following constraint must be true for both Mappers and Reducers:

$$\frac{(D_1)}{perf(P_1)(D_1)} \times \text{Max}\,(C_{11}, C_{12}) \approx \frac{(D_2)}{perf(P_2)(D_2)} \times \text{Max}\,(C_{21}, C_{22}) \approx$$

$$\ldots \approx \frac{(D_i)}{perf(P_i)(D_i)} \times \text{Max}\left(C_{1ij}, C_{2ij} \ldots C_{Nij}\right) \quad (6)$$

Since the process is in parallel the maximum communication overhead is considered, as shown in example 3.3 later.

The goal of the model is to help improve the performance of the MapReduce system by optimizing the communication overheads in the system.

3.3 COMMUNICATION PERFORMANCE DEGRADATION

Communication may degrade performance for different reasons. These possibilities includes the Mapper and Reducer spending more time communicating rather than processing the task. This situation arises as a result of the nature of task distribution and the goal of the task or network issues during processing of tasks. If there is a lot of network delay during processing, communication could be very slow and thus, leads to degradation in the performance of the job.

For instance, from figure 3.3 below:

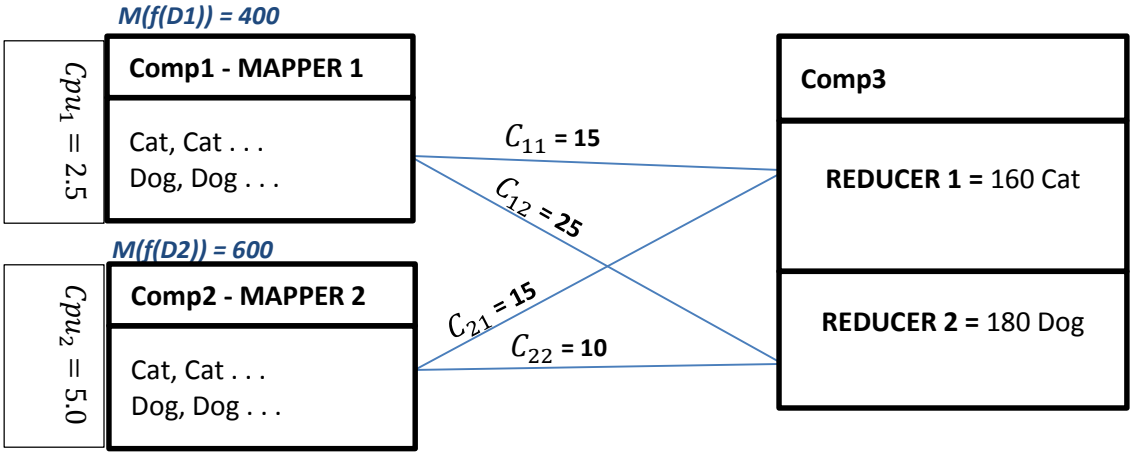

Figure 3.3 - communication degrading performance

In figure 3.3, "MAPPER 1" spends more time communicating than expected as in communication C_{12}. This is an example of a straggler processes. To resolve this, we include an additional factor into the communication model known as "Performance Degradation" which is denoted by - PD

Thus, from Figure 3.3, the total communication between Mappers and Reducers are:

Mapper 1 to Reducer 1 and Mapper 1 to Reducer 2 = *Max (15, 25)* = ***25*** and $(D_1) = 400KB$

Mapper 2 to Reducer 1 and Mapper 2 to Reducer 2 = *Max (15, 10)* = **15** and $(D_2) = 600KB$

To optimize performance, load balancing is achieved by regaining balance in the system, For every Mapper M_i in a straggler state due to excess communication, reduce the workload, (D_i) – assigned to that Mapper and denote the reduction as PD (Performance Degradation). Then, add PD to any communicating task that has the minimum communication overhead.

By default, $PD = 0$; but if there exists a straggler, we compute PD as:

$$PD_i = \Delta C \; X \; D_i \tag{7}$$

where:

$\Delta C =$ rate of change in communication (percentage).

$D_i =$ Workload of the machine to be reduced.

