THESIS

DISTRIBUTED SYSTEMS IN SMALL SCALE RESEARCH ENVIRONMENTS: HADOOP AND THE EM ALGORITHM

Submitted by

Jason Remington

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2011

Master's Committee:

Advisor: Bruce Draper
Co-Advisor: Wim Böhm

Patrick Burns

ABSTRACT

DISTRIBUTED SYSTEMS IN SMALL SCALE RESEARCH ENVIRONMENTS: HADOOP AND THE EM ALGORITHM

Distributed systems are widely used in large scale high performance computing environments, and often conjure visions of enormous data centers full of thousands of networked machines working together. Smaller research environments may not have access to such a data center, and many jobs in these environments may still take weeks or longer to complete. Systems that work well on hundreds or thousands of machines on Terabyte and larger data sets may not scale down to small environments with a couple dozen machines and gigabyte data sets.

This research determines the viability of one such system in a small research environment in order to determine what issues arise when scaling down to small environments. Specifically, we use Hadoop to implement the Expectation Maximization algorithm, which is iterative, stateful, inherently parallel, and computationally expensive. We find that the lack of support for modeling data dependencies between records results in large amounts of network traffic, and that the lack of support for iterative Map/Reduce magnifies the overhead on jobs which require multiple iterations. These results expose key issues which need to be addressed for distributed systems to perform well in small research environments.

ACKNOWLEDGEMENTS

It is my pleasure to sincerely thank those who supported me and made this thesis possible. I will never forget the enduring support and encouragement of my thesis advisors Dr. Bruce Draper and Dr. Wim Böhm who's help and guidance enabled me to develop a much deeper understanding of the subject, and helped drive this research to completion despite all of the obstacles along the way. I would also like to thank Dr. Patrick Burns, who generously provided advice and assistance, despite only having been added to my committee at the last minute.

I am deeply grateful to my wife Elizabeth for her unselfish, and tireless support which kept my eyes open and a smile on my face, even when my confidence was wavering. I can't imagine having done this research without her.

It is an honor for me to thank my late Father, whos never ending interest in my life and work bolstered my ambition and confidence, and who taught me how to tackle problems that seem insurmountable. I will never forget how hard he worked to make sure his children could succeed and be happy in this world, and his unfailing ability to brag about us to his friends.

I would like to show my gratitude to both my Mother and Sister. It is their strength of spirit that has inspired me to continue pushing forward in times when I have felt like giving up. I am indebted to them for helping shoulder my burdens, and for their selfless desire to see me succeed.

Lastly, I would like to thank all of my friends and colleagues. Thank you so much for your moral support, and for all of the memories.

To my cherished family; future, past, and present. May nothing ever separate us.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Distributed systems are widely used in large scale high performance computing environments, and often conjure visions of enormous data centers full of thousands of networked machines working together. Smaller research environments may not have access to such a data center, and many jobs in these environments may still take weeks or longer to complete. Systems that work well on hundreds or thousands of machines on Terabyte and larger data sets may not scale down to small environments with a couple dozen machines and gigabyte data sets. These distributed systems are often benchmarked on large clusters consisting of thousands of machines using granular data consisting of a large number of small data records. Examples of these benchmarks include word count, grep, sorting numbers, and processing log files [1, 2]. Use of these systems for scientific computing on non-granular data records is a potentially useful benchmark for use in a research environment. Such data sets may consist of only a few dozen records, each of which are very large.

Use of a distributed system in small research environments, such as those found at many universities, is of great interest. Many research jobs consist of complex calculations on large scale data sets. These jobs can take days or weeks to complete on a sequential system. A distributed system has the potential to cut the runtime down on these jobs dramatically allowing for more productive research. Some distributed systems may be more applicable to this type of environment, while others may not scale down well or may not handle the task of scientific computing well. The goal is

to determine the viability of such systems in small research environments in order to determine what issues arise when scaling down to such an environment.

One such system is Apache's Hadoop MapReduce framework [1, 3, 2]. Hadoop is widely used in industry by companies such as Amazon, Adobe, AOL, Ebay, Facebook, Hulu, Twitter, Yahoo!, and even Google uses Hadoop in some of its university teaching initiatives [1, 2]. Hadoop is open-source and freely available making it a good candidate for small research environments. We use Hadoop to implement the Expectation Maximization algorithm, which uses non-granular data and is iterative, stateful, inherently parallel, and computationally expensive. This algorithm is applicable to a wide range of scientific fields including computer vision, machine learning, and artificial intelligence, and provides a realistic scientific benchmark for Hadoop.

The goal of this research is not to test a wide variety of distributed systems, but rather to use a single distributed system to learn more about the interaction and requirements of such a system in a small research environment. Similarly the goal is to use EM simply as a benchmark, meaning many of the possible non-deterministic optimizations are removed such as those which check for convergence. The EM algorithm is implemented efficiently, but is intended only to be used to provide a realistic source of work for the system, allowing for useful analysis.

Hadoop has been shown to work very well on granular data both on large and small processing clusters. The authors of [4] show a near linear speedup can be achieved for collaborative filtering based recommendation systems using Hadoop, and the authors of [5] have used Hadoop extensively in a university setting for the purpose of student projects, showing its viability as a learning tool. In both cases the projects use data which is highly granular, consisting of a large number of small data records. The research we conduct here deals with relatively few large data records.

The authors of [6] show that modifying Hadoop to gain more explicit control over the data distribution provides significant speedups when using Hadoop for scientific computing. One such optimization is allowing for the enforcement of splitting along

record boundaries. Their experiments still take place on large processing clusters. The research we conduct is restricted to small research environments, and therefor a small processing cluster.

Many authors have explored modifications to Hadoop's code to support different features. The authors of [7] look at the task scheduling algorithm used by Hadoop to improve performance in heterogeneous environments to limit the number of tasks which are restarted due to timeouts. The authors of [8, 9] both explore adding the support for iteration to Hadoop, but their systems are not nearly as widely used or supported the way Hadoop is.

Hadoop is just one of many systems available. Examples include the Granules system [10, 11] which supports Map/Reduce as well as iterative, periodic, and data driven models, but is not available at the time of this research, and Google Map/Reduce [12] which supports iteration, although the Google Map/Reduce system is proprietary and is not available for detailed performance testing.

The remainder of this thesis is divided into several Chapters. Chapter 2 provides background information on the EM algorithm, distributed systems, Hadoop Map/Reduce, and iso-efficiency analysis. Chapter 3 provides information on the data, data generation, the cluster specifications, code libraries, implementation details for the sequential and parallel code, details on the Hadoop deployment, and experimental protocol. Chapter 4 provides an analysis of the results of the experiments, and Chapter 5 provides information on future work and the conclusion.

Chapter 2

Background

This chapter provides background information for the concepts, algorithms, and frameworks used in this thesis. The material covered here is by no means complete, but provides a level of coverage sufficient for the understanding and analysis of the experiments. The remainder of this section is as follows: First is a discussion of the expectation maximization algorithm including modifications to the algorithm to allow it to run reliably in high dimensional spaces on computers which are limited by finite precision. Next we discuss distributed systems and fault tolerance, defining the key characteristics of each. We then introduce the Hadoop Map / Reduce framework for fault tolerant distributed computing; the target of much of the analysis in this paper, and finally we give an overview and example of iso-efficiency analysis.

2.1 Expectation Maximization

Expectation Maximization, commonly referred to as EM, is an iterative method for finding maximum likelihood parameter estimates for underlying distributions given incomplete data [13, 14, 15, 16, 17]. EM is broadly applicable to many statistical models, including mixtures of Gaussian densities and Hidden Markov Models [18]. One well known application for EM is its use as a supervised clustering algorithm in machine learning, although its scope extends into a variety of fields.

The remainder of this Section is broken into the following parts. First a discus-

sion of the standard expectation maximization algorithm as applied to a mixture of multivariate normal distributions is provided. Next is a discussion of a variant of this algorithm which stabilizes EM for use in high dimensional spaces. Finally a log transform is introduced which allows for a more robust implementation on computers with finite precision.

2.1.1 EM - Mixture of Gaussians

A well known use for EM is to estimate the maximum likelihood parameters of an underlying mixture of multivariate Gaussian distributions with latent variables, where the latent variables are the likelihoods that samples belong to clusters. This parameter estimate produces a fuzzy clustering of the data where each data point has soft membership in each of the multivariate Gaussian distributions in the mixture.

The algorithm is often broken down into two steps. The expectation (E) step calculates the expected value of the likelihood function given the model defined by the current parameter estimates and the observed data. The likelihood function estimates the likelihood that the current model is the process which produced the observed data. The maximization (M) step estimates the process parameters which maximize the likelihoods, given the process source estimates from the E step [13].

When using a Gaussian mixture model, each Gaussian is represented by its mean μ_i and covariance matrix Σ_i. Collectively they are referred to as the parameters $\Theta_i = \{\mu_i, \Sigma_i\}$. Let $\chi = \{x_1, x_2, x_3, \cdots, x_n\}$ be a data set of independent observations from a mixture of k multivariate Gaussian distributions of dimension D, and let $\Theta = \{\Theta_1, \Theta_2, \Theta_3, \cdots, \Theta_k\}$ be the parameter estimates for the k multivariate Gaussian distributions from which χ originates. Then the likelihood that a sample x originated from distribution i with parameters Θ_i can be estimated using the probability of that sample given those parameters [13, 18, 19]. This probability is calculated using the probability density function of a multivariate normal distribution.

$$P\left(x|\Theta_i\right) = \frac{1}{(2\pi)^{\frac{D}{2}}\left|\Sigma_i\right|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu_i)^T\Sigma_i^{-1}(x-\mu_i)}$$

The probabilities across all clusters for each sample are normalized to sum to one, and the means and covariance matrices are then updated to maximize the likelihood estimates from the previous step.

Each mean μ_i is updated using a weighted average of the samples, where the weights are the normalized probabilities for that distribution calculated in the previous step.

$$\mu_i = \frac{\sum_j^n P\left(x_j|\Theta_i\right) x_j}{\sum_j^n P\left(x_j|\Theta_i\right)}$$

Each covariance matrix Σ_i is calculated using samples weighted with the normalized probabilities for that distribution. Let $\widehat{\chi_i}$ be the samples χ weighted with the probabilities calculated for distribution i.

$$\Sigma_i = cov\left(\widehat{\chi_i}\right)$$

The E step and M step are repeated in this sequence until convergence which can be determined by measuring the changes in the probability matrix and comparing them to some threshold value. There is no guarantee that this algorithm will result in a globally optimal solution, and in some cases, the covariance matrix may be singular. One such case is when the number of samples is smaller than the number of dimensions. An example of when this might occur is in the field of computer vision, where feature vectors are calculated from images. These vectors often have the same dimensionality as the number of pixels in that image, and a typical 1000x1000 HD image produces a feature vector of length 1 million. This would require a million image data set to avoid a singular covariance matrix. It is difficult to come up with a million image dataset, and if such a set were obtained the time to produce feature vectors for each image might be unreasonable. In this case the number of samples will likely be far less than the number of dimensions, and clustering the feature vectors from these images using EM would result in singular covariance matrices. The next section discusses the solution provided in [19], which is used in this thesis.

2.1.2 EM in High Dimensional Spaces

In the case of high dimensional data, where there are more dimensions than there are samples, the covariance matrix used to represent the multivariate Gaussian distribution will be singular. To get around this problem, the authors in [19] propose representing the distribution using its eigenvalues and eigenvectors in place of the covariance matrix, and forcing the eigenvalues to maintain a small minimum value ϵ which prevents any dimensions with small or zero variance from completely collapsing. As shown in [19], this helps solve convergence problems seen with other high dimensional variants of EM which generally fit a lower dimensional distribution to higher dimensional data by throwing out the dimensions with small variance. Keeping these small dimensions can help convergence because in some cases where those small variance dimensions are the dimensions which are most discriminating between two clusters of data. Throwing these small dimensions out may prevent the algorithm from finding separation between those two clusters.

As described in [19], the eigenvalues and eigenvectors for each distribution are calculated using the singular value decomposition of the samples weighted with the normalized probabilities for that distribution. Let $\Lambda_j = \{\lambda_1, \lambda_2, \lambda_3, \cdots, \lambda_D\}$ be the eigenvalues for distribution j, and let Q_j be the eigenvector matrix for distribution j, collectively they are referred to as the parameters $\Theta_j = \{Q_j, \Lambda_j\}$. Then, given a sample x_i, the projection of that sample into the subspace defined by Q_j is $y_{ij} = Q_j x_i$. Then, as before, the likelihood that a sample x originated from distribution j with parameters Θ_j can be estimated using the probability of that sample given those parameters [19].

$$P\left(x_i|\Theta_j\right) \approx \frac{1}{(2\pi)^{\frac{D}{2}} \prod_{d=1}^{D} max\left(\epsilon, \lambda_d^{\frac{1}{2}}\right)} e^{-\frac{1}{2}\sum_{d=1}^{D} y_{ijd}^2 / max(\epsilon, \lambda_i)}$$

As before, the probabilities across all clusters for each sample are normalized to sum to one, and then the eigenvalues and eigenvectors are updated to maximize the likelihood estimates from the previous step using the singular value decomposition. The E step

and M steps are repeated until convergence. This variant, like the standard version in the previous section, provides no guarantee of global optimality.

2.1.3 EM Log Transform

If EM is implemented as shown in the previous sections in a computer, it is very possible that underflow issues may occur in some of the calculations. Underflow can occur when calculating the exponential in the numerator, or when calculating the product of eigenvalues in the denominator. Even though the numbers in these calculations can be extremely small, they are still significant for the purpose of the algorithm since the probabilities are later normalized. To solve this issue, the probabilities are log-scaled.

Log scaling data is a method where data are represented as the log of a quantity instead of the quantity itself which brings data that covers a large range of values into a smaller more manageable range. This method is widely used in mathematics, and well known examples include the Richter magnitude scale for measuring the seismic energy of earthquakes, the Decibel scale for measuring acoustic power, and Entropy in thermodynamics. Using this method, the above equation can be stabilized to avoid underflows. The equation below is produced simply by taking the log of the equation in the previous section, and produces the log of the probability instead of the probability itself. The exponential in the numerator, and the product in the denominator become summations under this transform, and are much less likely to underflow.

$$log\left(P\left(x_i|\Theta_j\right)\right) \approx -\frac{1}{2}\sum_{d=1}^{D}\frac{y_{ijd}^2}{max\left(\epsilon,\lambda_i\right)} - \frac{1}{2}\sum_{d=1}^{D} log\left(max\left(\epsilon,\lambda_d\right)\right) - \frac{D}{2}log\left(2\Pi\right)$$

Since the probabilities are normalized, the constant can be dropped from the end giving the form of the equation used in this thesis.

$$log\left(P\left(x_i|\Theta_j\right)\right) \approx -\frac{1}{2}\sum_{d=1}^{D}\frac{y_{ijd}^2}{max\left(\epsilon,\lambda_i\right)} - \frac{1}{2}\sum_{d=1}^{D} log\left(max\left(\epsilon,\lambda_d\right)\right)$$

Once these values are calculated for a given sample across all clusters, the max of those values for that sample is subtracted from those same values. This effectively scales the maximum value to zero, which corresponds to a probability of one. This is done to prevent an underflow from occurring when taking the exponential of all values to remove the log transform. Underflows can still occur, but only when a cluster has a normalized probability that is so close to zero that a roundoff occurs. In this case, the issue is negligible.

2.2 Distributed Systems - Reliability and Fault Tolerance

The Google Code University (GCU), which provides supplemental curriculum and coursework for computer science students and educators, defines a distributed system as "an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks" [20, 21]. In large distributed systems, with thousands of nodes, systems fail, are removed, and are replaced on a daily basis, but the system must continue to run uninterrupted 24/7. System failure is business as usual in this environment, and as a result fault–tolerance is a system requirement [20, 21, 1]. If a system is fault–tolerant, and disks are failing daily, it doesn't matter if the data center uses high-end server grade disks, or standard commodity hardware, therefore a fault–tolerant system allows for cheaper data centers [21]. The goal, then, is for a distributed system to run reliably on unreliable commodity hardware in the presence of node failures and network failures in a scalable fashion. The remainder of this section defines what reliability and fault–tolerance mean in this environment.

2.2.1 Reliability

A distributed system provides significant parallel resources to remote users allowing them to execute large tasks which would otherwise take too long to execute on a

sequential system. Such a system must be reliable, and reliability is the central issue in the design of a distributed system [22, 21]. A reliable distributed system is fault–tolerant, highly available, recoverable, consistent, scalable, predictable, and secure [22].

A fault–tolerant system will continue to operate correctly in the presence of regular component failures. When a component fails, a fault–tolerant system will recover without having performed any incorrect actions [22, 21]. This is one of the most difficult aspects of reliability to achieve. Fault–tolerance is often achieved through replication and redundancy of data, tasks, and bookkeeping along with avoiding state through the use of indempotency, or keeping state recoverable through the use of transactions, and the use of data verification techniques such as checksums.

A highly available system can be restored after component failures allowing it to continue working with minimal downtime [22, 21]. If fault–tolerance is achieved, the data will not be corrupted, and with sufficient tracking and logging, incomplete tasks can be restarted or resumed, and the system can continue providing service.

A recoverable system can reintegrate failed components on the fly once they have been repaired [22, 21]. A recoverable system will rebalance its workload and begin utilizing recovered components as they are added back in [22, 21]. This can involve re-registering one or more nodes with the system, redistributing data, and rescheduling tasks to utilize the newly available components.

A consistent system will complete the specified jobs through the efficient coordination of available components as one system working concurrently and in the presence of failures [22, 21]. A consistent distributed system appears to be one large non-distributed system, which is accomplished through the creation of complex protocols which allow all the components to communicate and coordinate execution with one another [22, 21].

A scalable system will not degrade in performance as various aspects of the system are made larger in size [22, 21]. Adding more nodes, more network connections, more

data, more users, etc. should not have a major negative impact on the performance of the system [22, 21]. Scalability is achieved through minimal tracking of state, and through efficient and clever use of buffers, storage, and network bandwidth.

A predictable system provides acceptable levels of responsiveness to its users [22, 21]. Jobs should execute efficiently, and return in an acceptable amount of time [22, 21]. This is achieved through concurrency, efficient management of resources, and efficient use of resources. A predictable system will make good use of the connected components as required by the job.

A secure system authenticates users of the system, preventing unauthorized users from gaining access to the data and services provided by the system [22, 21]. This is accomplished through the usual means using per user secure login information, biometrics, or other security protocol for gaining access to the system.

