The Performance Characteristics of MapReduce Applications on Scalable Clusters

Kenneth Wottrich
Denison University
Granville, OH 43023
wottri_k1@denison.edu

Advisor: Dr. Thomas Bressoud
Denison University
Granville, OH 43023
bressoud@denison.edu

ABSTRACT

Many cluster owners and operators have begun to consider MapReduce as a viable processing paradigm for its ability to address immense quantities of data. However, the performance characteristics of MapReduce applications on large clusters are generally unknown.

The purpose of our research is to the characterize and model the performance of MapReduce applications on typical, scalable clusters based on fundamental application data and processing metrics.

We identified five fundamental characteristics which define the performance of MapReduce applications. We then created five separate benchmark tests, each designed to isolate and test a single characteristic. The results of these benchmarks are helpful in constructing a model for MapReduce applications.

1. INTRODUCTION

In the information age of today, there is a growing disparity between the amount of data being generated and the ability to process and analyze this data. However, as the size of data sets grow ever larger, it becomes more difficult both to store such quantities of data, and to provide processing ability on a large enough scale to make use of them.

Already, there are systems in place that attempt to house and process enormous quantities of data, on the scale of petabytes and exabytes, by utilizing clusters of computers in order to parallelize both processing and storage. Yet, the expansion of data may outpace the growth of clusters due to limitations in the ability of such clusters to scale. In order to address these potential issues, in 2004 Google published a paper describing a parallel processing paradigm and associated runtime called MapReduce[2]. In MapReduce, the input data is partitioned among independently-operating units of computation across a parallel system of CPUs. However, the aggregate performance of a MapReduce application does not necessarily scale linearly as we increase the size of the input data set. What is lacking is an understanding of the factors that affect MapReduce application performance.

Our objective was to characterize the performance of MapReduce applications on typical, scalable clusters based on fundamental application characteristics, with the ultimate goal of constructing a useful model of these applications. The model could then be used in quantifying performance as the application scales, and ultimately to model the performance in the presence of hardware and software failures. This paper describes a tractable model of MapReduce application performance and the initial steps of benchmarking the key factors affecting that performance.

2. BACKGROUND

After Google released MapReduce, the Apache Software Foundation announced Hadoop[1], which is set of projects featuring an open-source implementation of the MapReduce cluster-computing model. The primary Hadoop distribution is comprised of two components. The first is the Hadoop Distributed File System (or HDFS), which is a framework that allows non-identical storage hardware across multiple computers to act as a single, unified file system for storing data larger than the capacity of any individual system[3]. The second component is the MapReduce application framework, which runs applications that are broken into two distinct types of phases: a Map Phase, and a Reduce Phase. These phases are explained in greater detail below.

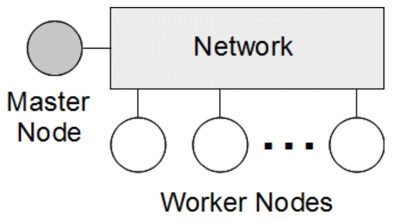

Figure 1: Hadoop network structure

Hadoop separates a small set of nodes to act as MASTER NODES, while the majority of nodes are classified as WORKER NODES (see *Figure 1*). It is the responsibility of the MASTER NODE(s) to run a NameNode, which is the controller for the HDFS, and a JobTracker, which handles the Map Tasks and Reduce Tasks involved in MapReduce applications. These tasks, which encompass the actual computation of the application, are assigned to WORKER NODES. WORKER NODES run a TaskTracker, which accepts tasks from the JobTracker on a MASTER NODE, in addition to a DataNode, which controls the storage of data on the WORKER NODE's local hardware.

Hadoop's application framework and distributed filesystem function in tandem to implement Google's MapReduce paradigm. However, Hadoop is not affiliated with Google. The diagrams and terminology used in this paper describe Hadoop's implementation of MapReduce. Google's implementation of the application framework and distributed filesystem may perform differently. We are unable to test Google's software, as it is not freely available.

2.1 Hadoop Example Application

In order to further explain the way in which Hadoop functions, we have included a detailed description of a basic MapReduce application. All MapReduce applications have the requisite property that, given a partitioning of the input data set, the correct solution can be obtained by independent tasks processing their share of the input and then combining the results. The example application, WORDCOUNT, takes in a textual corpus volume, and outputs the number of occurrences of each unique word in the volume.