For optimum performance the following constraint must be true for the Mappers and Reducers:

$$\frac{(D_1) - (PD_1)}{perf(P_1)(D_1)} \times \text{Max}\,(C_{11}, C_{12}) \approx \frac{(D_2) + (PD_1)}{perf(P_2)(D_2)} \times \text{Max}\,(C_{21}, C_{22})$$

$$\ldots \approx \frac{(D_i) \pm (PD_i)}{perf(P_i)(D_i)} \times \text{Max}\left(C_{1ij}, C_{2ij \ldots} C_{Nij}\right) \tag{8}$$

Continuing with our example in figure 3.3, we can see that Mapper 1 spends more time communicating than Mapper 2.

Taking this as an example, hypothetically, let $perf(P_1) = 10 \; and \; perf(P_2) = 10$ and rate of change in communication which was measured in percentage is: $(40 - 25)\%$.

This means:$PD = (40 - 25)\% \; X \; 400 = 90$

$$\text{Machine 1: } = \frac{(D_1) - (PD)}{perf(P_1)(D_1)} \times C_1 = \frac{310}{10 \; X \; 400} \times 0.040$$

$$\text{Machine 2} := \frac{(D_2) + (PD)}{perf(P_2)(D_2)} \times C_2 = \frac{690}{10 \; X \; 690} \times 0.025$$

Thus, from the calculations, we see that the constraint specified by eq.(8) is satisfied. Hence, we distribute the workload to create a workload balance. This results in achieving close to optimal performance.

CHAPTER IV

EXPERIMENTS AND RESULTS

This chapter describes our experiments to validate the functional model presented in the previous chapter. A comparative analysis of experimental results with a regular MapReduce system and the proposed optimized functional model with communication on the Hadoop MapReduce system is presented. The experiments show that the optimized functional model outperformed the regular MapReduce by a factor of 2 in terms of performance.

The physically file size of an HDFS cluster is 64/128 Memory – Blocks. We used the Hadoop Distributed File System (HDFS) for our experiments. The outputs of any Hadoop MapReduce jobs are immutable. The Java programming language is used for our experiments.

We ran three sets of experiments. In the first experiment we used standard Hadoop. This is exemplified by eq.(3). Here tasks or jobs are distributed equally to all the machines. In the second experiment we ran the optimized model that takes into consideration the processing power and communication. This model is represented by equation (6) in chapter 3. In this model, based on previous communication and computational times, the workload is distributed to the different machines. This is a static model as the workload in not re-distributed. In the third experiment, as in the second one, we take into consideration both processing and communications times. However this time we use a dynamic model represented by eq. (8). If the performance degradation is above a threshold level, the remaining workload is re-distributed.

4.1 DATA SOURCE - D_i

The data source for the various experiment was obtained from the Department of Biosystem and Agricultural Engineering, College of Agricultural Sciences, Oklahoma State University, Stillwater. The data used is from an "Eastern Red Cedar" project data that is currently on-going. The Eastern Red Cedar project is used to determine the Geolocation, best fit location of resources, among others. The Eastern Red Cedar project data serves as the workload *D* to our system. This workload *D* is divided into $D_{1,}$ ... D_i smaller workloads where each D_i is a file that has been assigned to each machine in Megabytes (MB). The total workload *D* is 19132 KB and D_1 varies according to equation (4).

4.2 MAPPER AND REDUCER MACHINES – M_i, R_i

The number of Reducers might vary according to the number of Mappers and the goal/aim of the job. Our major area of focus is factoring communication and processing power into the model to show how performance can be improved once we divide this data between the machines according to processing power. 6 machines are used for our experiments with 4 Mapper nodes on independent machines and 2 Reducer nodes on 2 other separate independent machines.

4.3 EXPERIMENTAL SET-UP

4.3.1 SYSTEM CONFIGURATION

The computer systems used for running the program for this experiments are of the Following configuration:

- CPU(s): Intel Core i7-3930K, its equivalent or above
- Motherboard: ASUS P9X79 WS , its equivalent or above
- Memory: 32GB (8x 4GB) G.Skill Ripjaws X DDR3 1600, its equivalent or above
- Drives: Corsair Force3 120GB, OCZ Vertex 3 120GB, its equivalent or above
- Power Supply: Corsair AX850 850w 80 Plus Gold, its equivalent or above
- We used Cloudera to install Hadoop MapReduce. See [10] for more details. Cloudera is a GUI utility and licensed Hadoop MapReduce IDE/API Open Source distributor

These machines are interconnected together on a local area network (LAN).