All of these aspects are present in some form or another in any modern day distributed system, although they don't come without cost. Increasing one aspect of reliability may have negative impacts on another aspect of reliability. The goal is to obtain acceptable levels of each.

2.3 Hadoop Map/Reduce

The Apache Hadoop MapReduce project is a distributed computing framework that enables developers to write applications which run reliably on a large number of unreliable machines with the goal of processing Terabyte and larger data sets in parallel on clusters consisting of thousands of nodes [1]. Hadoop is an open source implementation of the MapReduce framework inspired by Google MapReduce, and the Google File System (GFS) [12, 23], although the two systems are very different. These differences include a difference in implementation language, differences in support for iteration, and many other differences that are unknown since Google MapReduce and the Google File System are both proprietary private systems that are not available for public analysis.

The remainder of this section will be organized as follows. First an overview of the Hadoop Distributed Filesystem (HDFS) will be given. After that, the Hadoop Map/Reduce framework will be discussed.

2.3.1 HDFS

The Hadoop Distributed File System (HDFS), inspired by GFS from Google [23], is a distributed filesystem which runs on low cost commodity hardware in a fault tolerant manner to redundantly store Terabyte and larger data sets [3]. HDFS differs from GFS in many fundamental ways, but has many of the same goals, the most important of which is to provide reliable distributed access to large amounts of data. Typically, this filesystem is running on the same nodes as the Map/Reduce framework to allow tasks to execute locally to the data [1]. The data is stored in 64MB or larger blocks in a set of DataNodes which serve data to the applications, and there is typically one DataNode per node in the cluster [3]. A master NameNode maintains all of the metadata associated with the data stored on the DataNodes. The NameNode is responsible for coordinating access to the data, and coordinating replication of the data [3]. The NameNode stores all of this information in RAM for fast access, but keeps an image on disk in case of failure. If the NameNode fails, this image may be used to restart the NameNode [3]. The DataNodes themselves store no information about the data they contain [3]. The DataNodes simply store each block of data as a separate file on the local file system [3]. When the NameNode splits and distributes data it does not respect record boundaries in the data. It is often the case that the first and last record in a split are truncated [3]. In the case of truncated records DataNodes will transfer the remaining portion of the truncated records to the appropriate machines during runtime [3].

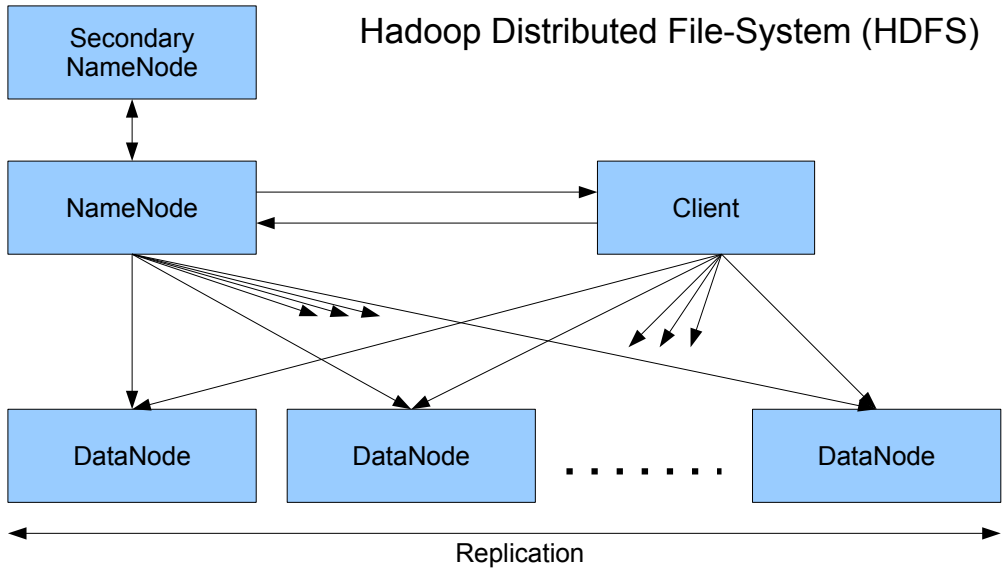

Figure 2.1: The HDFS architecture which consists of a primary NameNode that coordinates access to the data, DataNodes which store the 64MB data splits, and a secondary NameNode which duplicates the state of the primary NameNode in case of failure. The DataNodes typically run on the same machines as the map and reduce tasks for data locality.

The HDFS architecture is designed to be a write-once, read-many system [3]. Once a file is closed, it is replicated across the cluster, and cannot be written to again. The system is optimized for high sustained throughput streaming reads. This means it is faster to read an entire 64MB block, filtering out information that isn't needed, than it is to do a low latency seek to specific locations in that block to read the same information [3].

The Apache team identifies three types of failures. NameNode failures, DataNode failures, and network partitions [3]. The NameNode is a single point of failure, and if the NameNode fails, it must be manually restarted using the most recent image. This image may be replicated on multiple machines in case of disk failure on the NameNode. In the case of a network partition or a DataNode failure, data on the affected DataNodes are unavailable and are re-replicated on the surviving DataNodes [3]. A heartbeat message is repeated across the network during execution to

track the availability of the DataNodes for this reason [3].

Data in the HDFS can be accessed using the command line, a Java API, or a C language wrapper for that same API. From the users perspective, the data access is similar to that of a normal file system. The details of data storage in the HDFS discussed in this section are hidden from the user. The Java API for accessing HDFS data is very similar to the API's used for normal file access, allowing for all of the familiar operators including implementations of input and output streams which allow for buffered reads and writes using the standard Java API [3].

2.3.2 Map Reduce

The data sets for Hadoop MapReduce are stored in the Hadoop Distributed File System (HDFS) in data blocks which can be processed independently by map tasks in parallel. The entire Map/Reduce framework operates on $\langle key, value \rangle$ pairs, and individual map tasks never communicate with one another [1].

Each map task processes input records in isolation [1]. It is often the case that a record is simply a single newline delimited piece of data, although this aspect is highly customizable [1]. The map task then outputs zero to many intermediate output records [1]. Optionally, the output records from the map tasks may be sent through a combiner which consolidates the records local to a map task into a smaller set of equivalent records before being sent to the reducer [1]. This combiner can be implemented using the same function used for the reduce task [1]. The records are sorted by key, and all values with the same key are processed together by the same reducer in no particular order [1]. The reducer phase cannot start until the map phase has completed [1], and each reducer produces zero to many output records for the Map/Reduce job [1]. The data types used as the inputs to any given phase do not need to be the same as the outputs [1]. Furthermore, the application is not required to use the input keys or values for anything. It is also the case that a Map/Reduce job may not need both a Map and Reduce phase. Many jobs use an identity map

task, and simply process data in the reduce tasks, or vice versa.

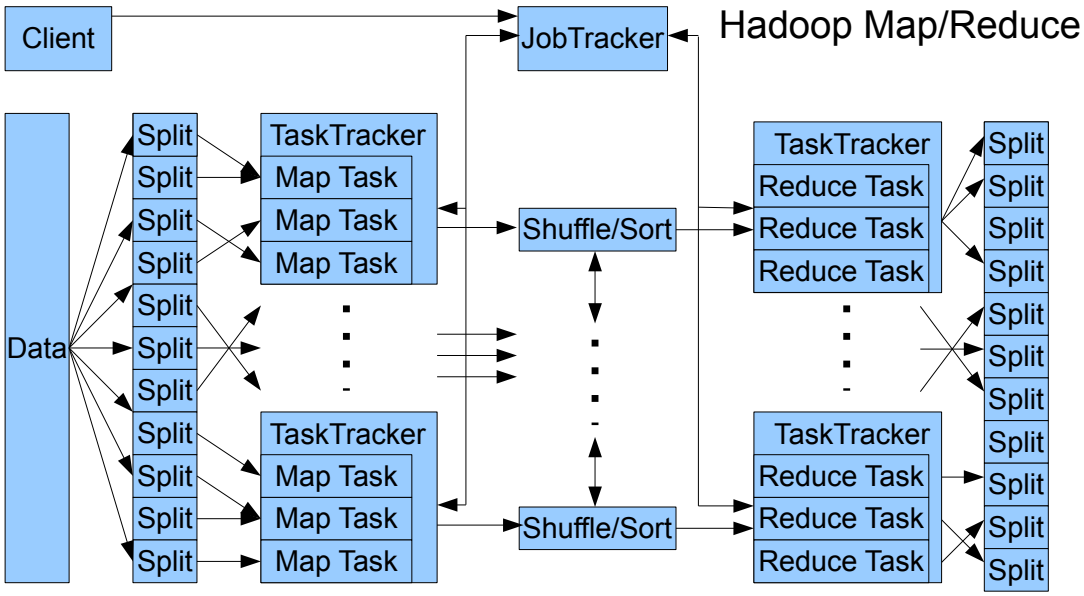

Figure 2.2: The MapReduce framework uses the split data blocks stored in the HDFS as input to map tasks which run local to the data. A JobTracker coordinates the distribution of map and reduce tasks to the TaskTrackers running on each of the machines. All data are passed in the form of key value pairs. The outputs of the map tasks are shuffled and sorted based on the key's and all values with the same key are guaranteed to be passed to the same reducer. The output of the reduce task is written back to the HDFS.

The minimum job length is estimated to be around thirty seconds to a minute, although the typical use case for Hadoop results in jobs which take hours, or days to complete on thousands of machines [2]. This thirty second overhead multiplies when the algorithm requires multiple Map / Reduce iterations, and is an important factor to consider when choosing whether to use the Hadoop framework for a given job.

2.4 Iso-Efficiency

On a sequential machine, algorithms are often evaluated by their runtime in terms of their asymptotic complexity (i.e. Big-O). On parallel machines, other factors such

as overhead, number of processors, and how well the algorithm can be parallelized become issues. The best known algorithms for a given problem on a sequential machine may be terrible choices in a parallel environment. Even among good parallel algorithms, the number of machines, and the size of the problem become issues as some algorithms scale better than others, or are better suited to certain environments [24]. The question to be answered is how scalable is a given parallel algorithm in terms of problem size, topology, and number of processors [25, 26]. To resolve this issue, a metric known as iso-efficiency is used to analyze the efficiency and scalability of a system. Specifically, iso-efficiency analyzes the size of a problem in terms of the topology and number of processors in order to keep efficiency constant [25, 26]. This is necessary since increasing the number of machines increases overhead, while increasing the problem size increases load on each machine. The goal is to match the right number of machines to the problem size in order to utilize the machines resources efficiently [25, 26].

2.4.1 Iso-efficiency Framework

The iso-efficiency framework defines cost, speedup, overhead, and efficiency in terms of the number of processors, topology, and the complexity of the algorithm [25, 26]. The first step is to concretely define these terms. Let T_p be the time an algorithm takes to solve a problem on P processors, and let T_1 be the time taken to solve the problem on one processor. Let T_1 be equal to the total work W done to solve a given problem.

$$T_1 = W$$

Let cost C_p be the number of processors multiplied by the time spent on each processor.

$$C_p = PT_p$$

A system is cost optimal $C_{optimal}$ if cost is asymptotically equal to W.

$$C_{optimal} = O(W) = O(C)$$

Let overhead $T_{overhead}$ be defined as the cost minus the work, and assume no overhead in the case of just one processor.

$$T_{overhead} = PT_p - W$$

Let speedup S_p be the ratio of the time spent on a single machine over the time spent per processor on P processors.

$$S_p = \frac{T_1}{T_p}$$

Let efficiency E_p be the ratio of speedup to the number of processors used.

$$E_p = \frac{S_p}{P}$$

Speedup is typically limited by the number of processors. Speedup on a single machine is equal to one, but does not necessarily increase linearly with the number of processors [25, 26]. Speedup tends to saturate due to increasing overhead, and as a result efficiency decreases as the number of processors increases [25, 26]. This is known as Amdahl's law, and is true for all parallel systems. Conversely speedup on a scalable system tends to increase as the problem size increases since overhead on a set number of processors does not increase as fast as the workload, and as a result efficiency goes up [25, 26]. For a system to be scalable, the ratio of work to overhead must remain constant [25, 26]. Let K be the constant ratio of $T_{overhead}$ to W.

$$\frac{T_{overhead}}{W} = K$$

$$W = KT_{overhead}$$

Iso-efficiency analysis determines the rates at which to increase the problem size and the number of processors used in order to keep the efficiency fixed for a given system [25, 26].

The growth of the problem size compared to the growth of the number of processors determines the scalability of the system. If the problem size must grow exponentially with respect to the number of processors, then good speedups are impossible for a large number of processors unless the problem size is massive [25, 26]. Such a system is poorly scalable; whereas if the problem size grows linearly with the number of processors the system is highly scalable [25, 26]. The iso-efficiency function determines the level of speedup obtainable from a system proportional to the number of processors used, and therefore when comparing two algorithms, the algorithm with the lower iso-efficiency function is the better choice [25, 26].

2.4.2 Adding Numbers on a Hypercube

The authors of [25, 26] use a simple parallel algorithm for adding numbers in parallel to demonstrate the use of iso-efficiency in a real scenario. This algorithm uses the topology of the hypercube as the basis for distributing the workload. Assume we have n numbers to be added on p processors, and $n \geq p$. We model the problem so that processors reside on the corners of a hypercube, and each processor is given $\frac{n}{p}$ numbers. The processor adds all of the numbers in its queue in $\frac{n}{p}$ steps, and then half of the processors send their partial sum to a neighboring processor, thus reducing the dimensionality of the hypercube. Each processor adds the two partial sums, and the values get sent again, reducing the dimensionality of the hypercube again. We assume the send step and the add step both take one time step. The send step and the add step are repeated $log(p)$ times. Using the equations in the previous section, we have the following relations.

$$T_1 = W = n$$

$$T_p = \frac{n}{p} + 2log\,(p)$$

$$C = n + 2plog\,(p)$$

$$T_{overhead} = n + 2plog\,(p) - n = 2plog\,(p)$$

$$S_p = \frac{np}{n + 2plog\,(p)}$$

$$E_p = \frac{n}{n + 2plog\,(p)}$$

If $n = p$, then the cost is $p + 2plog\,(p) = O\,(plog\,(p))$. This is not asymptotically equal to $T_1 = p = O\,(p)$ and therefore not cost optimal. If $n = plog\,(p)$, then the cost is $3plog\,(p) = O\,(plog\,(p))$ which is asymptotically equal to $T_1 = plog\,(p) = O\,(plog\,(p))$, and therefore cost optimal. Furthermore, the ratio of work to overhead is constant if $n = plog\,(p)$. If we want to keep 80% efficiency then the iso-efficiency function can be calculated by setting $E_p = 0.80$, and solving for n.

$$0.80 = \frac{n}{n + 2plog\,(p)}$$

$$n = 8plog\,(p)$$

This function now describes the rate at which to scale the number of processors with respect to the problem size in order to keep constant efficiency, and can be compared to the iso-efficiency function of another algorithm to determine which will scale better.

Chapter 3

Experimental Design

This chapter describes the methods used to evaluate the use of Hadoop on an iterative algorithm, on sub-Terabyte datasets, on a cluster consisting of a single rack of machines. The remainder of this Section is divided into the following parts. First, section 3.1 describes the data, as well as the method used to generate them. Next, section 3.2 describes the specifications of the computers used in the cluster. Section 3.3 details the libraries used in the implementation of the EM algorithm. Section 3.4 describes implementation details of both the sequential and parallel versions of the code. Section 3.5 describes experiment specific details of the Hadoop deployment on the cluster. Section 3.6 describes the metric used to evaluate the systems. Lastly, Section 3.7 details the experimental protocol used in this thesis.

3.1 The Data

The data used to benchmark EM on Hadoop are randomly generated from a mixture of Multivariate Normal Distributions. The points for each cluster are generated using the Box-Muller method [27], are rotated using a random rotation matrix, offset from the origin by a random distance, and once again rotated around the origin. This data generation has been tuned to give good separation between clusters, although overlapping clusters are still possible. This method also gives access to data sets of arbitrary size, dimension, and underlying cluster count. The implementation chooses

the number of points in each cluster from a normal distribution, so each cluster has a similar number of points, although not exactly the same number of points. As a final step, points are added or removed from random clusters to correct the sample count. Sample data sets generated using this method are displayed in Appendix A.

Since the data are generated from Gaussian processes, it is well suited for the application of EM as a clustering algorithm since EM models the data as a mixture of Gaussians. The data sets range in sample count N from $50,000$ to $350,000$, and in dimension D from 2 to 90; the source process ranges from 5 to 25 source data clusters K. The dimensionality and sample count determine the raw size of the data, and the source cluster count determines the parallelizability of the algorithm on that data set. The values are chosen to evenly cover the testable range of data sizes on the cluster. Larger sizes result in file transfer failures that may be the result of system quotas being exceeded, bandwidth throttling, or other administrative restrictions. Unfortunately these issues are beyond user control on the cluster, and the specific issue preventing the larger file transfers is still unknown.

The generated data sets vary one parameter at a time. As one parameter is varied the other two are held constant at the following values: the sample count N is held constant at $250,000$; the dimension D is held constant at 50; and the source cluster count K is held constant at 15. The source cluster count determines the inherent limit on the parallelization of the data. The number of processors is bounded by the underlying source cluster count.

The experiment is repeated on a smaller data set which has the same data ranges as the larger one, but the values at which the non-varied parameters are held constant are smaller. This is done to verify that the runtime trends are consistent at different workloads. These values are the following: the sample count N is held constant at $50,000$; the dimension D is held constant at 10; and the source cluster count K is held constant at 10. The results for these smaller data sets are included in Appendix A.7.

3.2 The Processing Cluster

The processing cluster used in this thesis consists of 30 rack-mounted x64 based SunFire X4100 servers each with dual 2.8Ghz AMD Opteron 254 processors and 8Gb of RAM. The machines all run 64 bit Linux, and are connected to a NFS share, and have drives available for local storage. All experiments are conducted on the processing cluster in the absence of other users. Some of the experiments do not utilize all of the machines, see Section 3.1 for details on the data dependent limitations on parallelization of the EM algorithm.

3.3 Math Libraries

The matrix operations required by the EM algorithm are all implemented using ATLAS optimized CLAPACK and CBLAS routines [28, 29, 30, 31, 32, 33, 34]. These libraries provide C language wrappers for Fortran-based linear algebra routines. Specifically, the functions for the singular value decomposition of a matrix as well as matrix-matrix and matrix-vector multiplies are used.