The first step, before the application is even run, is to store the input data for the application on the HDFS. The NameNode running on the MASTER NODE(s) splits the data into 64-megabyte blocks, and send each block across the network to three separate DataNodes[3]. Upon receiving a block from the NameNode, a DataNode saves it to local storage on the node. By having each block saved in three locations, the NameNode helps to resist data loss due to failure or corruption at the cost of effective storage capacity. Additionally, having each block of data in multiple locations allows a block to be read from the DataNode that is the least occupied at any given time, increasing HDFS performance.

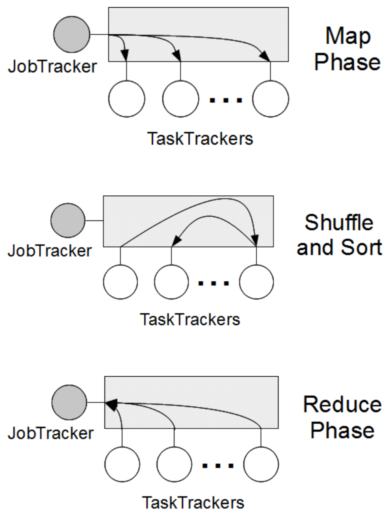

Figure 2: The stages of a MapReduce application

Once the input data is contained in the HDFS, the JobTracker on the MASTER NODE can begin the MapReduce application. The first phase of any application is the Map Phase. The JobTracker divides the input data into a set of partitions, and creates an equal number of Map Tasks. Then, the Map Tasks are instantiated by the TaskTrackers on the WORKER NODES (see the first diagram in *Figure 2*). In the WORDCOUNT application, each partition of data is a number of lines from the input data, and each Map Task parses the text and maintains a count for the number of occurrences of each unique word in its partition. Upon completing the parsing, the Map Task emits key/value pairs of the form word/sum, where word is a word from the input data and sum is the number of times that it was encountered. The Map Phase ends when all individual Map Task have completed.

Between the Map Phase and the Reduce Phase, the output of the Map Tasks is shuffled and sorted (see the second diagram in *Figure 2*). In WORDCOUNT, every unique word in the input text is a key value.

A single Reduce Task is responsible for finding the total associated with a given key value. During the shuffle, the values for a specific key are collected from every map task that emitted a pair with that key. The intermediate data is then sorted by key value for use in the next phase.

The final stage of computation is the Reduce Phase. The JobTracker creates a number of Reduce Tasks equal to the number of unique keys in the intermediate data. The tasks are again instantiated by the TaskTrackers in the cluster, which then process the intermediate data corresponding to their key. In WordCount, a Reduce Task is given a word as its key value, and it computes the sum of all occurrences by summing the values of each key/value pair corresponding to its key. Upon completion, the Reduce Tasks report back to the JobTracker (see the third diagram in *Figure 2*). The output data is then written back to the HDFS in the same way that the input data was—that is, it is broken into blocks and written to the storage housed in various Worker Nodes. The output of WordCount is a number of key/value pairs of the form word/sum, where word is a unique word from the input text and sum is the total number of instances in which it occurs in the text. Depending on the number of unique words in the original corpus, the output itself may be large.

In the following section, we describe the steps that we took to analyze the performance characteristics of MapReduce applications.

3. METHODS

There are multiple factors that affect the performance of MapReduce applications on a cluster. In order to address this, our first task was to construct a model in which we isolated five specific factors that affect MapReduce application performance. These factors are focused on quantities of input and output data. While application performance is also heavily influenced by the amount of processing time required to complete the task, we did not address this factor because we assumed that application performance would scale linearly with the availability of processing capability.

The first factor is the volume of input data to the application. The input data is partitioned among the map tasks, and the key performance metric is the bandwidth of the end-to-end pipe between the HDFS and the set of map tasks across the network. The second factor is the volume of intermediate data, which is emitted by the map tasks at the end of the Map Phase. This intermediate data must be shuffled across the cluster and sorted by key value. Here, the performance is determined by both the bandwidth and the efficiency of the sort.

Third is the volume of output data, which is emitted by the Reduce Tasks at the completion of the Reduce Phase. The NameNode on the Master Node is notified of the data to be written to the HDFS, and it instructs the DataNodes in the cluster where each block of the output should be written. The performance metric during this process is the bandwidth available to send the output data to each DataNode, in addition to the write speed of each DataNode's storage devices.

Beyond the data volumes, there are two other fundamental characteristics affecting the performance of a MapReduce application. The first is the number of Map Tasks among the available concurrently-executing cores in the cluster. The second is the number of Reduce Tasks.