4.3.2 PROGRAMMING TECHINQUES

We looked at the Mapper task and the Reducer task to fit into our model and improve the performance of Hadoop MapReduce. Some of the techniques used among others for optimizing the mapper and the MapReduce Jobs are:

1. Define a custom input format to be all strings of words
2. We work with custom input data types. That is, "TextInputFormat"
3. A custom partitioner can be defined. By default, MapReduce uses a hash partitioner and there might be situation for types of data for which a custom partition might run more efficiently. Since, we are using "TextInputFormat", the default hash partitioner was used.
4. Modified the **"jobconf"** file and implementation of the reducer and to ensure the GC elapsed time is set as the measure of communication.

In terms of sub diving tasks, the base line for any experiments is 30 to 40 seconds per Map task, depending on the goal/aim of the task. Thus, we compile the result of running a regular MapReduce job and compare it with the result from running the same Job using the optimized model.

4.4 IMPLEMENTATION

The Map module implementation for the Map function is the word counts program. . The Map function handles the Mapping of words that are frequently used in a large file and emits a key/value pair of <word, 1>. First, the mapper processes line by line through the file, as provided by the specified TextInputFormat. Second, it splits the line into tokens separated by the string separator specified; this separator by default is whitespaces used

through the StringTokenizer, and emits a key-value pair of <*word*, 1>. For the given sample input the first map emits: <employee, 1> <title, 1> <eId, 1> <city, 1>. The second map emits: <employee, 1> <client, 1> <region, 1> <city, 1>. As stated in section 2.4, the output of each mapper is passed through to the sort/combining phase. As part of the job configuration, this combining phase is part of the Reducer. The sort/combining phase is a local combiner which performs operation of a local aggregation. That is, map outputs are sorted by key and group the output. The Reduce module implementation used as the Reduce function in the experiments perform a summation of the values which are the occurrence count of each key. Thus the output of the job is: <employee, 2> < client, 1> <region, 1> <state, 1> <city, 2>. Various data sizes and partitioning schemes can be defined. In our experiments, for the regular MapReduce, we distribute the workload equally among machines. As proposed in the optimized model, equation (4) is used as a baseline for data sizes allocated to each machine.

The documents are stored in URLDirectory. Each document *i* represents D_i

- **MAPPER**

```
for each document in URLDirectory
{
        File = ReadFile();
        For each line in file
        T = tokenize(line);                    [initialize mapper]
        for each token in T
        {
        word.set(tokenizer.nextToken());       [map key/value pair]
```

output.collect(word, one); [store key/value pair]

}

}

CALL_REDUCER(Text);

[sends the result from mapper as key/value pair to reducer]

- **REDUCER**

  Values = getValuesFromMapper(); [initialize key/value pairs from mapper]

  int sum=0;

  while (values.hasNext())

  {

  sum += values.next().get(); [reduce key/values pair]

  }

  output.collect(key, new IntWritable(sum)); [store output of the reducer]

- **MAIN**

  Define input_path, output_path; [source and destination paths]

  Define setup method. [Initialize]

  Define jobconf as 'newjobconf'. [Hadoop preprocessor]

  Invoke the Hadoop jobClient. [Calls mapper]

  JobClient.runJob(conf); [execute]

4.5 RESULTS

4.5.1 EXPERIMENT 1: PERFORMANCE OF REGULAR MAPREDUCE

In the first experiment we used standard Hadoop. This is exemplified by eq.(3). Here tasks or jobs are distributed equally to all the machines.

The processing time of MapReduce with any set of input data may be affected by many factors. This includes the algorithm used during the implementation of the class code where the Map and Reduce functions are operational. Other functions that may be in this class codes includes the partition, combine and compress functions or sub-classes which also contribute to the factors affecting the MapReduce results. Some other external factors may also affect the performance of your MapReduce Job. From table 4.1 below, we see the cumulated data gathered during the experiments and its corresponding graph which depicts the performance of regular MapReduce. Thus, we can see how the performance varies. These variations can be attributed to the following factors:

- Hardware (or resources) such as CPU clock, disk I/O, network bandwidth, and memory size.

MapReduce requires the storage system to provide I/O interfaces for scanning data. Hence, during our experiment, there are two I/O modes namely: Streaming I/O and Direct I/O. According to [11], benchmarking on HDFS shows that Direct I/O outperforms the Streaming I/O by about 10%. To that end, we use direct I/O in our experiment to get the result shown in table 4.1 for regular MapReduce.