3.4 The Implementation

We test a computationally expensive algorithm that is widely used in the fields of machine learning and computer vision known as Expectation Maximization which is described in detail in section 2.1. The algorithm is iterative in nature, and in a Map/Reduce context requires both a map and reduce phase for each iteration due to the normalization step. This algorithm makes a good candidate because it is parallel in nature, its computationally expensive, and provides a realistic use case for Hadoop.

3.4.1 Sequential Implementation of EM

In order to provide the best baseline possible, a sequential version of EM is heavily optimized. The algorithm is implemented in C++ and all matrix operations are

implemented using the libraries described in section 3.3. The matrices which store the eigenvalues, eigenvectors, probabilities, and data samples are allocated once, and never copied or moved. All matrix operations involving these matrices are performed in place.

The initialization of the cluster centers uses the subset furthest first algorithm [35]. This algorithm is used because it is more resistant to outliers than the standard furthest first algorithm, while still producing an initialization that is well spread across the data. Careful attention is paid in the implementation of this algorithm, since a naive implementation results in an unacceptable time-bound and runtime, and a much more careful implementation is required to obtain the time-bounds advertised by the authors in [35].

The centers are used to initialize the probabilities using the normalized inverse euclidean distance from each sample to each center as per the recommendations in [19]. The first cycle through EM uses these initial probabilities to calculate new centers before the first set of true probabilities are calculated.

The main EM loop consists of three steps, the first of which updates the matrix containing the cluster centers. The second step uses a singular value decomposition of the samples weighted by their probabilities to update the eigenvalue and eigenvector matrices. The final step uses the updated centers, eigenvalues, and eigenvectors to update the probability matrix. All of the loops and functions have been meticulously optimized to maximize code locality, minimize cache misses, and written in such a way as to give the compiler as much of an opportunity to internally optimize the code as possible.

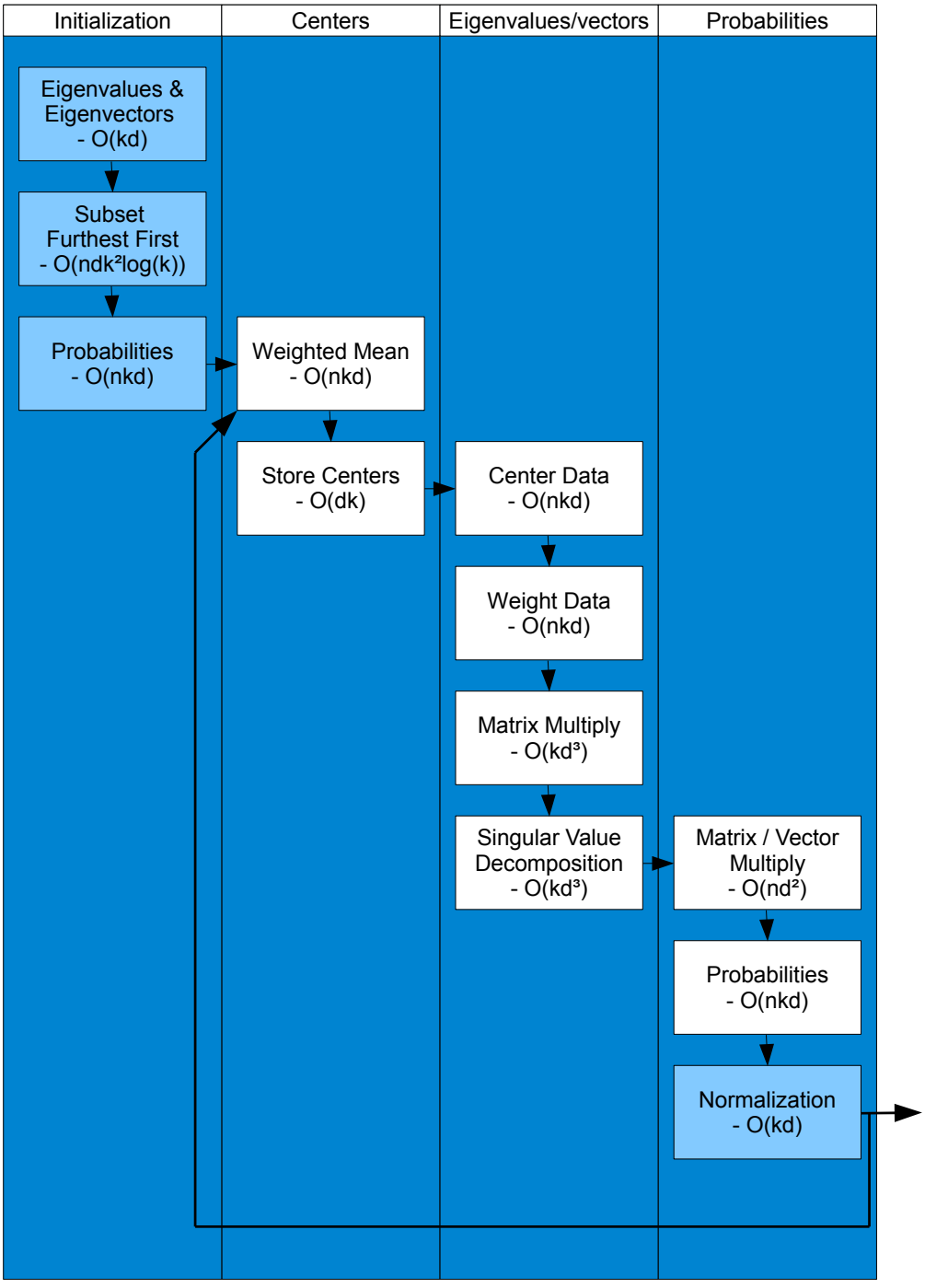

Figure 3.1: Block Diagram of the sequential implementation of the EM algorithm. The white boxes represent steps which are distributed in the parallel version while the shaded boxes represent steps which remain sequential. The initialization steps are not included in the timings, and the normalization step is taken care of in the reducer in the distributed version.

3.4.2 Parallel Implementation of EM

The Hadoop Pipes interface provides a C++ access point to the Map/Reduce framework via a wrapper which uses sockets to communicate with the Java code. This allows the direct reuse of code from the sequential implementation of EM in the parallel version. The initialization stage is not done in parallel and is handled outside of the Map/Reduce framework. Nearly all of the code from the sequential version can be parallelized, so each mapper contains the portion of the code from the sequential version needed to update a single data cluster. Since each processor updates a single data cluster, the inherent parallelization of the algorithm is limited by the number of underlying data clusters. The only code that is not in the mapper is the normalization step, which requires the output of all the mappers to complete. For this reason, the normalization is taken care of in the reducer.

Since EM is iterative, and since each iteration is a complete Map/Reduce job, the job submission and iteration is handled by a Java based driver which submits job configurations programmatically via the Submitter object. Each submission starts a full Map/Reduce iteration, the output of which becomes the input to the next iteration. Timings are only taken during the iterations, so the negligible amount of time between iterations is not included in the timings. Unfortunately, the Map/Reduce framework itself has no concept of iteration. Once mapper outputs are reduced, they cannot be fed into another Map/Reduce cycle in the same job.

One of the great strengths of Hadoop is also one of its great weaknesses. The fact that data are distributed and replicated automatically is a very nice feature, but in EM there are major internal data dependencies, and while the structure of the data can be modeled in Hadoop, these dependencies cannot. As a result there is an inevitable slow down when data are unavailable to mappers that need it. Hadoop handles this situation, but it slows things down, and the resulting network overhead is one of the limiting factors on scaling the workload on the infrastructure used for this paper.

3.5 Hadoop

Version 0.20.0 of Hadoop Map/Reduce [1] is used in this research. In order to provide the most direct comparison between the serial C++ implementation of EM, and the Hadoop implementation, the Hadoop Pipes interface is used, which provides a C++ interface to the Map/Reduce framework. This allows direct reuse of nearly all of the code used in the serial C++ version.

The configuration of the Hadoop software for use on the SunFire cluster is based on recommendations found in [1, 2]. The DataNodes use the local temp directories on each machine for storing data, while the Hadoop installation is located on NFS. The read-only data matrix needed by all of the mappers is stored in the distributed cache, and the matrix of probabilities from each iteration is stored in the HDFS.

The number of map tasks is directly dependent on the number of clusters in the data. As a result, this value is set programmatically and is specified as an input to the job. Each map task executes the EM algorithm for a single cluster in the data, excluding the normalization step. The output of each task are the updated probabilities for the cluster assigned to that map task. There is a single reduce task, whose sole job is to collect all of the probabilities for each of the clusters, and then execute the normalization step of the EM algorithm. The output of the reduce task becomes the input to the next Map/Reduce job submitted to Hadoop.

Clusters are pre-balanced using a priming job, since the first iteration after a fresh cluster startup was observed to take much longer than successive iterations due to initial load balancing. This priming job, which consisted of a dummy map reduce job, prevented that effect from affecting the data, making all iterations comparable. The motivation is that in a real environment, the cluster would not be restarted frequently.

Normally, iterations will continue until a convergence criteria is met, but in the case of this research, the same number of iterations is used across all runs to provide an equal comparison. The number of iterations chosen exceeds the number of iterations needed for convergence on the data sets used.

3.6 Metric

The metric used to evaluate the performance of EM on Hadoop is the empirical runtime measured on the cluster in the absence of any other users or jobs. The serial version is executed on one of the machines in the cluster in order to provide the most directly comparable results possible. The runtimes are collected using millisecond accurate timings, and runtimes are recorded for both the overall runtime of the Hadoop job, as well as times spent inside of each mapper and reducer. This allows for a calculation of overhead contributed by the Hadoop framework.

3.7 Protocol

Before each experiment, the Hadoop system is stopped, HDFS is formatted, and Hadoop is configured for the experiment. The cluster is started, and the job is submitted. This gives each experiment an equal environment to eliminate any possible interaction effects from previous jobs. As described in Section 3.5, a priming job is used before each experiment to get rid of the first run spike due to initial load balancing after cluster startup.

Detailed timings are recorded for both work and overhead. For the parallel version, the work timings are recorded separately within mappers and reducers, allowing for a further breakdown. Timings are recorded in both the sequential and parallel versions as data parameters are varied. Each of the sample count N, the underlying cluster count K, and the dimensionality of the data D are varied one at a time to show the growth of work and overhead with respect to those parameters. This process results in a large number of timing values from each map and reduce task from each iteration from each job with varied values of N,K, or D. Within a given job, the timings for all of the mappers tend to have very small standard deviations, and therefore can be summarized accurately by their means. Since all map tasks execute in parallel, this time represents the portion of the total time spent in the mapping phase. Both the

means and standard deviations for the map phase are reported in the appendix in section A.1. Reporting just the mean allows for a much simpler visualization of the data. Since there is only one reducer, the reported timings for that phase are not summarized in statistics.

Each of the data dimensions N,K, and D are varied from a small value up to the maximum achievable range for that parameter on the experiment infrastructure. Larger data sets result in errors which Hadoop logs indicate to be caused by failed or timed out file transfers on the underlying system.

The overhead is calculated by taking the total time taken by a job, and subtracting the mean time spent in the map phase for each iteration, as well as the time spent in the reduce phase for each iteration. The time left over consists of time spent by Hadoop in initializing map and reduce tasks, IO overhead, load balancing etc, while no map or reduce is executing. In an iso-efficiency sense, we are taking the total cost minus the work to get this value of the overhead.

The serial version has no parallel overhead, and has no sense of a map or reduce phase. The serial timings consist of the difference in time between the start and finish of each iteration in a given job. These timings act as the baseline for comparison of the parallel timings. These timings are collected on all of the same data sets used in the parallel version.

The goal is to determine if Hadoop is viable on a widely used, easily parallelizable, computationally expensive, iterative algorithm in a small research environment, and to identify what factors, if any, limit the viability of Hadoop in this environment. If Hadoop fails to serve as the framework for this algorithm, the results will give insight into both the weak and strong points of the framework.

Chapter 4

Results

This chapter explains the results of the experiments described in section 3.7 for comparing the Hadoop implementation of EM to the sequential baseline implementation. This chapter is divided into two sections. Section 4.1 discusses the sequential baseline results, while section 4.2 compares the results from the Hadoop implementation to that baseline.

4.1 Sequential Baseline

The sequential baseline timings show the time spent in a single iteration as each data parameter is varied. These experiments use the same datasets as the Hadoop experiments. All three figures display roughly linear growth as the parameters are varied. This implies that the runtime scales linearly with the work as a whole. The lack of overhead gives the sequential version an advantage on smaller datasets, but the assumption is that as work is scaled up, a crossover point will be found. Cost optimality is achieved when a P fold speed-up over this sequential baseline is achieved where P is the number of processors.

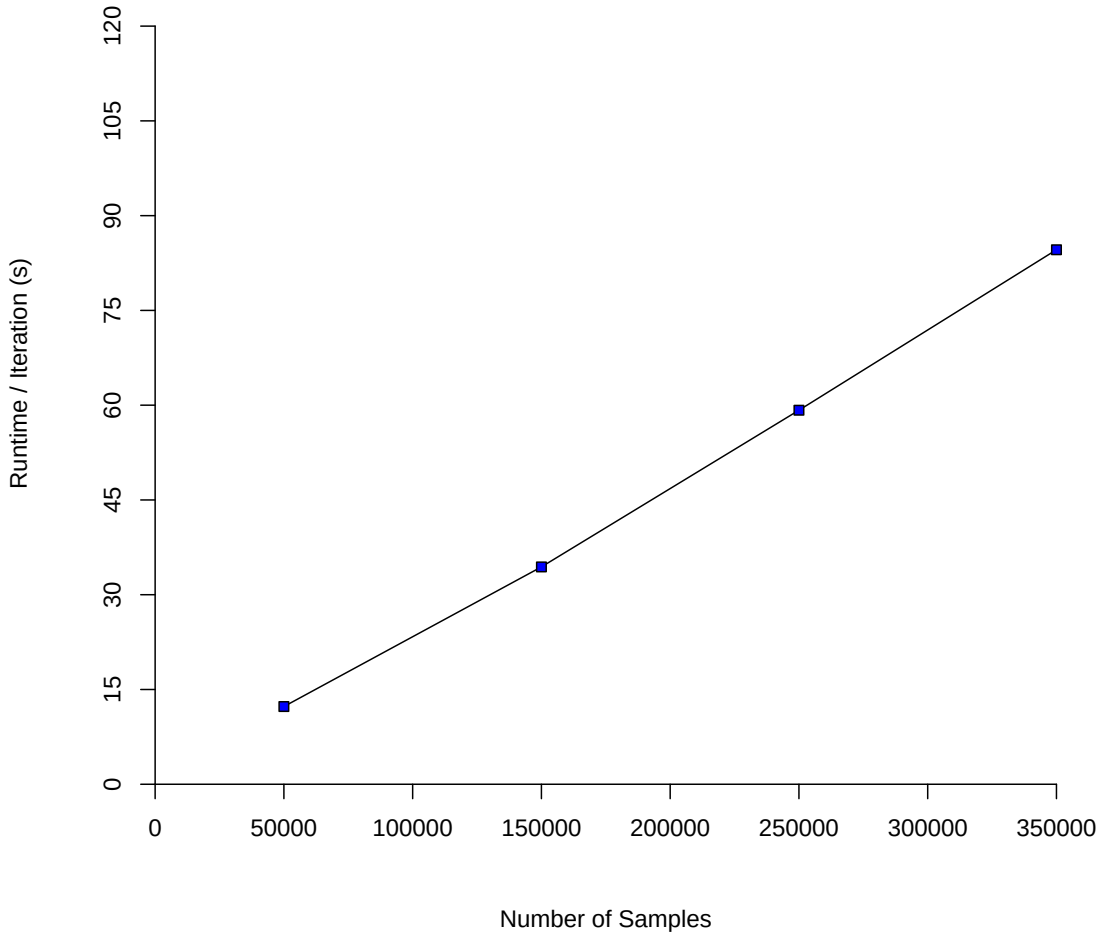

Figure 4.1: Runtime per iteration as the number of samples N is varied. Compare to Figure A.16 in Appendix A.6 for directly comparable runtimes. This serves as one of three baseline plots for the experiment.

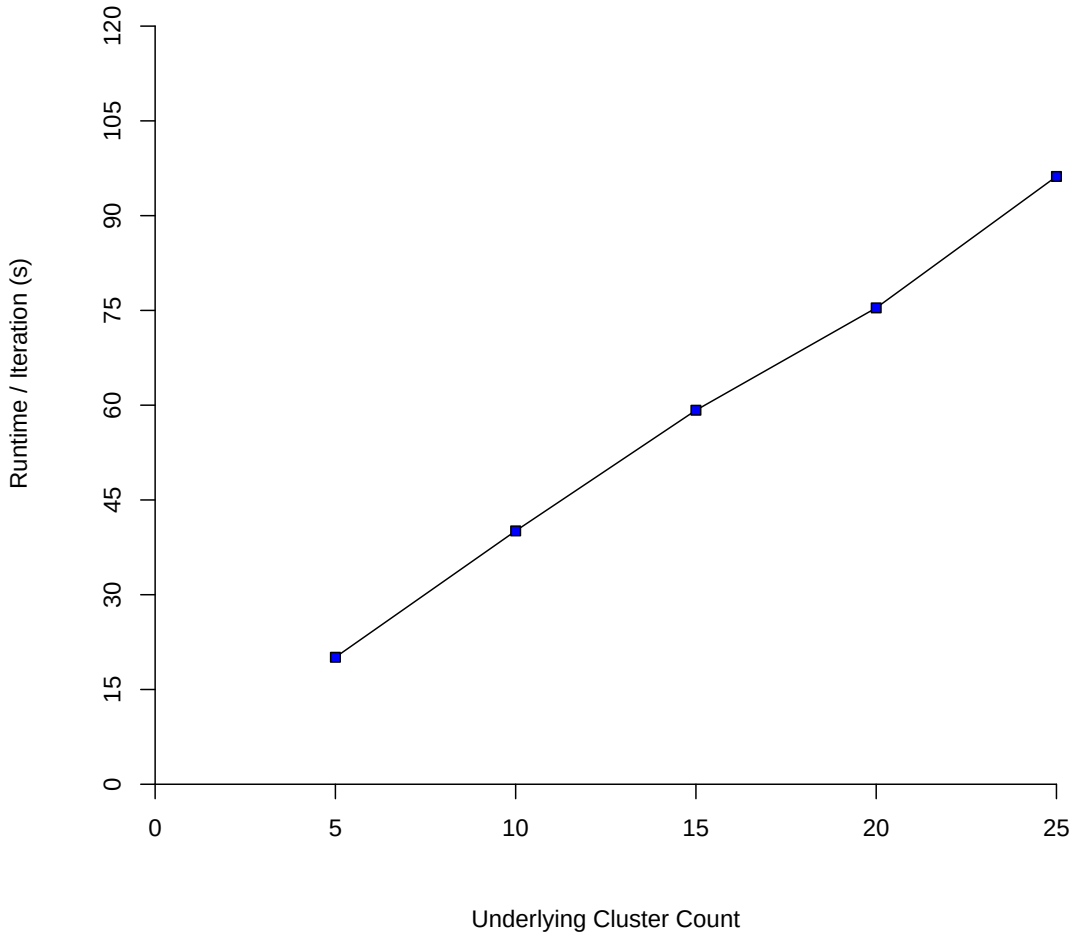

Figure 4.2: Runtime per iteration as the underlying source cluster count K is varied. Compare to Figure A.12 in Appendix A.6 for directly comparable runtimes. This serves as one of three baseline plots for the experiment.