Any MapReduce application may be characterized by quantifying these five factors. In order to evaluate the influence of the factors on the overall performance of a MapReduce application, we isolated these factors individually in each of our five sets of microbenchmarks.

In order to run our benchmarks, we designed and constructed two independent computing clusters to perform our testing. The first cluster, which we will refer to as Cluster 1, served as the primary testing cluster. It consists of a single computer performing the functions of a Master Node, indicated on the left in *Figure 1*, as well as 16 computers as Worker Nodes. The Worker Nodes execute both TaskTrackers and DataNodes, while the Master Node provides the NameNode and JobTracker for the cluster. Each of the 17 nodes is equipped with a quad-core Intel Core 2 processor running at 2.66 GHz. Each machine also has 4 GB of RAM and a 160 GB hard drive. The machines are connected by a 100 megabit network switch.

Cluster 2 is very similar to Cluster 1, but with some slight differences. It is also comprised of 16 Worker Nodes and a single Master Node. In contrast to Cluster 1, each of the 17 identical machines contains a quad-core Intel Core i7 processor running at 3.40 GHz, 4 GB of RAM, and a

500 GB hard drive. The computers in Cluster 2 are networked by a 1000 megabit network switch. Cluster 2 was assembled from machines residing in a classroom, and our window of opportunity to perform testing was relatively narrow. As such, only the latter three sets of benchmarks could be run on Cluster 2.

Both of our clusters are running Ubuntu Linux version 10.10. Each node is configured with Java version 1.6.0_21 and Hadoop 0.20.2. The Hadoop Distributed File System on Cluster 1 has a total capacity of 2.2 terabytes, and the HDFS on Cluster 2 has a total capacity of 6.9 terabytes.

3.1 Input Data

The first set of benchmark tests focused on the effect that the quantity of input data had on overall application performance. For this benchmark, we ran an application which accepts a text file consisting of single-digit integers, one per line. The application creates 512 Map Tasks and 1 Reduce Task. During the Map Phase, each Map Task records the number of 3's it encounters in its partition of the input, then emits a key/value pair of the form 3/*num*, such that num is the number of 3's in the partition. The Reduce Phase then combines the input from the Map Phase into a key/value pair of the form 3/*sum*, where *sum* is the number of 3's in entire input. This test was run with six separate data sets, comprised of 2, 4, 8, 16, 32, and 64 gigabytes of data, respectively.

3.2 Intermediate Data

The second test involved varying the amount of intermediate data that is emitted by the Map Tasks at the conclusion of the Map Phase of a job. For this benchmark we created an application that accepts a trivial input, and created 64 Map Tasks and 3 Reduce Tasks during its execution. The Map Phase involves each Map Task emitting a certain number of key/value pairs of the form *int*/1, such that *int* is a pseudo-randomly-generated integer between 0 and 2. The Reduce Task simply emits a key/value pair of the form *int*/*sum*, where *sum* is the total number of integers counted by particular Reduce Task. We ran applications where the Map Phase output ranged from 1 to 8 gigabytes, with a step size of 1 gigabyte.

3.3 Output Data

Next, we tested the effects of the quantity of output data emitted by the Reduce Tasks on overall job runtime. The application for this test accepts a trivial input and created 64 Map Tasks and 128 Reduce Tasks. Each Map Task emitted two key/value pairs such that the entire set of intermediate data was composed of 128 pairs of the form *int*/1, where *int* was a number 0-127. The Reduce Phase generated text files composed of single-digit integers pseudo-randomly chosen between 0 and 9, with one integer per line. We ran applications that created output files totaling between 8 and 64 gigabytes of data, with a step size of 8 gigabytes.

3.4 Number of Map Tasks

The fourth test involved varying the number of Map Tasks that were generated during the Map Phase. Here, we used a similar application to that of our Intermediate Data test, except that the total amount of data emitted by all Map Tasks was fixed at 64 GB. Then, we varied the number of Map Tasks that were generated. This benchmark test was run using applications that generated 64, 128, 256, 512, 768, 1024, 1366, and 2048 Map Tasks.

3.5 Number of Reduce Tasks

Our final test evaluated the performance of an application in relation to the number of Reduce Tasks that were generated during the Reduce Phase. We modified the application that we used in our Output Data test so that the Map Phase would generate a specific number of unique key/value pairs equal to the desired number of Reduce Tasks. The Reduce Phase would then generate a total of 64 gigabytes of data. The amount of data each Reduce Task emitted was set to change dynamically with the number of Reduce Tasks so that the total size of the output would remain the same. We tested applications with between 64 and 3072 Reduce Tasks.