Table 4.1 – Performance of regular MapReduce

Exp	**Workload (D) KB**	(D_1) **KB**	(D_2) **KB**	(D_3) **KB**	(D_4) **KB**	P_1 **Kb/sec**	P_2 **Kb/sec**	P_3 **Kb/sec**	P_4 **Kb/sec**	$Rt(P_1)$ ***sec***	$Rt(P_2)$ ***sec***
1	**816**	204	204	204	204	9.97	5.51	10.00	5.48	0.100	0.181
2	**1092**	273	273	273	273	9.86	4.94	9.88	4.98	0.101	0.202
3	**1208**	302	302	302	302	8.86	4.67	8.90	4.69	0.113	0.214
4	**1360**	340	340	340	340	8.82	4.54	8.80	4.57	0.113	0.220
5	**1572**	378	378	3782	378	8.79	4.44	8.77	4.45	0.114	0.225
6	**2016**	504	504	504	504	7.55	4.21	7.52	4.23	0.132	0.238
7	**2448**	612	612	612	612	6.80	4.03	6.85	4.01	0.147	0.248
8	**2568**	642	642	642	642	6.30	3.88	6.33	3.86	0.159	0.258
9	**2828**	707	707	707	707	6.20	3.65	6.20	3.64	0.161	0.274
10	**3224**	806	806	806	806	5.88	3.41	5.87	3.42	0.170	0.293

Table 4.1 - Continuation of Performance of regular MapReduce.

$Rt(P_3)$ ***sec***	$Rt(P_4)$ ***Sec***	***MAX*** $(Rt(P_i))$ ***Sec***
0.100	0.182	**0.182**
0.101	0.201	**0.202**
0.112	0.213	**0.214**
0.113	0.219	**0.220**
0.114	0.225	**0.225**
0.133	0.236	**0.238**
0.146	0.249	**0.249**
0.158	0.259	**0.259**
0.161	0.275	**0.275**
0.170	0.292	**0.293**