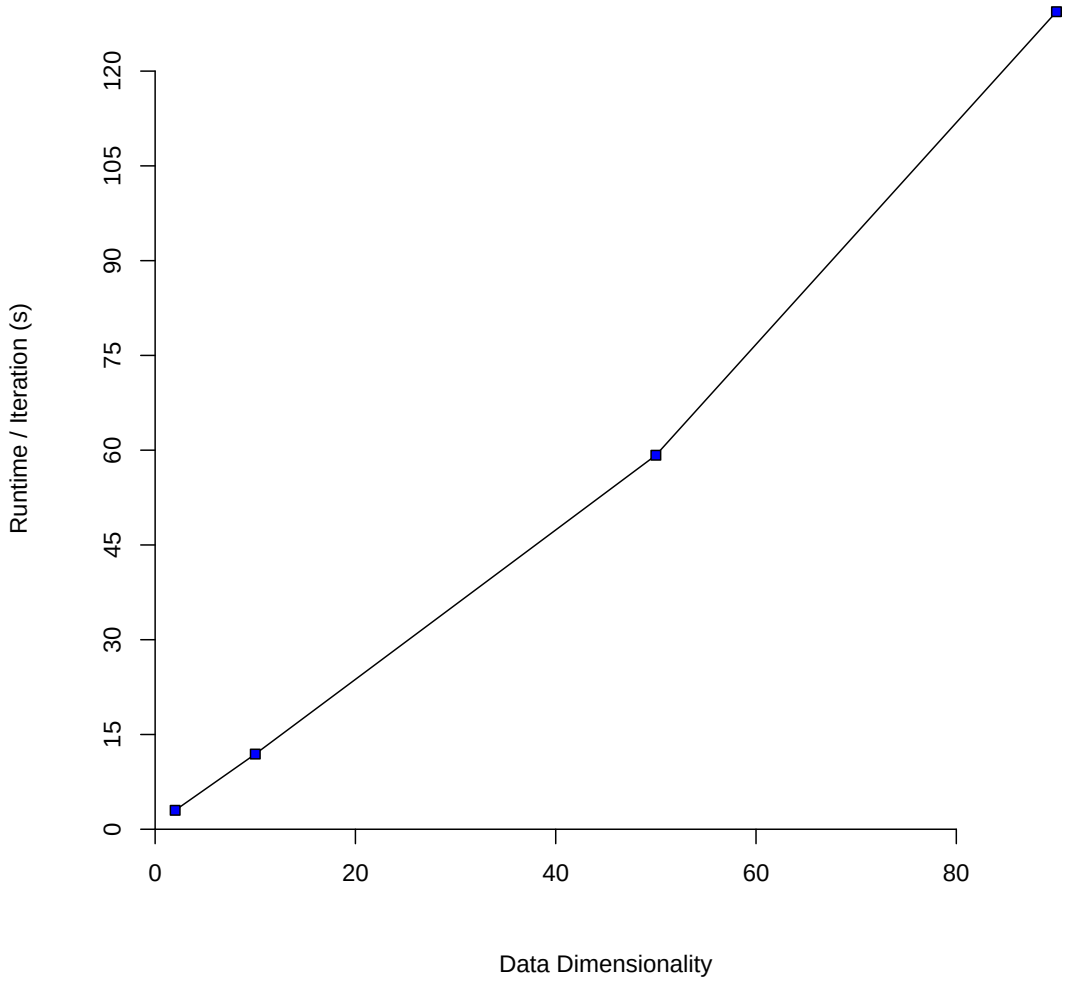

Figure 4.3: Runtime per iteration as the data dimensionality D is varied. Compare to Figure A.8 in Appendix A.6 for directly comparable runtimes. This serves as one of three baseline plots for the experiment.

The next Section compares these baseline results to the timings from the Hadoop implementation. This will determine if a crossover point is reached, and if cost optimality is attainable in a small research environment.

4.2 Distributed Results

The runtimes for the Hadoop implementation of EM are broken down into three sets of numbers: overhead time, which is the time spent outside of the map and reduce phases; map time, which is the mean time spent within the mappers for a given job; and reduce time which is the mean time spent in the reduce phase for a given job. The means are reported here because we find that the standard deviations of the times spent in the map and reduce phases are small. For those interested, these standard deviations are included in appendix A.

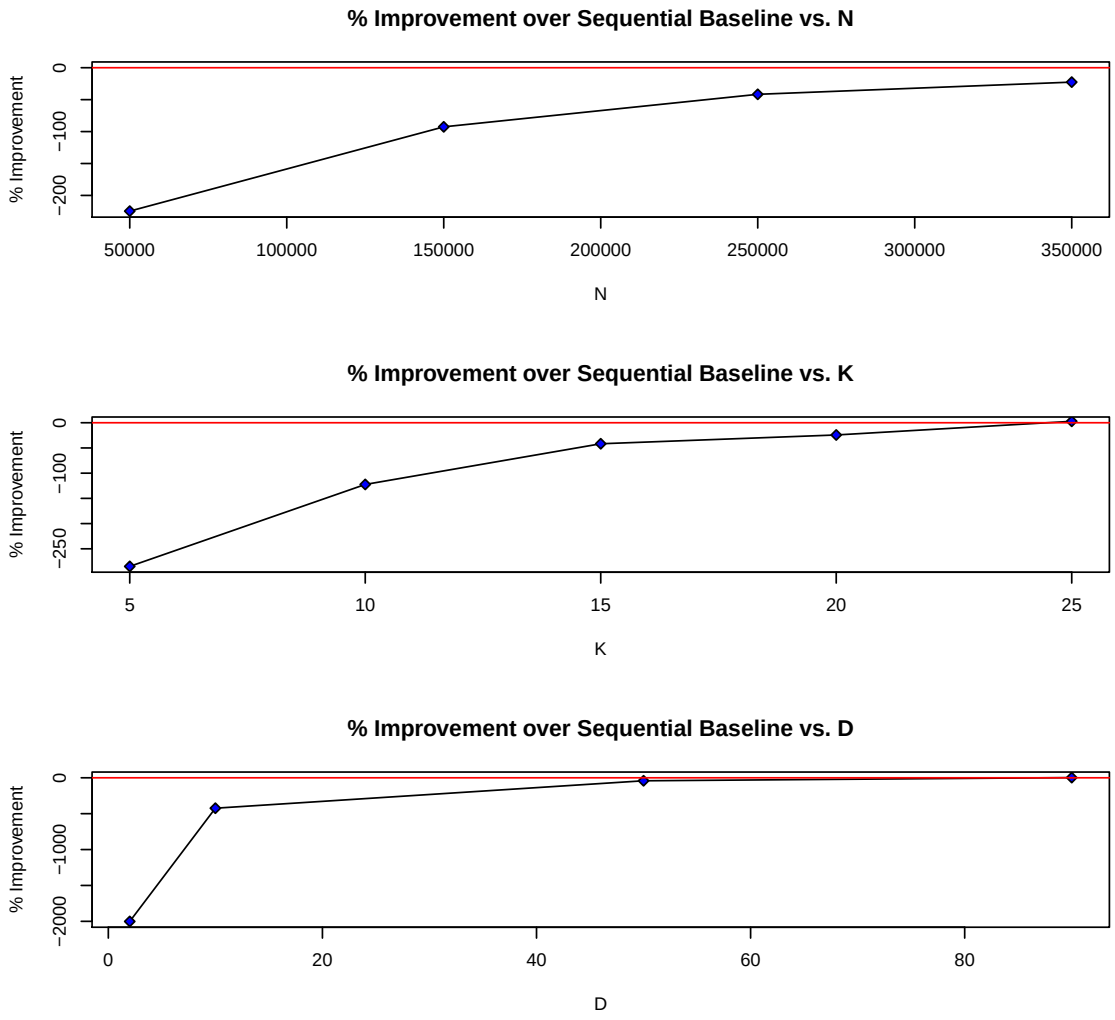

Figure 4.4: Percentage improvement vs. the sequential baseline as the number of samples N (top), the number of underlying clusters (and therefore processors) K (middle), and data dimensionality D (bottom) are varied. At best there is zero improvement over the sequential baseline which indicates that the system does not scale down well enough in this case.

In Figure 4.4 we visualize the total runtimes for the Hadoop implementation of the code as a percentage improvement over the sequential version when run on the same "large" scale data sets. We vary the number of samples N, the underlying source cluster count K, and the dimensionality of the data D within the testable limits of

our infrastructure, and as the figure shows, at the upper most testable limit of each set of experiments, we can only match equally the performance of the sequential version at best. This is partially due to the upfront overhead incurred when using Hadoop, and partially due to the network transfers which occur since the data consist of non-granular large records which require multiple 64MB blocks per record. Since there is no control over data distribution, and since record boundaries are not respected when splitting the data, no machines typically start out with a complete record to process. This transfer overhead along with the minimum upfront overhead of Hadoop is magnified by the iterative nature of the EM algorithm which requires the overhead to be paid in each iteration since the output data is written back to the HDFS and redistributed with no respect for record boundaries, and because each iteration is a new MapReduce job.

While we are able to at least hit the crossover point in our environment at the uppermost limit of our infrastructure, we are nowhere near cost optimal, and we find that the system does not scale down well enough in this case. Smaller datasets are also tested and visualized in the same manner. The plot for those data can be found in Appendix A.7.

From an iso-efficiency perspective, the observed speedup is less than or equal to one. Cost optimality is unachievable in this environment since the data can not be scaled large enough on the infrastructure used in this experiment. To see why, the next figure displays an analysis of runtimes broken down into overhead and workload. Additionally, the workload is also broken down into the workloads for both map and reduce.

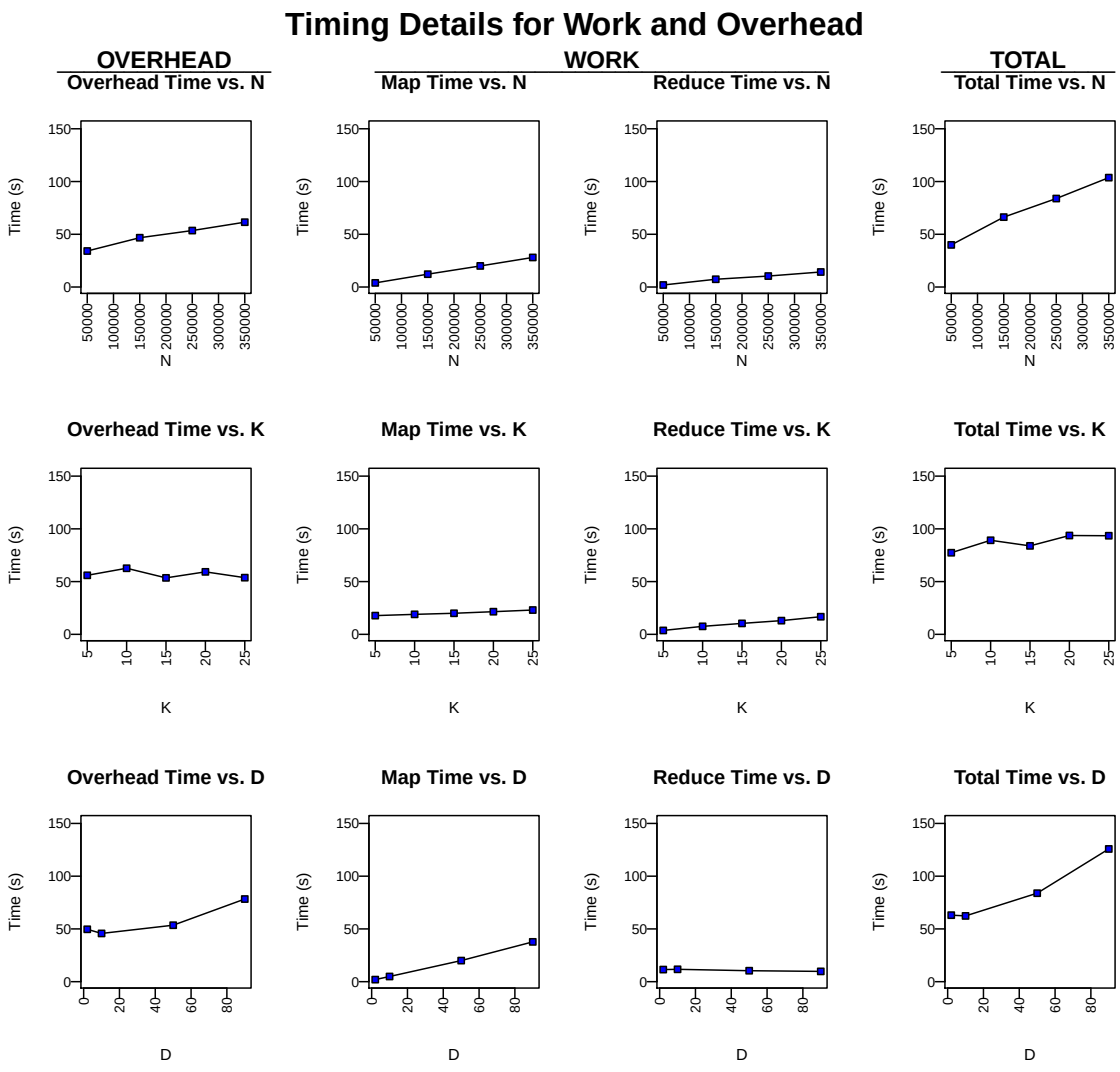

Figure 4.5: Runtimes for a single iteration of EM broken into overhead (left column), map and reduce phase work (center columns), and total runtime (right column) as the number of samples N (top row), underlying source cluster count K (center row), and data dimensionality D (bottom row) are varied. The work grows linearly with the data size despite quadratic algorithmic runtime bounds. This implies the computation is IO bound. The overhead accounts for most of the runtime. Since the work scales linearly, the system is scalable, and cost optimality may be possible on much larger data sets.

Figure 4.5 shows an overview of the runtimes for the Hadoop implementation of EM as the number of samples N, the underlying source cluster count K, and the data dimensionality D are each varied. Larger versions of these plots are included in Appendix A.6. The figure shows that in every case, the majority of the total runtime

is accounted for in the overhead. As N and D grow, the overhead also grows, but as K is increased, the overhead stays constant. In the map phase runtime increases linearly as N, K and D are increased despite quadratic algorithmic time bounds. This indicates that the runtime is driven by the space complexity of the data, meaning the map phase is IO bound. The reduce phase runtimes increase linearly in N, and K, but remain constant in D as expected since D plays no part in the reduce. Smaller datasets are also tested and visualized in the same manner. The plot for those data can be found in Appendix A.7.

As parameters increase, overhead appears to increase linearly, and in the testable data range the work increases linearly as well. From an iso-efficiency perspective this means that the ratio between work and overhead stays constant, which means the algorithm does scale well, and there does exist a point where this system becomes efficient. The large minimum overhead due to iteration effectively sets a requirement for a large minimum workload to obtain good efficiency, but the infrastructure used in this experiment cant support that workload.

Chapter 5

Conclusion

Hadoop is useful when the problem lends itself to the Map/Reduce framework. Multiple iterations of Map/Reduce are required for implementing EM using Hadoop, and those iterations are a major contributing factor to Hadoop not scaling well in a small research environment. Each iteration requires large non-granular records to be split without respect for record boundaries resulting in overhead from network traffic, combined with the minimum startup overhead for each iteration.

Iso-efficiency tells us that at some point we can find an efficient combination of workload and processors such that EM will be efficient. The workload cannot be scaled up large enough in our environment due to data dependencies that can not be modeled in Hadoop, which result in too much network overhead during execution. The infrastructure itself is a small research cluster comparable consisting of a single rack of machines. In small research environments, sequential implementations of algorithms such as EM may take weeks on sizable datasets. It is of great interest to find a framework that will allow easy parallelization of these projects to allow for more timely execution. This paper examines the use of an easily parallelizable algorithm which is computationally expensive and widely used in many fields of research with the goal of examining the performance of Hadoop in a small research environment and analyzing any possible issues that may prevent such a framework from working well.

5.1 Future Work

This research has brought up several questions which remain unanswered. The iso-efficiency analysis shows that at the testable ranges of data, computation is largely IO bound. It may be the case that on data sets orders of magnitude larger this will no longer be the case due to the increased workload. If that is so, then the iso-efficiency analysis will change, and what is currently a very scalable algorithm may no longer be so. Speedup may saturate before reaching an acceptable level of efficiency, or either the workload or overhead may begin to grow exponentially with the number of processors.

We are unable to test these larger data ranges because the underlying file transfers used by Hadoop begin to fail at larger data ranges. Since the data consists of very few, but very large records, no one machine has a complete record to start with since data splits do not respect record boundaries. Since the entire cluster starts off with truncated files, there is a burst of network traffic at the start of each iteration as Hadoop transfers the required pieces. This burst of network traffic may be a contributing factor to the failed network transfers which prevent larger data-sets from being used. It is also possible that this may be an issue with disk quotas, bandwidth throttling, or some other administrative issue. These experiments may be run using much larger data sets that increase workload to processor capacity on a small research cluster which does not have these limitations. One possible workaround may be to use a binary data format which allows for larger datasets in the presence of these limitations. Another possible solution is to directly support iteration, where the $< key, value >$ pairs are sent from the reducer straight back into another MapReduce cycle, bypassing the HDFS altogether. This would eliminate the issue of non-granular records being split and reformed each iteration.

It is also yet to be seen how well other frameworks will perform using the same EM implementation. Hadoop is one of many distributed systems available, and other frameworks which support iteration or modeling of data dependencies may perform

better. The Google Map / Reduce system [12] differs from Hadoop in many ways including support for iteration, although the Google Map / Reduce system is proprietary and is not available for detailed performance testing. The Granules system [10, 11] supports Map / Reduce as well as iterative, periodic, and data driven models. A survey of several systems would be beneficial to the community to determine the pro's and con's of each and to allow for informed choices.