4. RESULTS

The results for each of the benchmark tests that we performed are given here in terms of real-time seconds from the initiation of a job to its reported completion.

4.1 Input Data

As one might expect, the time required to run an application scaled almost exactly linearly with the amount of input data. There was very little variance in this data, and it fits the trend line almost exactly (see *Figure 3*). We saw that, on average, each gigabyte of input data increased job time by 13.2 seconds.

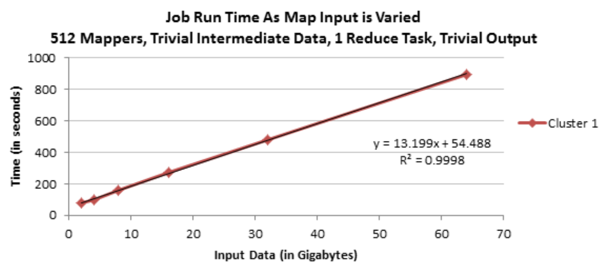

Figure 3: Input Data results

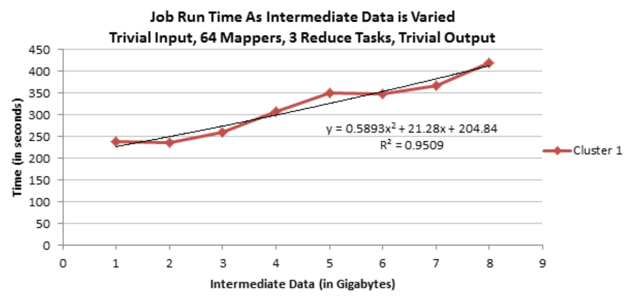

Figure 4: Intermediate Data results

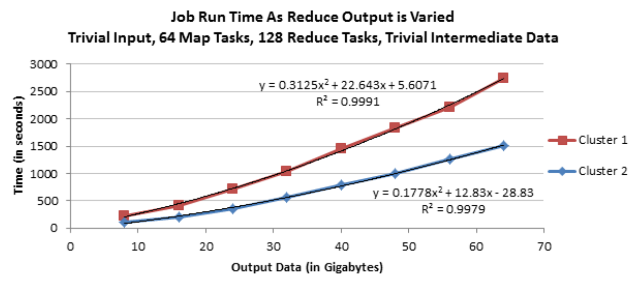

Figure 5: Output Data results

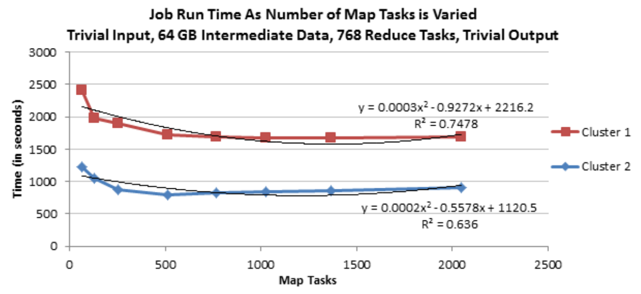

Figure 6: Number of Map Tasks results

4.2 Intermediate Data

The results for our tests revolving around the quantity of intermediate data had a high degree of variance. However, as one can observe in *Figure 4*, the resultant data still fits a quadratic curve reasonably well ($R^2 \geq 0.95$).

Here, there are multiple factors playing into the effect that the quantity of intermediate data has on total job run time. As data is emitted from Map Tasks, it is stored locally in a non-HDFS volume on the Worker Node. Then, the data is sorted by key value before the Reduce Phase begins. Hadoop implements a quicksort algorithm to sort the intermediate data, which takes $O(n \log(n))$ steps to complete. As the amount of data to sort increases, so too does network congestion. As network traffic increases toward the maximum capacity of the network hardware, performance typically drops exponentially.

4.3 Output Data

Once we began running tests on both clusters, we were able to verify that the patterns in our data were similar in both cases. When we tested the effects that variation in the quantity of output data had on total job time, we found that the data fit a quadratic regression very well ($R^2 \geq 0.99$ for both). This regression is visible in *Figure 5*.