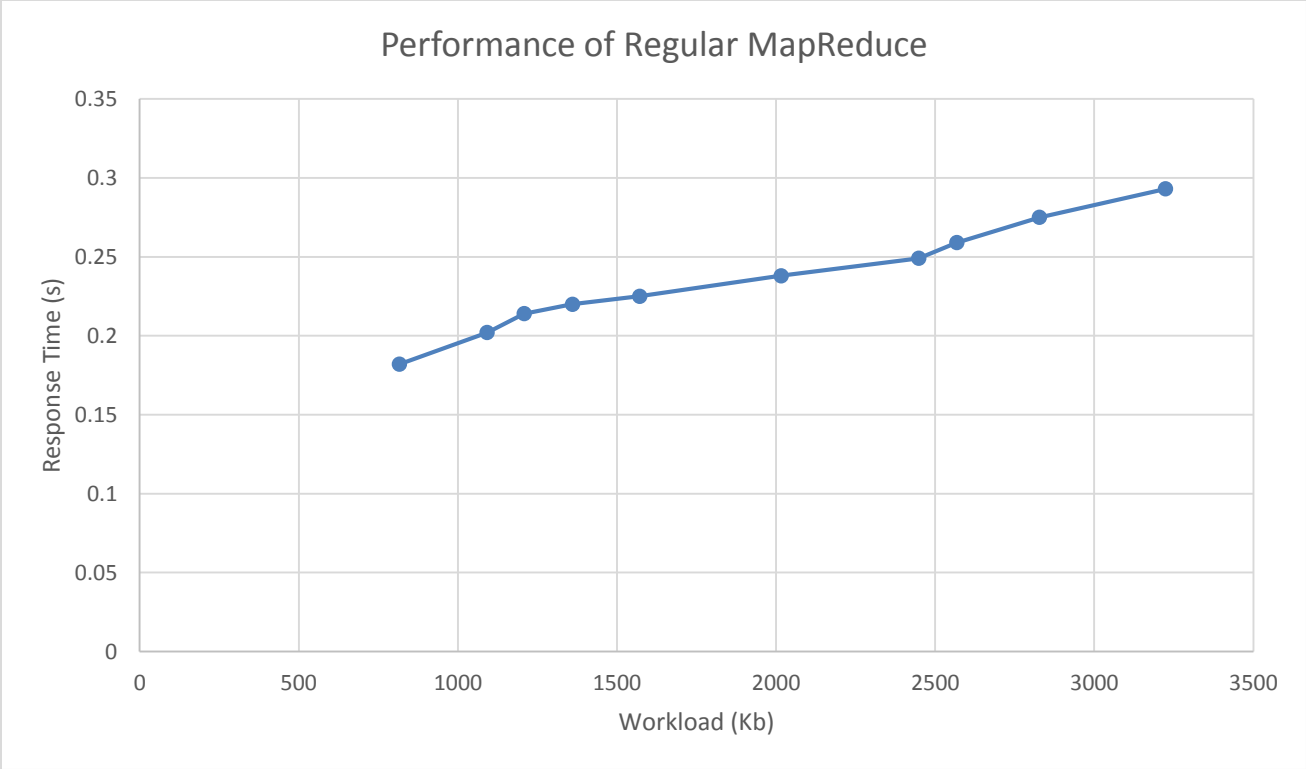

Figure 4.1 - Performance of regular MapReduce

Table 4.1 - Performance evaluation of the regular MapReduce. Where,

D_i – workload for mapper i

P_i – performance of task i

$Rt(P_i)$ – Response Time of task i

The results show that the respones time incresases almost linearly with workload.

4.5.2 EXPERIMENT 2: OPTIMIZED MAPREDUCE

In the second experiment we ran the optimized model that takes into consideration the processing power and communication. This model is represented by equation (6) in chapter 3. In this model, based on previous communication and computational times, the workload is distributed to the different machines. This is a static model as the workload in not re-distributed. The same map and reduce functions as used for the regular MapReduce was applied here also. The experimental setup was the same as for the regular MapReduce. The workload was distributed based on eq. (6) as the results are shown in table 4.2.

Table 4.2 – Performance evaluation of optmized MapReduce

Exp	**Workload (D) KB**	(D_1) **KB**	(D_2) **KB**	(D_3) **KB**	(D_4) **KB**	C_1 **(sec)**	C_2 **(sec)**	C_3 **(sec)**	C_4 **(sec)**	P_1 **Kb/sec**	P_2 **Kb/sec**	P_3 **Kb/sec**
1	**816**	136	272	136	272	0.017	0.034	0.018	0.035	13.41	13.40	13.44
2	**1092**	182	364	182	364	0.019	0.036	0.020	0.036	13.44	13.46	13.45
3	**1208**	201	403	201	403	0.020	0.039	0.022	0.040	13.46	13.47	13.47
4	**1360**	227	453	227	453	0.022	0.040	0.022	0.040	13.48	13.44	13.48
5	**1572**	262	524	262	524	0.023	0.042	0.024	0.043	13.58	13.58	13.6
6	**2016**	336	672	336	672	0.034	0.048	0.035	0.049	15.10	14.55	15.00
7	**2448**	408	816	408	816	0.036	0.051	0.037	0.053	15.23	14.81	15.23
8	**2568**	428	856	428	856	0.040	0.052	0.038	0.054	15.25	14.83	15.24
9	**2828**	471	943	471	943	0.041	0.054	0.041	0.055	15.26	15.10	15.25
10	**3224**	537	1075	537	1075	0.044	0.057	0.043	0.058	15.31	15.44	15.31

Table 4.2 – Continuation of performance of optmized MapReduce

P_4 **Kb/sec**	$Rt(P_1)$ ***sec***	$Rt(P_2)$ ***sec***	$Rt(P_3)$ ***sec***	$Rt(P_4)$ ***Sec***	***MAX*** $(Rt(P_i))$ ***Sec***
13.41	0.092	0.109	0.092	0.110	**0.110**
13.47	0.093	0.111	0.094	0.110	**0.111**
13.49	0.094	0.113	0.096	0.114	**0.114**
13.50	0.096	0.114	0.096	0.114	**0.114**
13.6	0.097	0.116	0.098	0.117	**0.117**
14.56	0.100	0.117	0.102	0.118	**0.118**
14.93	0.102	0.119	0.103	0.120	**0.120**
15.01	0.106	0.119	0.104	0.121	**0.121**
15.13	0.107	0.120	0.107	0.121	**0.121**
15.44	0.109	0.122	0.108	0.123	**0.123**