Appendix A

A.1 Mapping Phase Means and Std. Deviations

Table A.1: Mean Map times and corresponding standard deviations for each iteration of EM on various datasets. The standard deviations are small compared to the means, making the mean a good indicator of the total time spent in the map phase by each mapper.

N	K	D	Mean Time (s)	Std. Deviation (s)
50000	15	50	3.90	0.15
150000	15	50	12.16	0.37
250000	5	50	17.75	0.61
250000	10	50	18.95	0.59
250000	15	2	1.94	0.07
250000	15	10	4.90	0.17
250000	15	50	19.97	0.62
250000	15	90	37.73	1.14
250000	20	50	21.47	0.68
250000	25	50	23.04	1.02
350000	15	50	28.01	0.69

The analysis in section 4 reports the mean time spent in the mapping phase for each iteration on each data set. That analysis uses only the means in visualizing the results since the standard deviations are relatively small making the means a good representation of all iterations on a given data set. The standard deviations are reported here along side of the corresponding mean values used in the analysis.

A.2 ATLAS CBLAS Matrix Multiple Runtime Anomaly

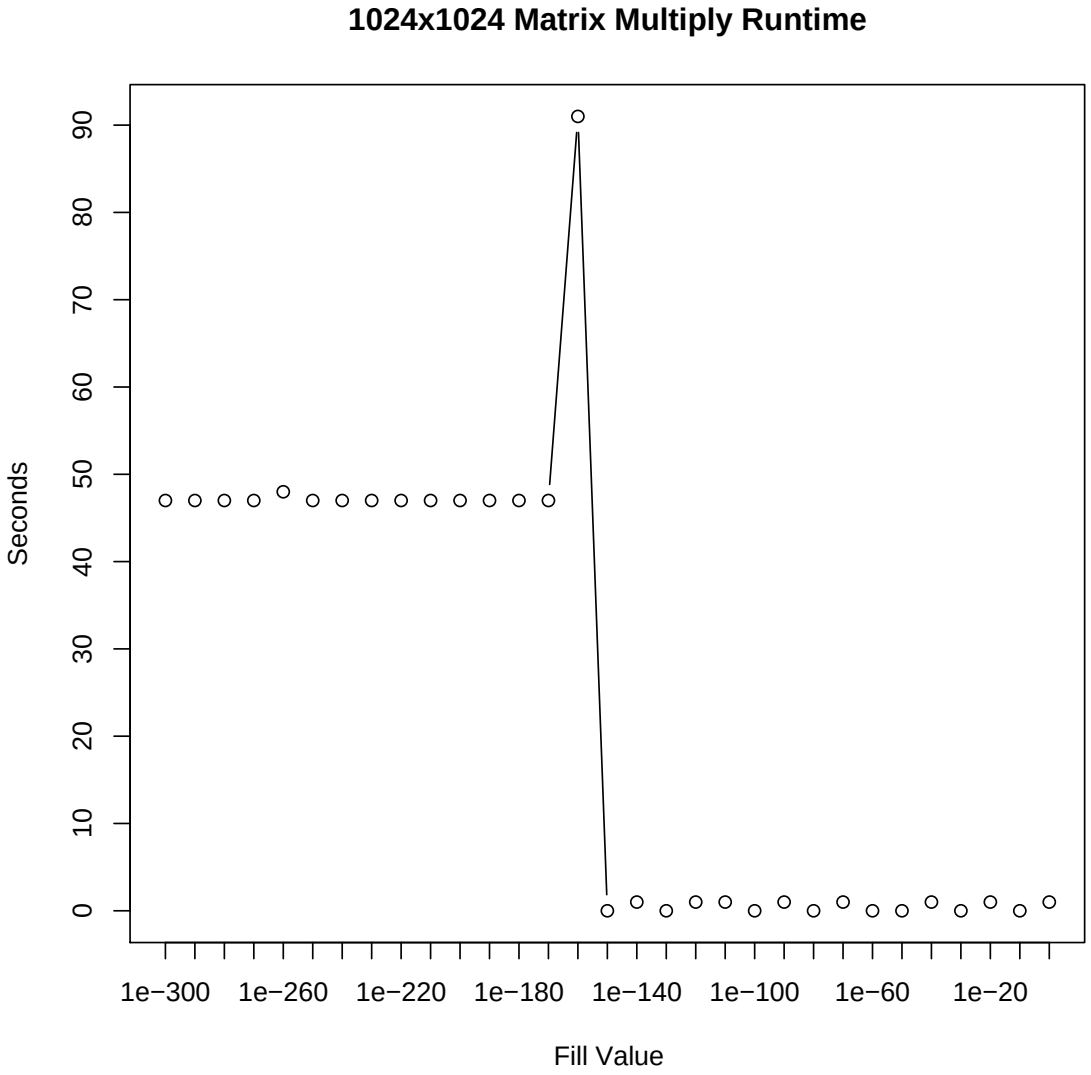

Figure A.1: ATLAS CBLAS matrix multiply runtimes with fill values ranging from 1e-0 to 1e-300. The runtime for a loop containing the matrix multiply jumps by an order of magnitude when the matrices contain values smaller than 1e-150.

The ATLAS optimized CBLAS matrix multiply significantly slows down when the matrix contains very small numbers around 1e-150 or so. To demonstrate this phenomenon, we constructed a pair of 1024x1024 matrices and timed a looped CBLAS matrix multiply operation while varying the values used to fill the matrices. The

values ranged from 1e-0 down to 1e-300. Figure A.1 shows the results. The matrix multiply operations on values larger than 1e-150 took less than 4 seconds, while the same operations on matrices containing values of 1e-150 and smaller took more than ten times as long. We chose not to pursue the reason for this, but we did modify the EM code to round matrix values in the normalized probability matrix whose value is equal to or smaller than 1e-150 to zero. This resulted in a substantial boost in run time with no discernible impact on the output of the algorithm. This is because normalized probabilities of that magnitude are negligible and indicate that the corresponding data point is unlikely to be a member of the corresponding cluster anyway.

A.3 Sample Randomly Generated Dataset

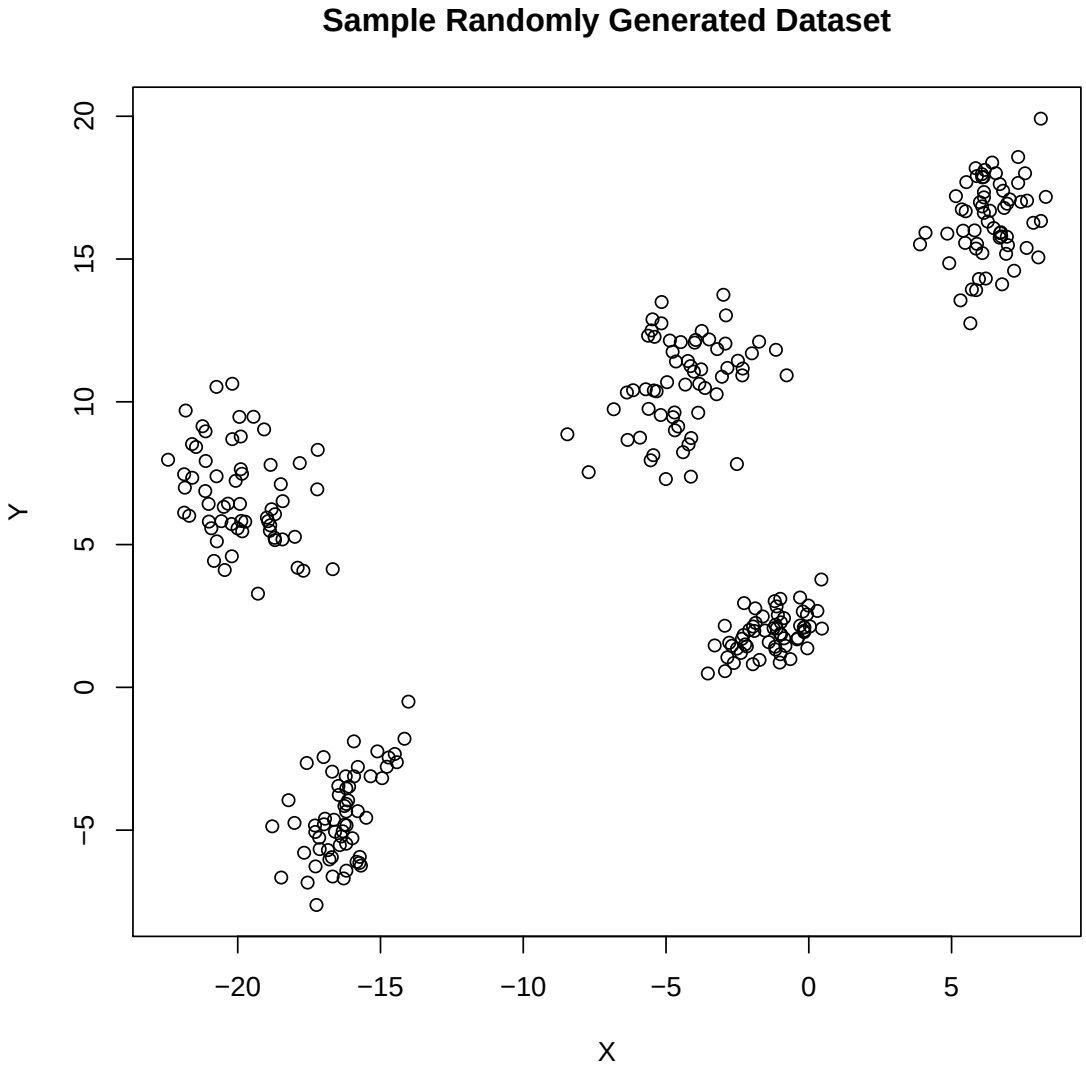

Figure A.2: A sample 2D Dataset randomly generated using the data generation code developed for thesis. See Section 3.1 for details.

The data set shown in Figure A.2 is an example of a 2D dataset generated by the method described in Section 3.1. A 2D dataset is chosen for this example because it is easily visualized. A clustering of this data is presented in Appendix A.4 for reference.

A.4 Sample Clustering Using EM

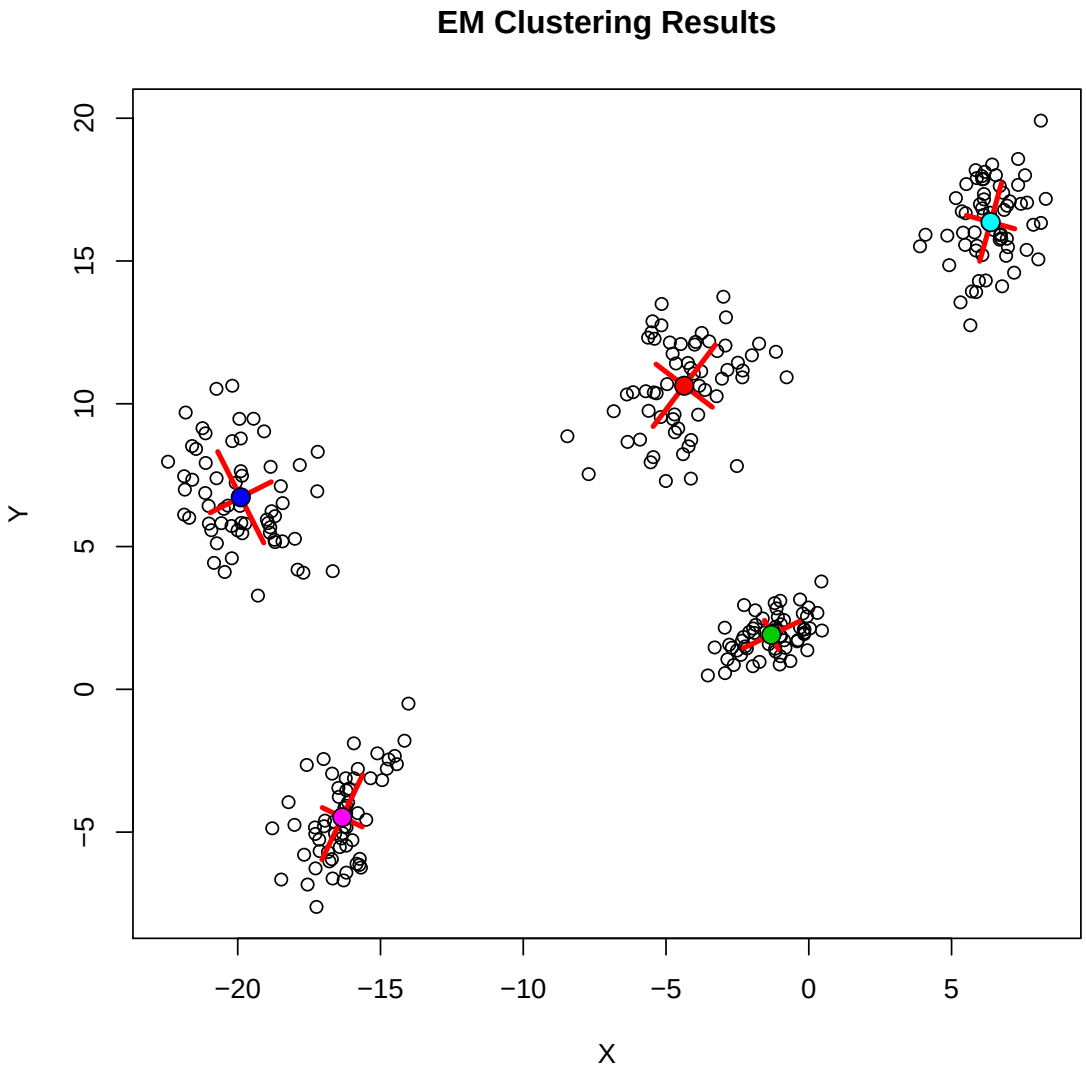

Figure A.3: The same sample dataset as displayed in Appendix A.3, now clustered using the EM code developed for this thesis. The centers, and eigenvectors are displayed to represent the multivariate normal distributions found by EM to have the highest probability of generating the data. The eigenvectors are scaled to one standard deviation from the cluster center.

The dataset shown in Figure A.3 shows the resulting clustering of the data from Appendix A.3 by the EM code used in this thesis. The dots and lines represent the means and eigenvectors of the multivariate normal distributions which have the

highest probability of having been the process which produced the data. The length of the eigenvectors are scaled to represent one standard deviation from the mean.

A.5 Sample Clustering Using EM (Hard)

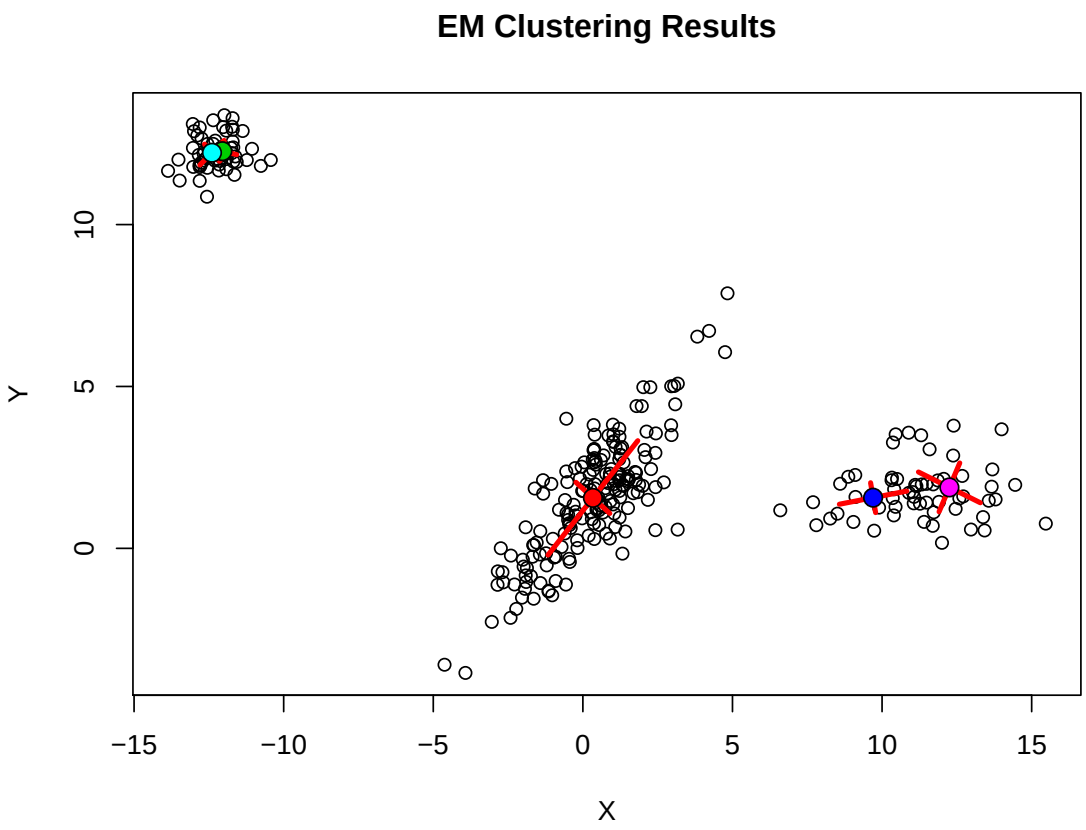

Figure A.4: A sample 2D Dataset randomly generated using the data generation code developed for thesis clustered using the EM code developed for this thesis. In this case some of the clusters are overlapping making this a hard data set to correctly cluster. As expected, the resulting model does not match the true underlying source clusters from which the data is generated. The centers, and eigenvectors are displayed to represent the multivariate normal distributions found by EM to have the highest probability of generating the data. The eigenvectors are scaled to one standard deviation from the cluster center.

This is another example of a randomly generated data set clustered using EM. In this example, five Gaussian processes are used to generate the data, but by chance some

of them are overlapping. The resulting fit does not match the original processes used to generate the data, and is shown as an example of EM on a harder data set.