In both these benchmarks and those examining the effects of the volume of input data, performance is limited by network bandwidth and hard drive speed. However, the read speed of a hard disk is significantly faster than the write speed. In the Input Data test, we hypothesized that network speed, which is limited at 100 megabits per second on Cluster 1, was more limiting than drive read speed. Then, in the Output Data test, since the hard drive bottleneck is its write speed rather than its read speed, it is possible that disk speed was more limiting than network bandwidth.

4.4 Number of Map Tasks

We observed an increase in job performance as the number of Map Tasks was increased, but only up to a certain point, after which performance decreased. As granularity increased, performance increased as to the number of Map Tasks increased, illustrating cluster's ability to execute Map Tasks in parallel. However, at around 1024 Map Tasks per application, increasing the number of Map Tasks no longer improved performance. *Figure 6* shows that Cluster 2 ran the application faster with 512 Map Tasks than with 768.

Our expectation for this test was that it would reveal a data distribution with a local minimum. We predicted that after a certain point, the increased

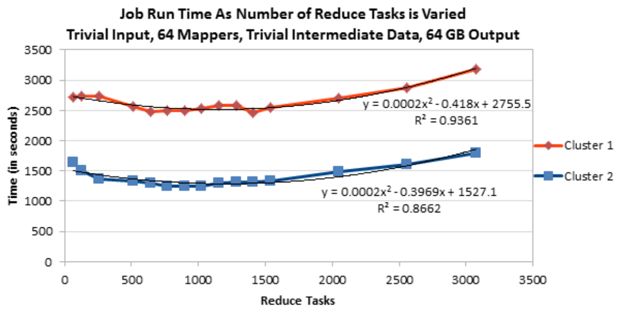

Figure 7: Number of Reduce Tasks results

overhead of creating and distributing Map Tasks would begin to outweigh the benefits of increased granularity.

4.5 Number of Reduce Tasks

In testing the effects of the granularity of the Reduce Phase on application run time, we saw an even more defined minimum than in the previous set of benchmarks. We suspected that a minimum would appear in the data for the same reasons as those in the previous testing phase.

We believe that the local minimum in the data, which occurs at around 640 Reduce Tasks in CLUSTER 1, and around 896 Tasks in CLUSTER 2, represents the optimal granularity of an application with this specific number of Map Tasks and these quantities of data (see *Figure 7*). The factors that affect where this minimum falls are the same as before: disk performance and network bandwidth. It is likely that such an optimum for a given application exists in every cluster, and its characteristics are dependent on the hardware of the cluster and the parameters of the application.

5. CONCLUSIONS

After examining our results of the five benchmark tests, the following trends were observed.

The amount of input data for a given MapReduce application had a linear effect on total application run time, where the required run time for an application increased at a rate of 13 seconds per gigabyte of data. In the intermediate data and output data tests, a polynomial trend was observed between data volume and application run time. Application run time was improved when increasing either the number of Map Tasks or the number of Reduce Tasks up until approximately 512 Tasks, beyond which no measurable benefits were observed. In fact, increasing the number of Reduce Tasks beyond this threshold had an adverse effect on application run time. This means that an optimal number of Map Tasks and Reduce Tasks exist for a given MapReduce application running on a specific cluster.

6. FUTURE DIRECTIONS

Our research is the first step in creating a model of the performance of MapReduce applications on large, scalable clusters. It is possible to take our research further in the future by executing these microbenchmarks on larger computing clusters. This would verify our results from CLUSTER 1 and CLUSTER 2 and allow us to examine the performance characteristics of these applications on different cluster sizes.

Following further testing, the performance characteristics could be used to create a model which can simulate the performance of MapReduce Applications. By combining this model with real hardware failure data, it would be possible to create a discrete event simulator that emulates actual cluster performance in the face of such failures. Such a simulator would allow us to generalize the performance characteristics of MapReduce applications on larger, enterprise-class clusters such as those that exist in data centers or cloud-computing environments.

7. ACKNOWLEDGEMENTS

This research was supported by funding from the Anderson Endowment of Denison University and the Bowen Endowment of Denison University.

8. REFERENCES

[1] Hadoop. <https://hadoop.apache.org/>, August 2011.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[3] Scott Rixner Jeffrey Shafer and Alan L. Cox. The hadoop distributed filesystem: Balancing portability and performance. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[4] Michael G Noll. Running hadoop on ubuntu linux (multi-node cluster). <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/>, August 2011.

[5] Michael G Noll. Running hadoop on ubuntu linux (single-node cluster). <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/>, July 2011.

[6] Tom White. *Hadoop: the Definitive Guide.* O'Reilly, Sebastopol, CA, 2011.