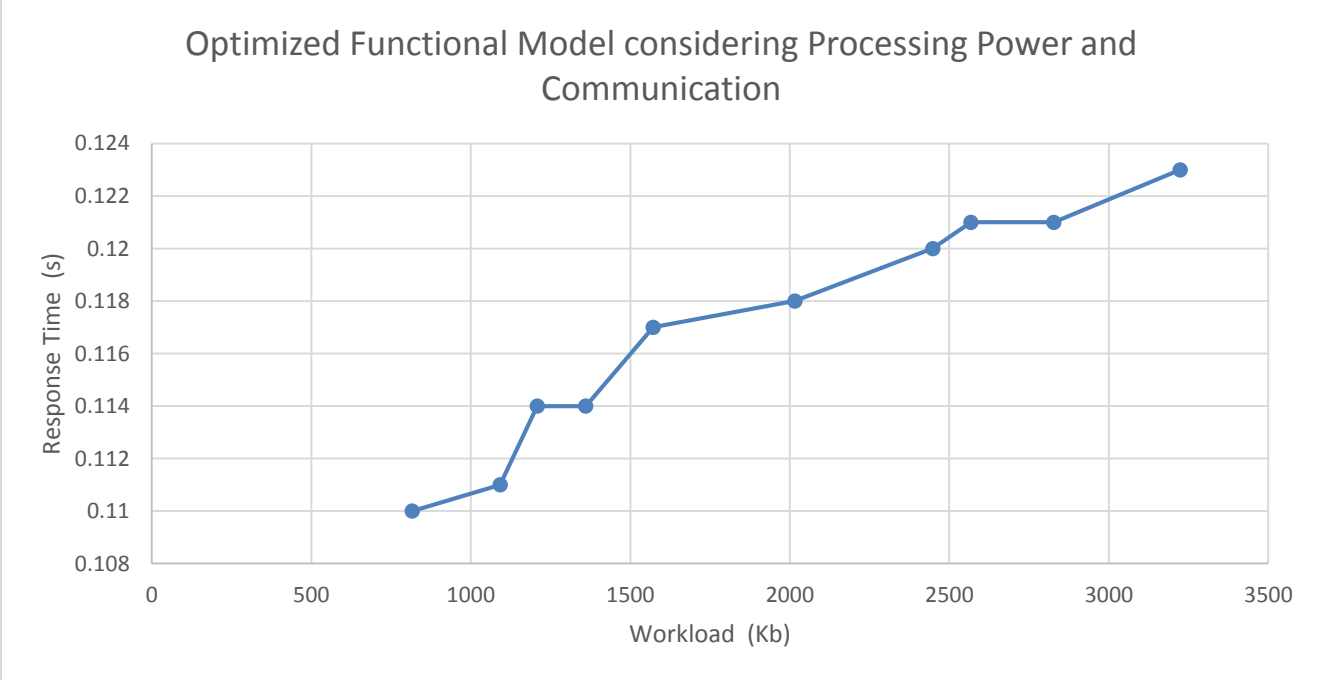

Figure 4.2 - Performance of optmized MapReduce

The result show the optimized model outperforms the regular MapReduce model by a factor of 2. Here:

D_i – workload for mapper i

C_i – Maximum communication that takes place on task i

P_i – performance of task i

$Rt(P_i)$ – Response Time of task i

4.6 EXPERIMENT 3: WORKLOAD REDISTRIBUTION IN OPTIMIZED MAPREDUCE

In the third experiment, as in the second one, we take into consideration both processing and communications times. However this time we use a dynamic model a represented by eq. (8). If the performance degradation is above a threshold level, the remaining workload is re-distributed.

From 4.1 and 4.2 above, we can see the performance improvement between the optimized model regular MapReduce jobs. This performance improvement shows that the optimized version is about 2 times faster than the regular MapReduce job. Nonetheless, in a real world situation, the optimization has to be done in real-time or on the fly. In other words, as the MapReduce job is running the performance degradation (PD) must be observed and the workload redistributed (see section 3.3). However if the workload is constantly redistributed according to eq. (8), the communication overhead will be excessive. Initially, the workload is distributed based on eq. (6).