A.6 Fullsized Work and Overhead Plots

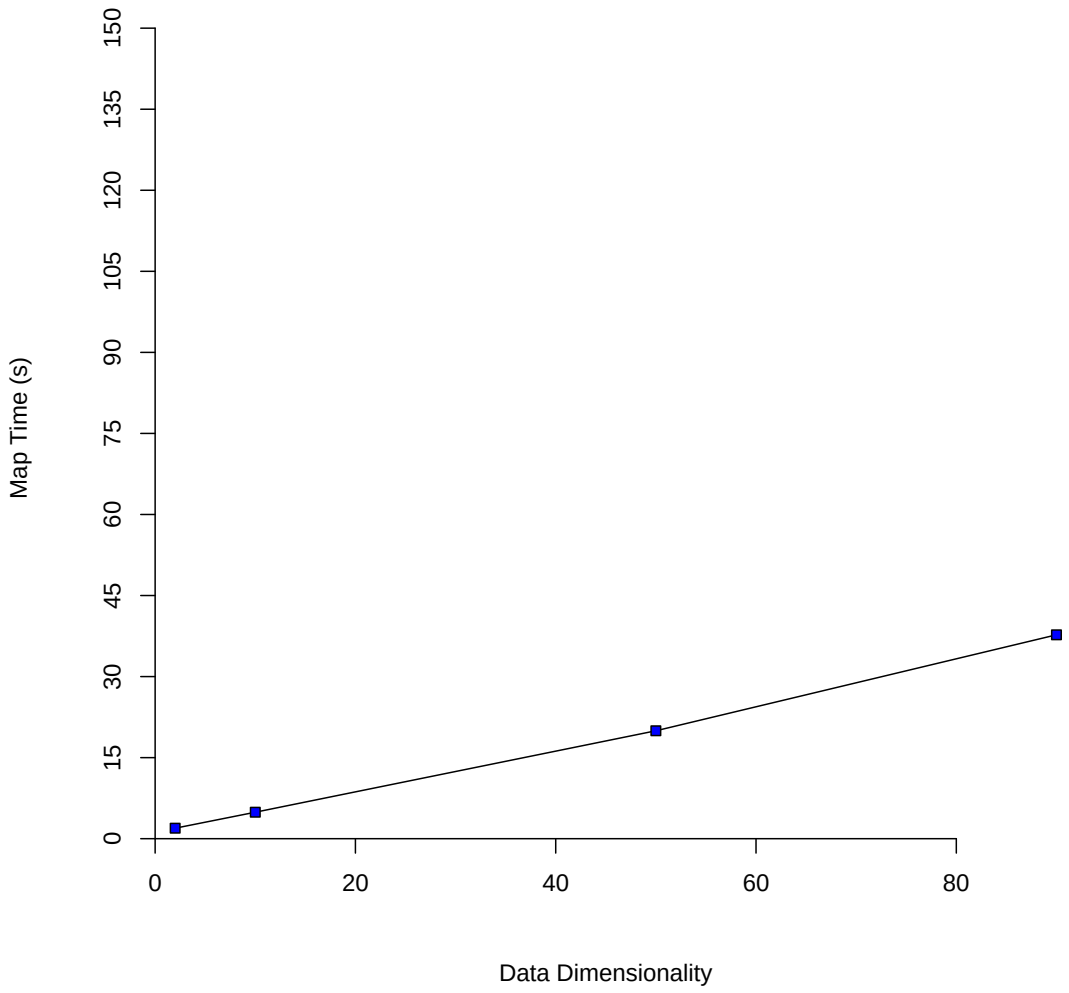

Figure A.5: Time spent in the mapping phase per iteration as data dimensionality D is varied. The Mapping phase contains algorithms which are quadratic in D, but the growth appears linear which suggests the runtime is IO bound.

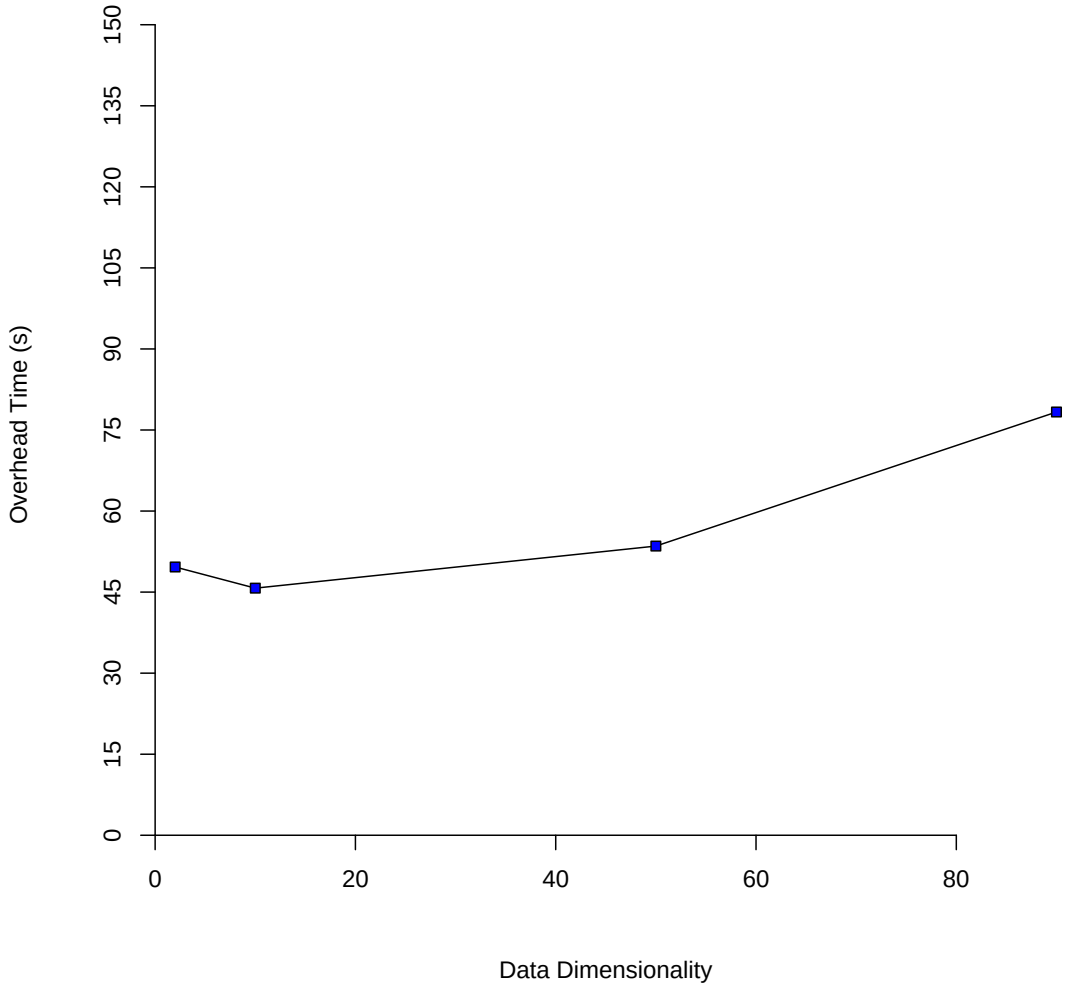

Figure A.6: Time spent in overhead per iteration as data dimensionality D is varied. The overhead appears to grow linearly, the slight curve seen in this figure is most likely due to noise. Subsequent runs of the experiment result in a variety of noisy linear plots. This may be due to random network delays, or the fact that Hadoop is implemented in Java, and is subject to random garbage collection delays.

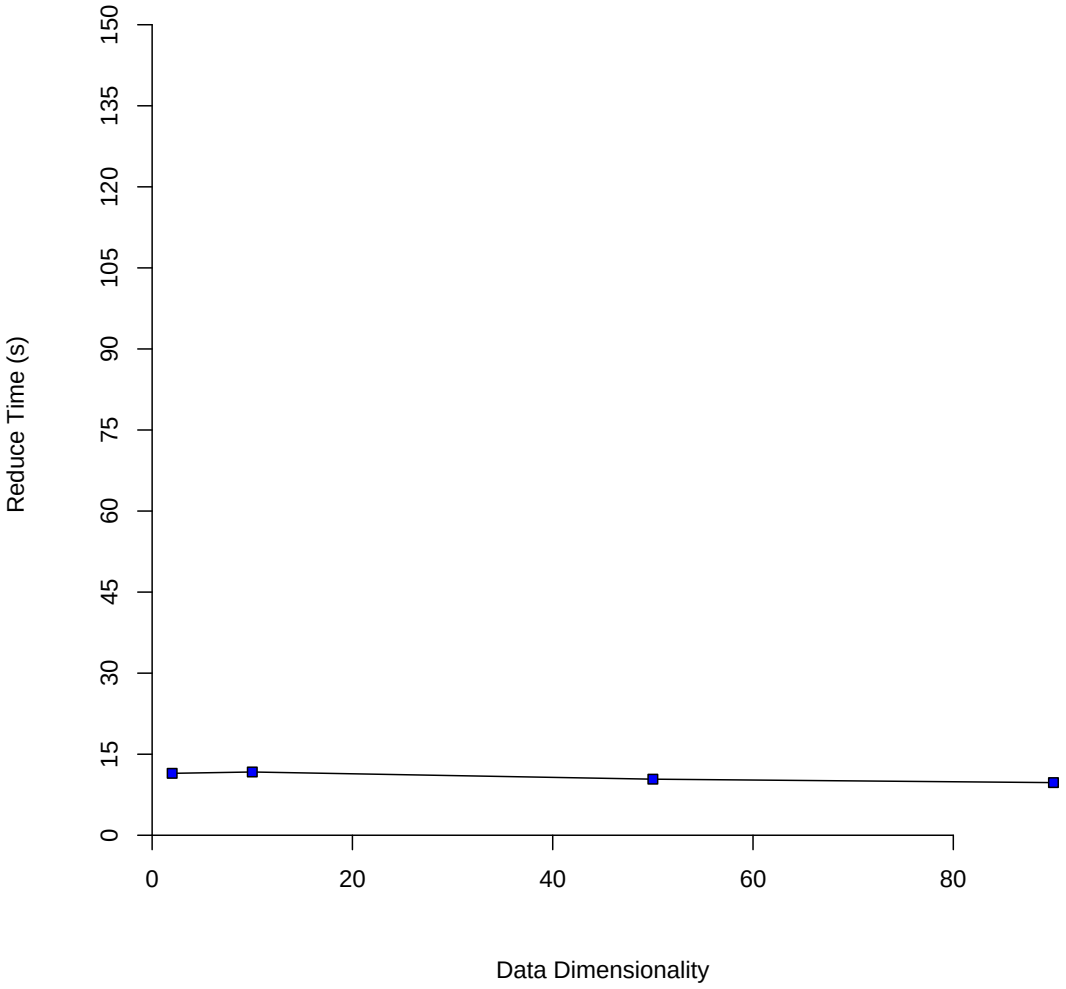

Figure A.7: Time spent in the reduce phase per iteration as data dimensionality D is varied. Data dimensionality plays no role in the reduce phase since the reducer only deals with a NxK matrix of probabilities. As expected the time spent in the reduce phase is unaffected by varying data dimensionality.

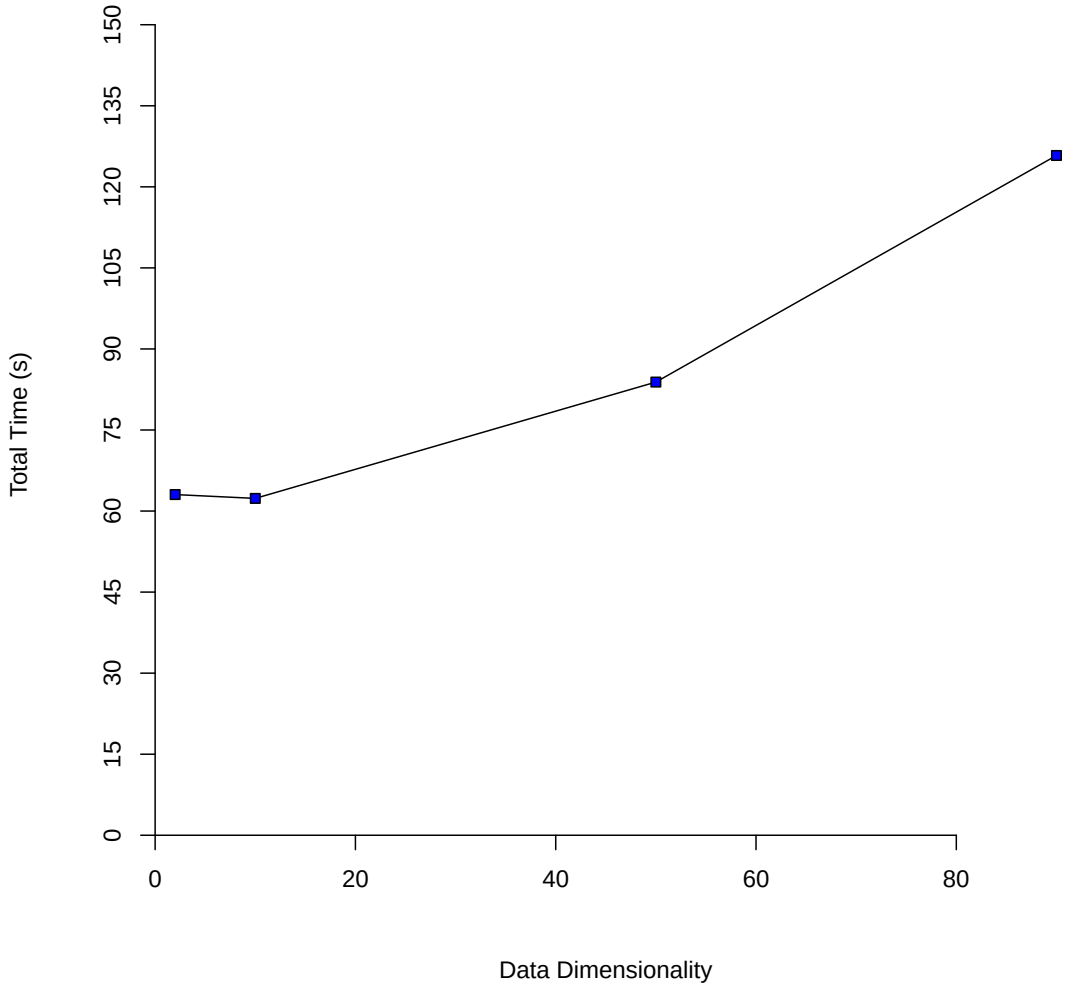

Figure A.8: Total time spent per iteration including overhead, map, and reduce time as data dimensionality D is varied. The total runtime is largely dominated by overhead.

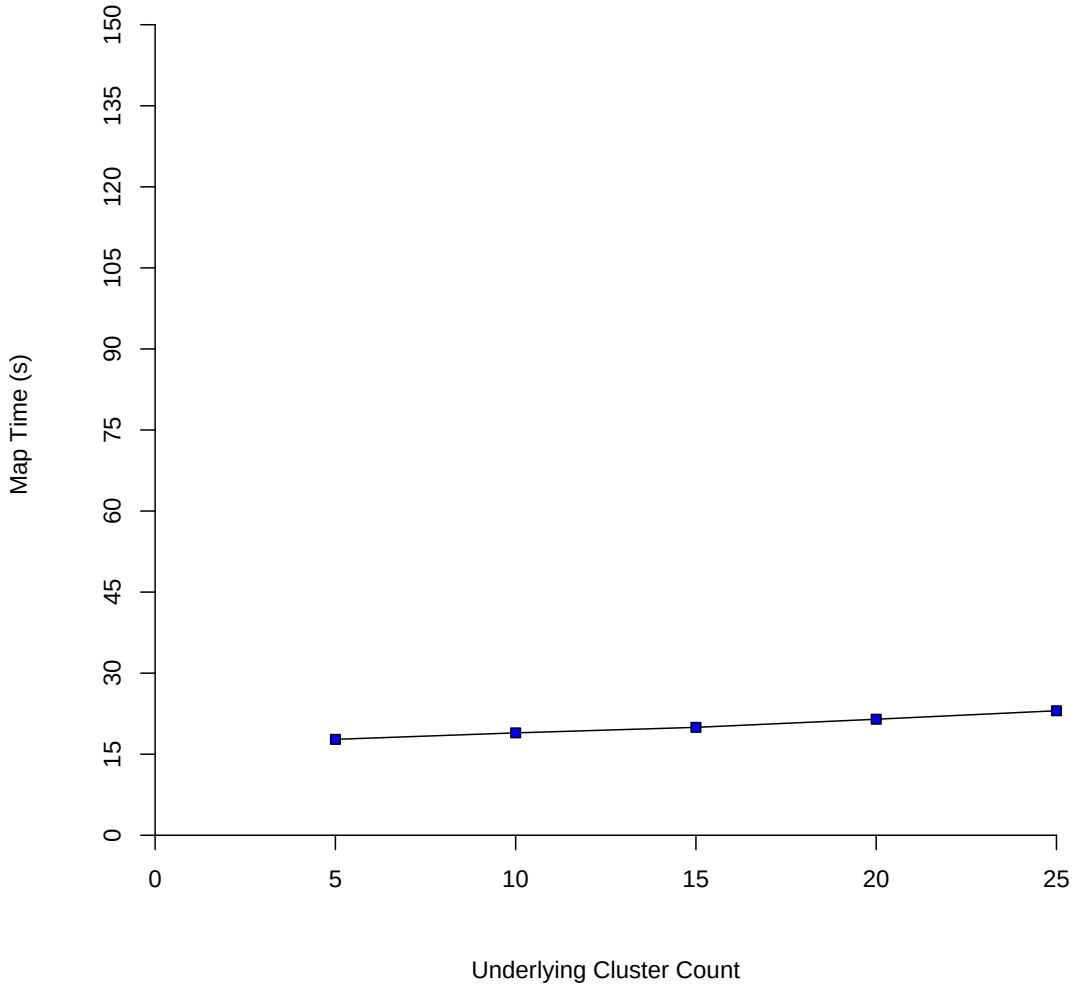

Figure A.9: Time spent in the mapping phase per iteration as the underlying source cluster count K is varied. The parameter K determines the inherent parallelizability of the EM algorithm given the data, and therefore the number of processors used to distribute the work. Since increasing K both linearly increases workload in the mapping phase and increases parallelism of the algorithm we see only a slow linear growth in runtime.

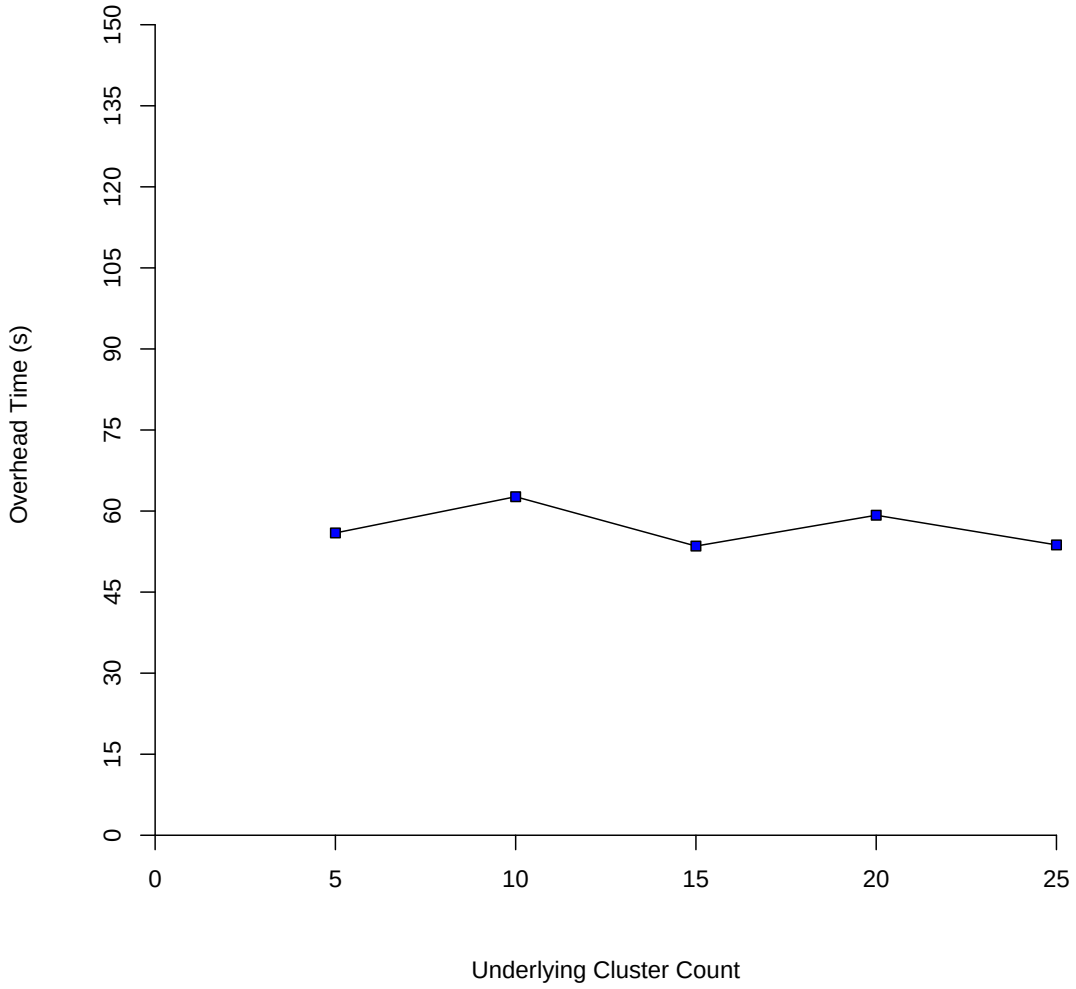

Figure A.10: Time spent in overhead per iteration as the underlying source cluster count K is varied. The parameter K determines the inherent parallelizability of the EM algorithm given the data, but has no effect on the data set size. As expected, we see no change in overhead as K is varied since there is no change in data size.

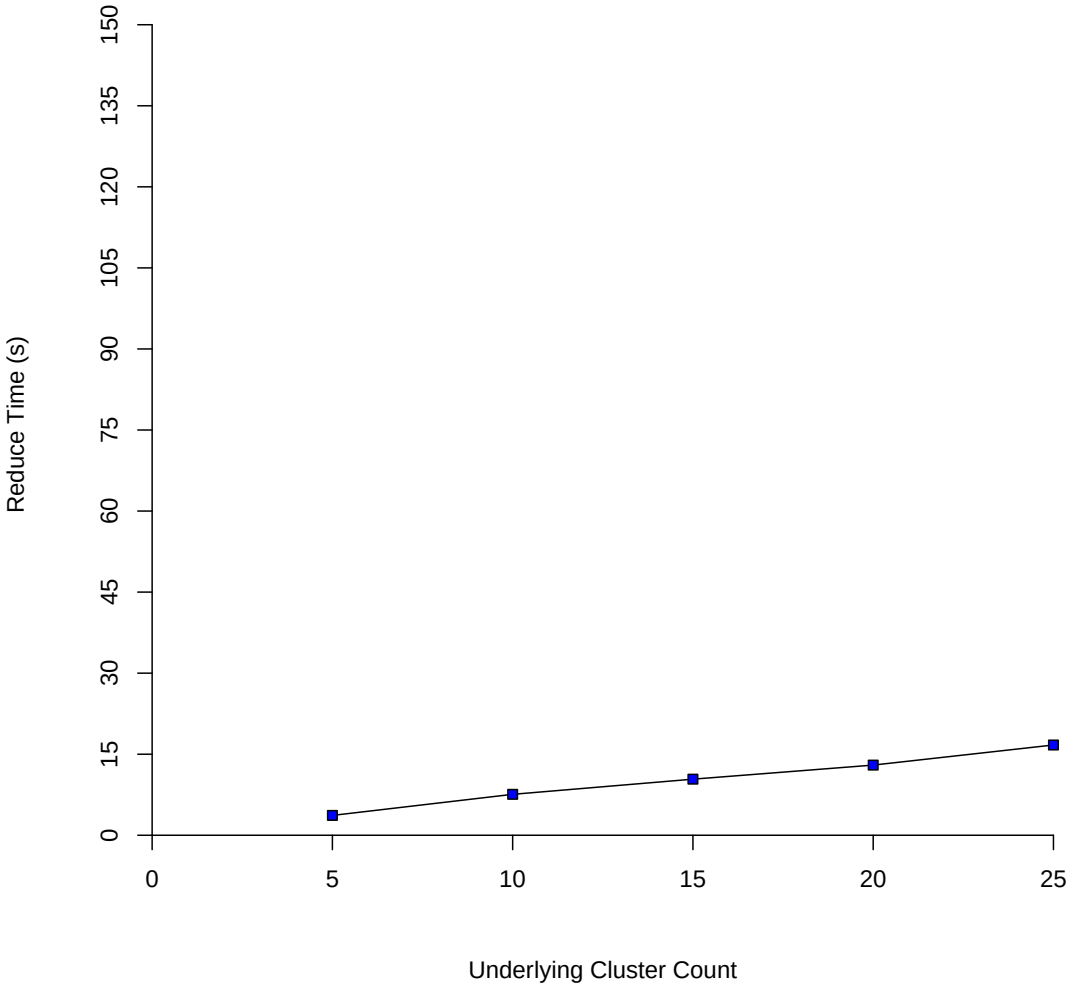

Figure A.11: Time spent in the reduce phase per iteration as the underlying source cluster count K is varied. The parameter K determines on dimension of the resulting NxK probability matrix which is processed by the reducer. The reducer phase is linear in K which is consistent with this figure although the reduce phase is most likely IO bound.

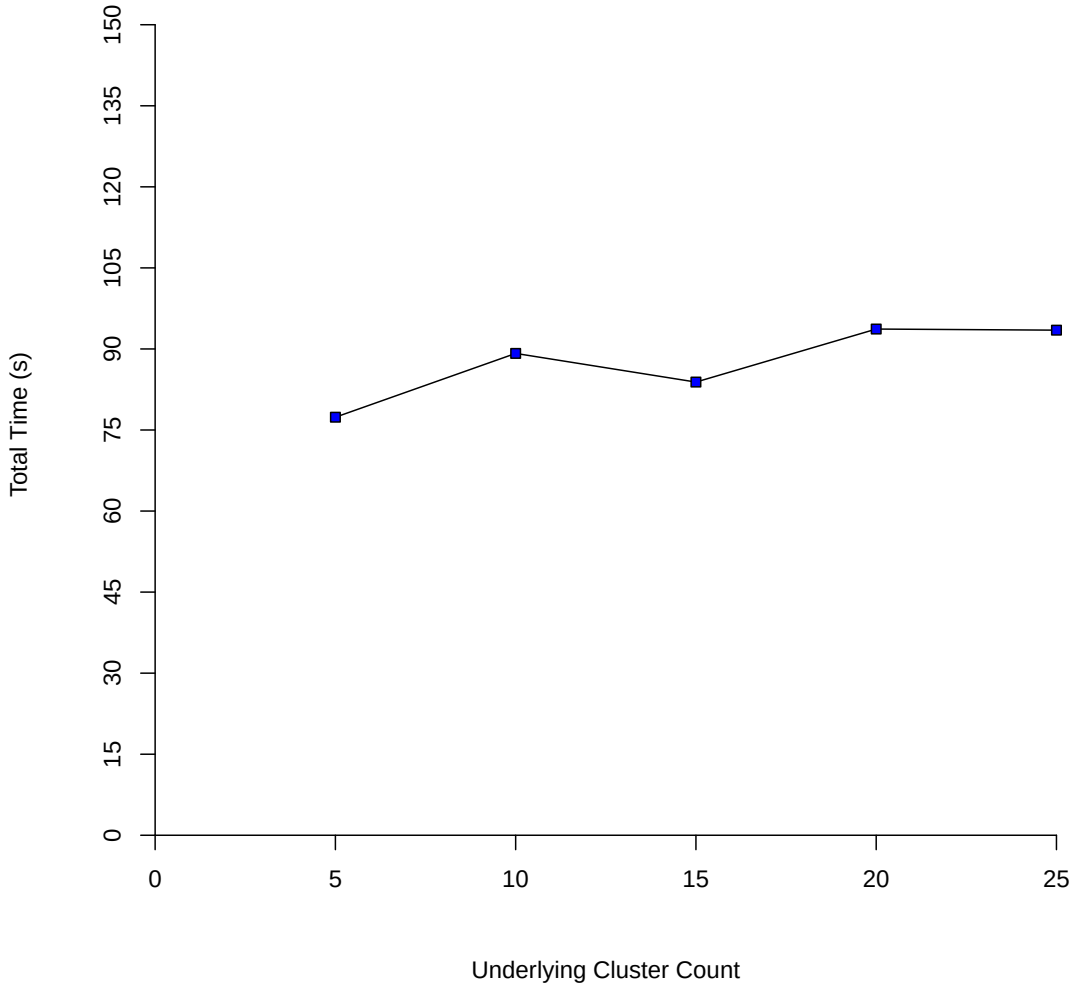

Figure A.12: Total time spent per iteration including overhead, map, and reduce time as the underlying source cluster count K is varied. The total time is dominated by overhead, but grows very slowly. Since increasing K linearly increases workload, has no effect on overhead, and increases the inherent parallelism of the algorithm, we see that increasing K is nearly free from a runtime perspective when compared to the other parameters.

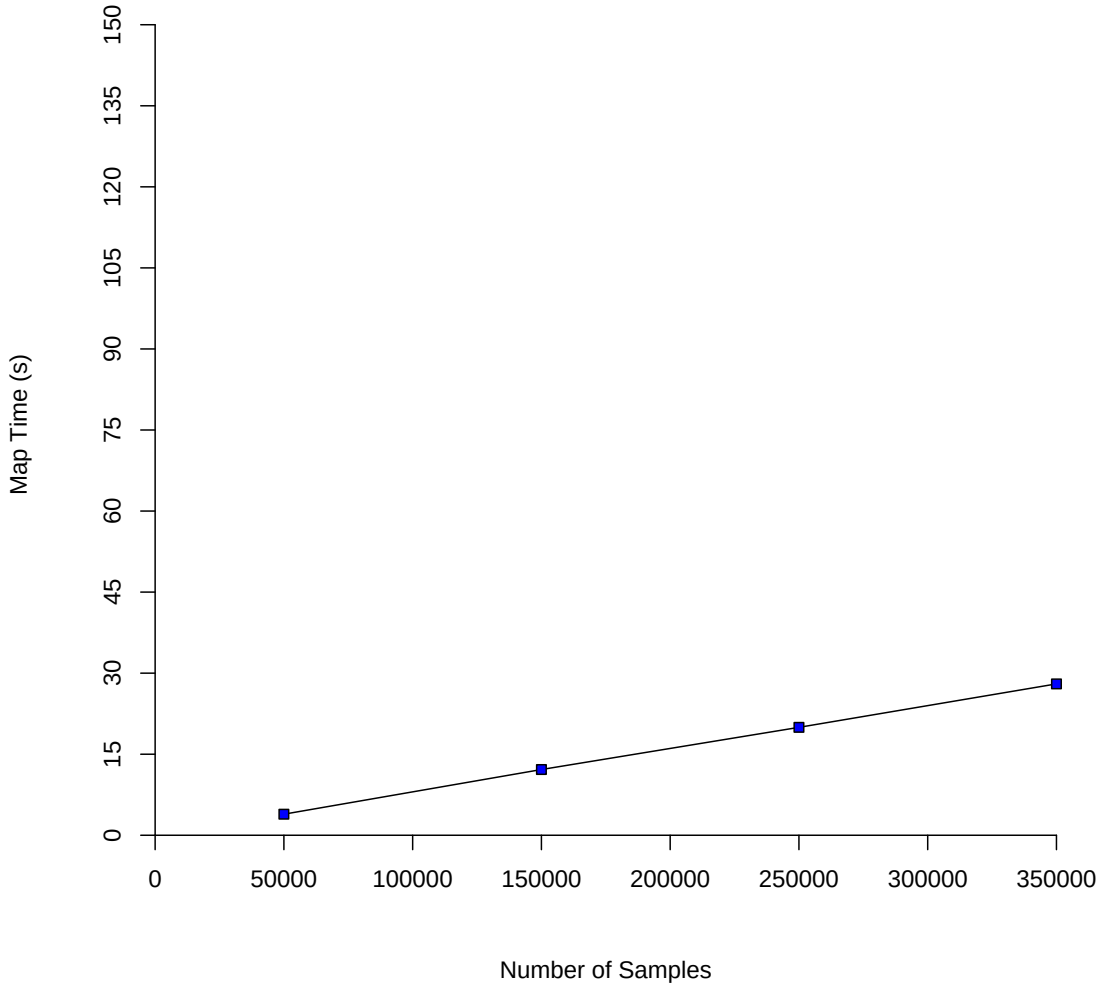

Figure A.13: Total time spent in the mapping phase per iteration as the number of samples N is varied. The Mapping phase contains algorithms which are linear in N which is consistent with this plot.

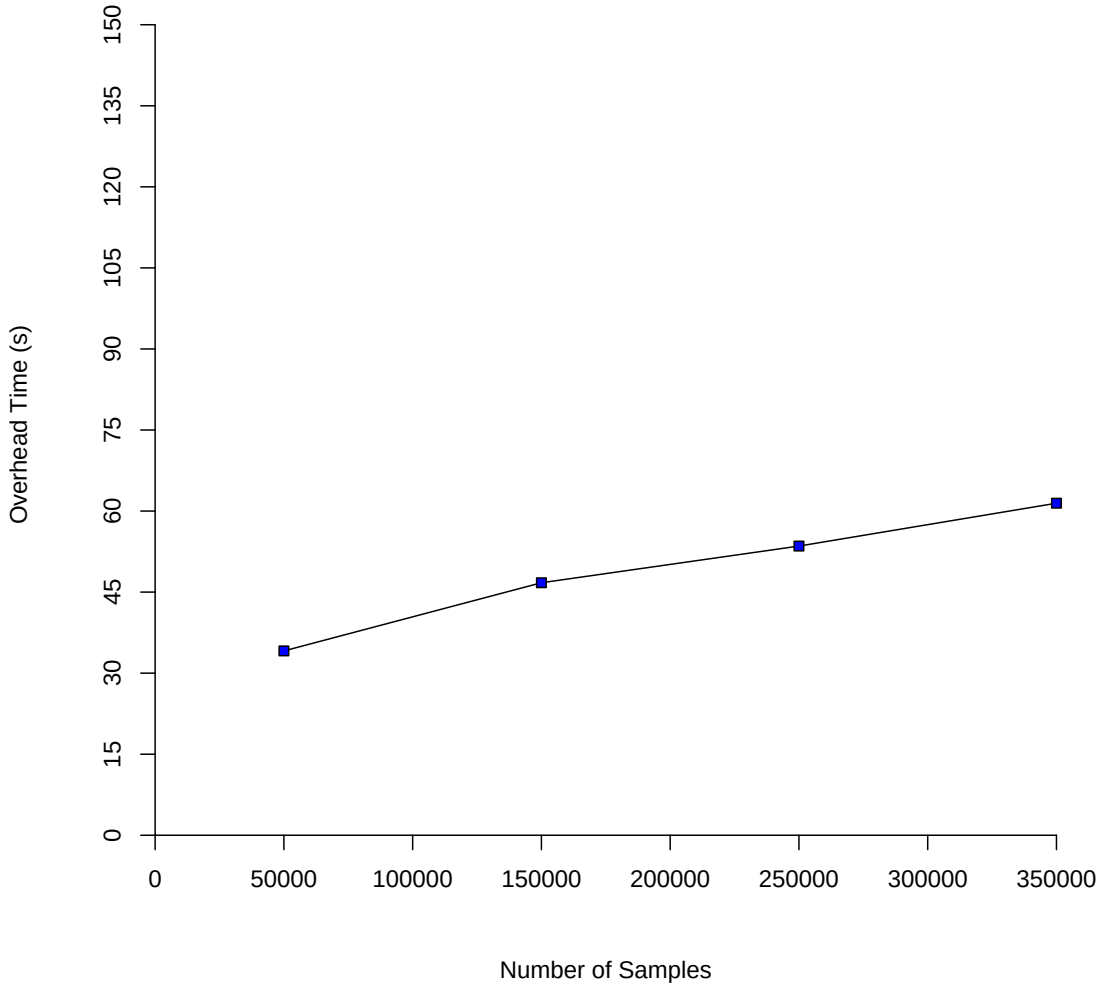

Figure A.14: Total time spent in overhead per iteration as the number of samples N is varied. The size of the NxD dataset grows linearly in N and as expected, the overhead grows linearly as well.