During runtime, if the PD of a particular machine exceeds a threshold, the workload is re-distributed with a machine that has the best PD. We look at the execution log and get an estimate of the communication overheads, the time remaining for a task and the workload to be transferred. . The result is shown in table 4.3

Table 4.3 – Workload redistribution using optimized alorithm

Exp	**Workload (D) KB**	(D_1) **KB**	(D_2) **KB**	(D_3) **KB**	(D_4) **KB**	C_1 **(sec)**	C_2 **(sec)**	C_3 **(sec)**	C_4 **(sec)**	P_1 **Kb/sec**	P_2 **Kb/sec**
1	**816**	204	204	204	204	0.035	0.03	0.02	0.031	13.41	13.4
2	**1092**	273	273	273	273	0.036	0.032	0.022	0.032	13.44	13.46
3	**1208**	302	302	302	302	0.039	0.033	0.024	0.034	13.46	13.47
4	**1360**	340	340	340	340	0.044	0.035	0.046	0.036	13.48	13.44
5	**1572**	378	378	378	378	0.049	0.038	0.052	0.037	13.58	13.58
6	**2016**	504	504	504	504	0.061	0.045	0.061	0.044	15.1	14.55
7	**2448**	612	612	612	612	0.064	0.047	0.065	0.048	15.23	14.81
8	**2568**	642	642	642	642	0.067	0.048	0.068	0.05	15.25	14.83
9	**2828**	707	707	707	707	0.071	0.05	0.07	0.052	15.26	15.1
10	**3224**	806	806	806	806	0.075	0.055	0.074	0.055	15.31	15.44

Table 4.3 – Continuation of workload redistribution using optimized alorithm

P_3 Kb/sec	P_4 Kb/sec	$Rt(P_1)$ sec	$Rt(P_2)$sec	$Rt(P_3)$ Sec	$Rt(P_4)$ sec	MAX $(Rt(P_i))$ sec
13.44	13.41	0.11	0.105	0.095	0.106	**0.11**
13.45	13.47	0.111	0.107	0.097	0.107	**0.111**
13.47	13.49	0.114	0.108	0.099	0.109	**0.114**
13.48	13.5	0.119	0.11	0.121	0.111	**0.121**
13.6	13.6	0.123	0.112	0.126	0.111	**0.126**
15	14.56	0.128	0.114	0.128	0.113	**0.128**
15.23	14.93	0.13	0.115	0.131	0.115	**0.131**
15.24	15.01	0.133	0.116	0.134	0.117	**0.134**
15.25	15.13	0.137	0.117	0.136	0.119	**0.137**
15.31	15.44	0.141	0.12	0.14	0.12	**0.141**

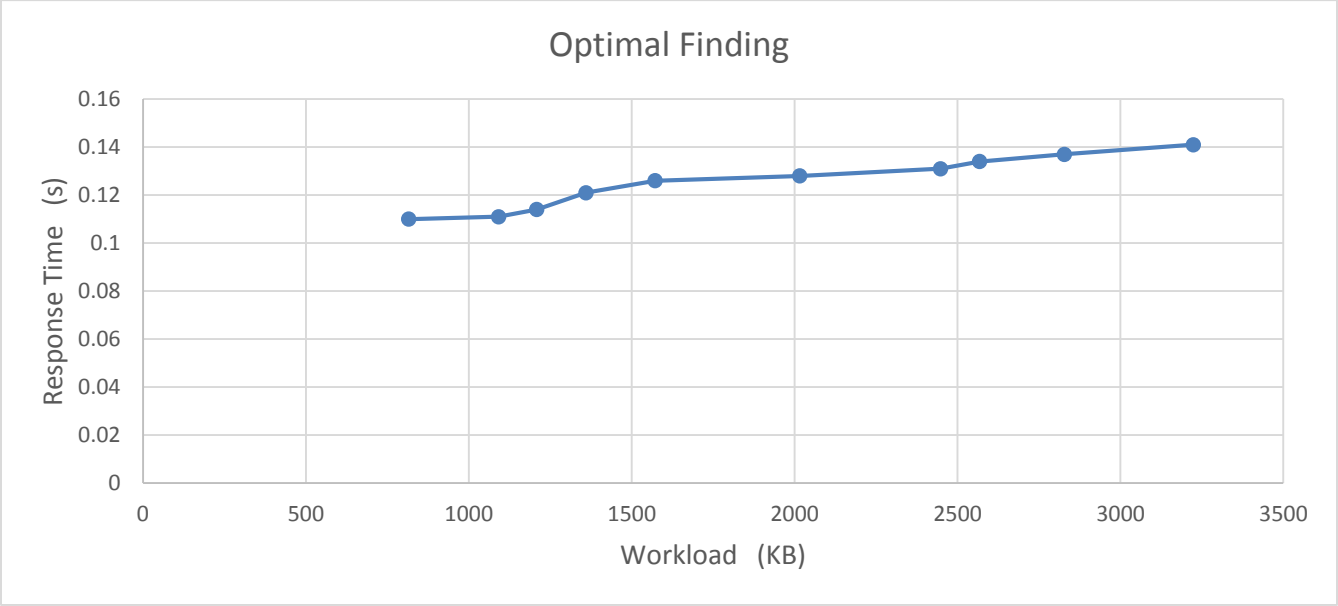

Figure 4.3 Workload redistribution using optimized alorithm

Thus, using datas from table 4.1, 4.2 and 4.3, an overall comparison is used to derive the next graph below:

Figure 4.4 Comparison of all results

Blue – Experiment 1 - Regular **MapReduce**

Dark red – Experiment 2 - Optimized **MapReduce**

Green – Experiment 3 – Workload redistribution in optimized Approach

Figure 4.4 – Comparison analysis.

This shows that the optimized approach is 2 times faster than the regular MapReduce. The Real time is about 20% worse than the optimized, but over 80% better than the regular MapReduce.

CHAPTER V

CONCLUSION

In this report, we first represent the regular Hadoop MapReduce execution with a functional model that takes into consideration processing power. Next we propose a functional model that, optimization Hadoop MapReduce further by taking communications and processing power alot into consideration. We ran experiments on a 6-node cluster. Our results show that optimized functional model outperforms the regular functional model of the Hadoop MapReduce. Results also show that the real-time approach performs better than the regular MapReduce. Future work will focus on adding optimal partitioning of the Big Data in a MapReduce job to our functional model.

REFERENCES

[1] Hadoop MapReduce http://www-01.ibm.com/software/data/infosphere/hadoop/MapReduce/ [518/2014]

[2] MapReduce Tutorial http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html [05/20/2014]

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. in *Proceedings of the 6th Symposium on Operating System Design and Implementation* (OSDI' 04). 2004. San Francisco, California, USA.

[4] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 2010. 53(1): p. 72-77.

[5] Junbo Zhang, Dong Xiang, Tianrui Li, and Yi Pan. *M2M: A Simple Matlab-to-MapReduce Translator for Cloud Computing.* Page 2. 2013 – IEEE

[6] Hadoop Basic – Creating a MapReduce program *http://java.dzone.com/articles/hadoop-basics-creating* [08/20/2014]

[7] HDFS Architecture Guide *http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html* [08/20/2014]

[8] HDFS Architecture https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html [08/20/2014]

[9] Sample Dictionary Words https://dl.dropboxusercontent.com/u/13762170/input.txt [07/1/2014]

[10] New to Cloudera http://www.cloudera.com/content/dev-center/en/home/developer-admin-resources/new-to-hadoop.html [7/20/2014]

[11] How-to: Use Eclipse with MapReduce in Cloudera&'s QuickStart VM http://blog.cloudera.com/blog/2013/08/how-to-use-eclipse-with-MapReduce-in-clouderas-quickstart-vm/ [7/30/2014]

[12] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. *The Performance of MapReduce: An In-depth Study*, p472 – 473, 2010 – PVLDB

[13] Improving MapReduce Performance in an heterogeneous environments https://www.usenix.org/legacy/event/osdi08/tech/full_papers/zaharia/zaharia_html/index.html [9/2/2014]

[14] MRONLINE – MapReduce Online Performance Tunning http://people.cs.vt.edu/~butta/docs/hpdc2014-mronline.pdf [9/20/2014]

[15] SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters http://www.sciencedirect.com/science/article/pii/S0743731513002141 [9/10/2014]

[16] Benchmarking approach for designing a MapReduce performance model http://www.cis.upenn.edu/~zhuoyao/Zhuoyao_Homepage/paper/ICPE13.pdf [9/11/2014]

[17] Improving MapReduce performance through data placement in heterogeneous Hadoop clusters http://www.eng.auburn.edu/~xqin/pubs/hcw10.pdf [9/11/2014].

VITA

ADEMOLA CHUKWUDI AINA

Candidate for the Degree of

Master of Science

Thesis: IMPROVING PERFORMANCE IN HADOOP MAPREDUCE

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2014.

Completed the requirements for the Bachelor of Science Computer Science at Novena University, Ogume, Delta State, Nigeria in 2011.

Experience:

More than 4 year experience in Software development, web development, and online security.

Professional Memberships:

National Association of Computer Science Student (NACOSS), Novena Chapter, Nigeria.