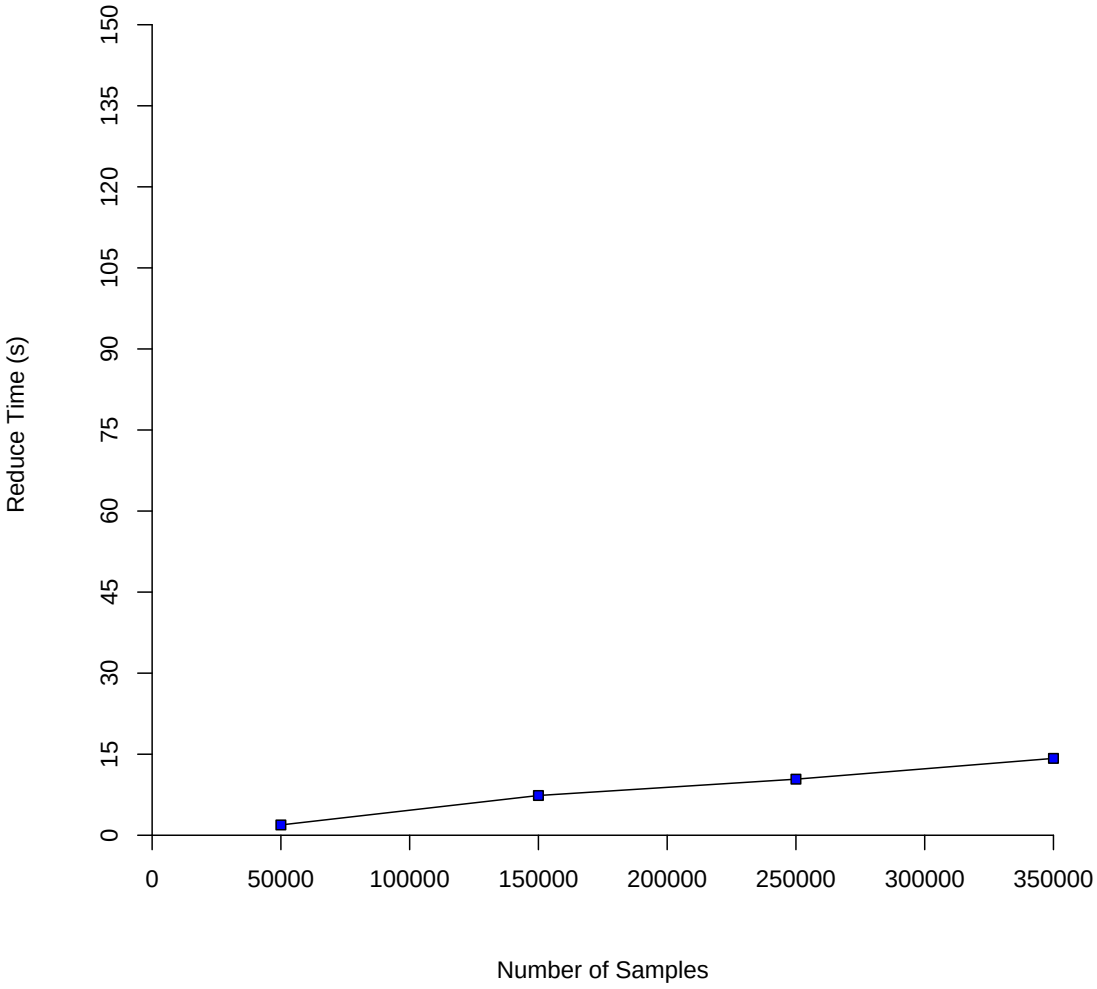

Figure A.15: Total time spent in the reduce phase per iteration as the number of samples N is varied. The size of the NxK probability matrix grows linearly in N and as expected, the time spent in the reducer grows linearly as well.

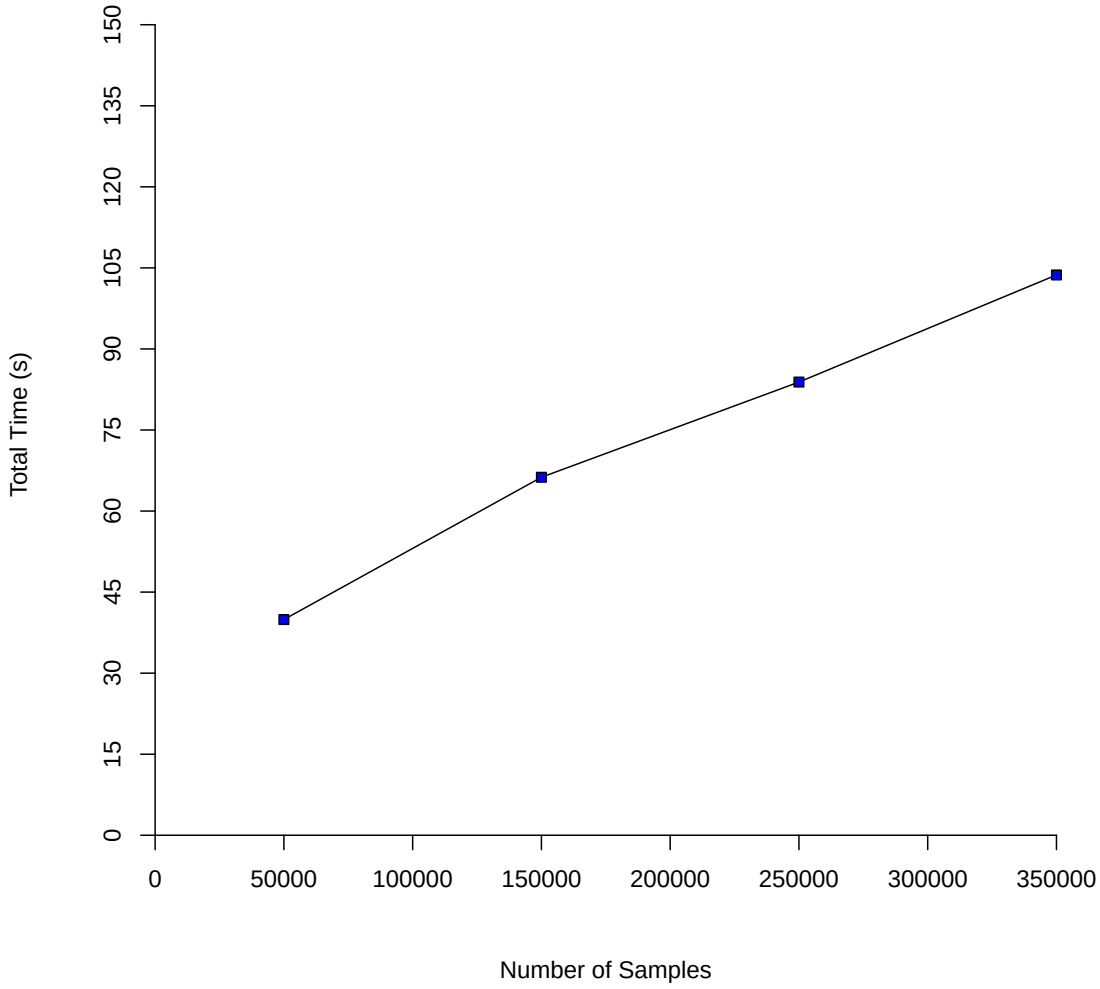

Figure A.16: Total time spent per iteration including overhead, map, and reduce time as the number of samples N is varied. The total time is dominated by overhead, but is still linear since the workload and overhead are both linear.

A.7 Smaller Data Set Results

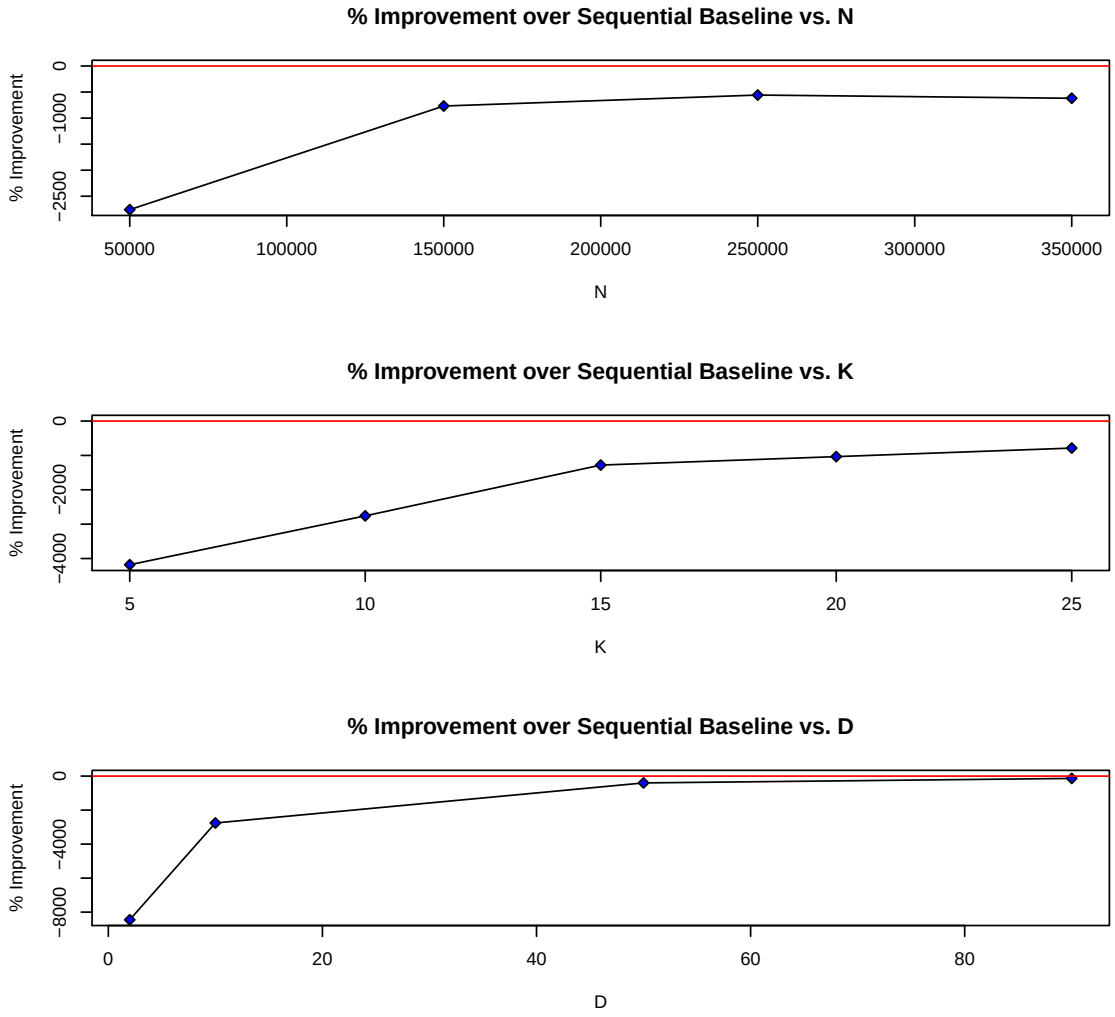

Figure A.17: Percentage improvement vs. the sequential baseline as the number of samples N (top), the number of underlying clusters (and therefore processors) K (middle), and data dimensionality D (bottom) are varied. This is the same experiment which produced the results in Chapter 4repeated on smaller datasets.

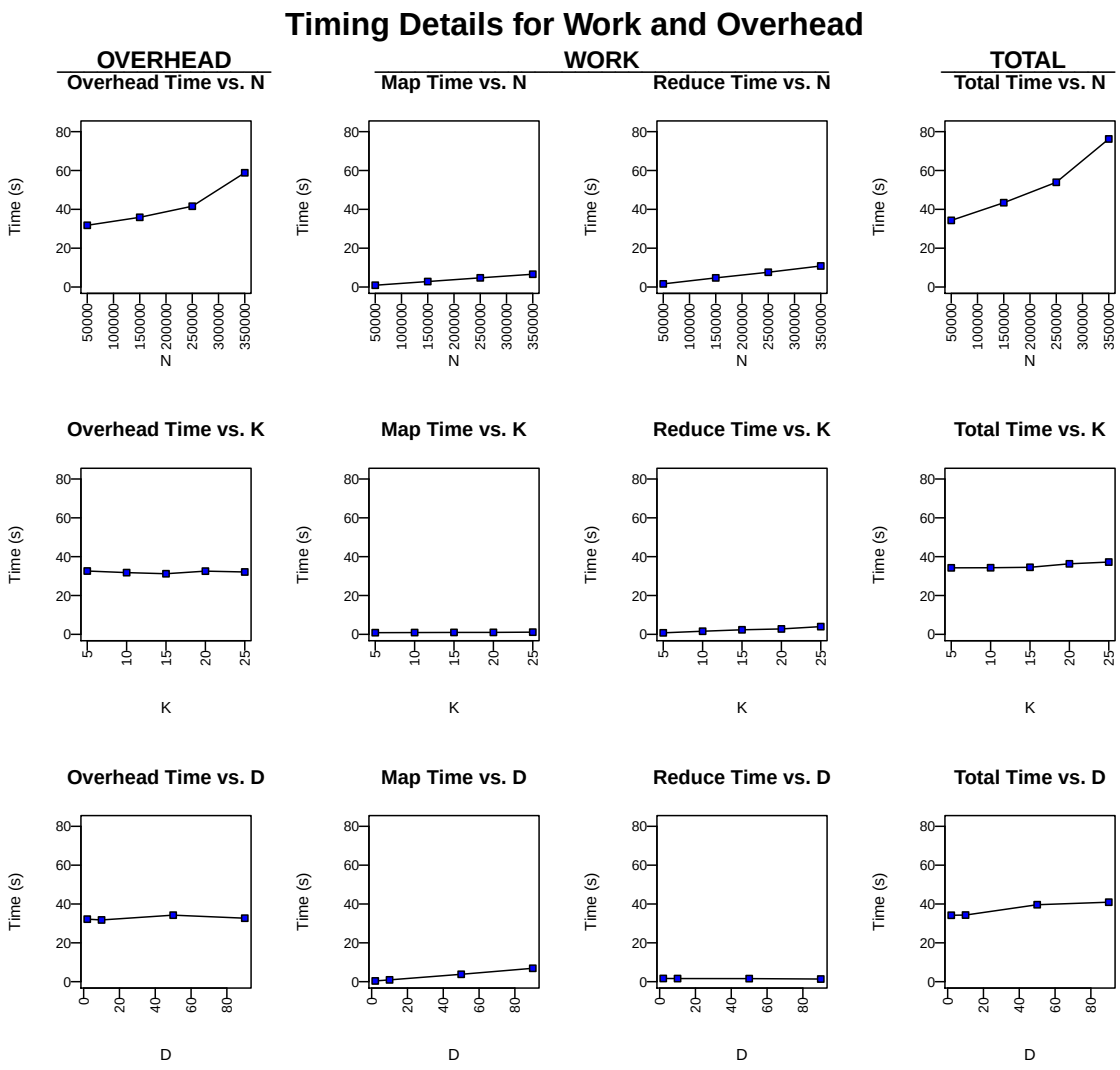

Figure A.18: Runtimes for a single iteration of EM broken into overhead (left column), map and reduce phase work (center columns), and total runtime (right column) as the number of samples N (top row), underlying source cluster count K (center row), and data dimensionality D (bottom row) are varied. This is the same experiment which produced the results in Chapter 4 repeated on smaller datasets.

REFERENCES

[1] Map/reduce tutorial, 2010. "`http://hadoop.apache.org/common/docs/current/mapred_tutorial.html`".

[2] Cloudera hadoop training: Mapreduce and hdfs, 2009. "`http://vimeo.com/3584536`".

[3] Hdfs architecture, 2010. "`http://hadoop.apache.org/common/docs/current/hdfs_design.html`".

[4] Zhi-Dan Zhao and Ming sheng Shang. User-based collaborative-filtering recommendation algorithms on hadoop. *International Workshop on Knowledge Discovery and Data Mining*, 0:478–481, 2010.

[5] Richard A. Brown. Hadoop at home: large-scale computing at a small college. *SIGCSE Bull.*, 41:106–110, March 2009.

[6] Chen Zhang, Hans De Sterck, Ashraf Aboulnaga, Haig Djambazian, and Rob Sladek. Case study of scientific data processing on a cloud using hadoop. In Douglas Mewhort, Natalie Cann, Gary Slater, and Thomas Naughton, editors, *High Performance Computing Systems and Applications*, volume 5976 of *Lecture Notes in Computer Science*, pages 400–415. Springer Berlin / Heidelberg, 2010.

[7] 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.

[8] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[9] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[10] S. Pallickara, J. Ekanayake, and G. Fox. An overview of the granules runtime for cloud computing. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 412 –413, dec. 2008.

[11] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] A P Dempster, N M Laird, and D B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society Series: B (Methodological)*, 39(1):1–38, 1977.

[14] C F Jeff Wu. On the convergence properties of the em algorithm. *The Annals of Statistics*, 11(1):95–103, 1983.

[15] Rolf Sundberg. *Maximum likelihood theory and applications for distributions generated when observing a function of an exponential family variable*. PhD dissertation, Stockholm University, 1971.

[16] Rolf Sundberg. Maximum likelihood theory for incomplete data from an exponential family. *Scandinavian Journal of Statistics*, 1(2):49–58, 1974.

[17] Rolf Sundberg. An iterative method for solution of the likelihood equations for incomplete data from exponential families. *Communications in Statistics - Simulation and Computation*, 5(1):55–64, 1976.

[18] Jeff Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, ICSI, 1997.

[19] Bruce A. Draper, Daniel L. Elliott, Jeremy Hayes, and Kyungim Baek. Em in high-dimensional spaces. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 35(3):571–577, 2005.

[20] Bill Venners. Designing distributed systems, a conversation with ken arnold, part iii, 2002. "`http://www.artima.com/intv/distrib.html`".

[21] Introduction to distributed system design, 2010. "`http://code.google.com/edu/parallel/dsd-tutorial.html`".

[22] Ken Birman. *Reliable Distributed Systems Technologies, Web Services, and Applications*. Springer, 2006.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[25] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *J. Parallel Distrib. Comput.*, 22:379–391, September 1994.

[26] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Concurrency*, 1:12–21, 1993.

[27] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29:610–611, June 1958.

[28] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. "`http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`".

[29] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and theATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000("`http://www.netlib.org/lapack/lawns/lawn147.ps`").

[30] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[31] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** URL: "`http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`".

[32] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`".

[33] See homepage for details. Atlas homepage. "`http://math-atlas.sourceforge.net/`".

[34] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[35] Douglas Turnbull and Charles Elkan. Fast recognition of musical genres using rbf networks. *IEEE Trans. on Knowl. and Data Eng.*, 17:580–584, April 2005.

[36] Lei Xu and Michael I. Jordan. On convergence properties of the em algorithm for gaussian mixtures. *Neural Computation*, 8:129–151, 1995.

[37] Martaza Jamshidian and Robert I. Jennrich. Acceleration of the em algorithm by using quasi-newton methods. *Journal of the Royal Statistical Society Series: B (Methodological)*, 59(3):569–587, 1997.