University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Masters in Computer Science	Spring semester, 2012 Open / Restricted access
Writer: Erick Chiliboyi	[illegible] (Author's signature)
Faculty supervisor: Prof. Chunming Rong External supervisor(s):	
Title of thesis: Investigating How Partial Aggregation Impacts K-means	
Credits (ECTS): 30	
Key words: Data Mining, Cluster Analysis, Parallel Algorithms	Pages: …49……… + enclosure: 2CDs…… Stavanger, 2012-06-14 Date/year

Investigating How Partial Aggregation Impacts K-means

June 14, 2012

Abstract

K-means is the most commonly known partitioning algorithm used for data clustering [5]. It was originally designed to run on a single processor. Therefore, this created a limitation on dealing with large amounts of data because of the requirement to have data resident in memory. However, the advent of distributed systems has led to the design of parallel versions of the algorithm. This is done with the intention to allow the algorithm to work with large sets of data that would otherwise be impossible to handle on a single machine, due to limited processing power and memory.

Since there are currently many applications that are generating large sets of data, for example in oil exploration, social media and image processing, research in parallel clustering algorithms has received a great deal of attention so as to find efficient ways of data analysis. Clustering algorithms that can be formulated according to the MapReduce framework such as K-means provide a great opportunity for data analysis. This is because the actual data is used to get statistics. Otherwise, statistics would only be obtained by applying sampling methods on the original data.

Running a parallel algorithm on a distributed platform to handle large data sets, requires alot of data transfer on the network. This exchange of data could choke the network if it is beyond the capacity of the network. Therefore, there is a way of reducing the amount of traffic on the network and that is by partial aggregation. Reducing traffic on the network has been known to increase efficiency [13].

In this thesis, we look at two things. First, how partial aggregation impacts the K-means algorithm running on a distributed system. Second, we compare two implementations of the K-means algorithm, Java and R.

Acknowledgements

First, I would like to thank my supervisor Prof. Chunming Rong for his guidance during the period I was working on my master thesis. He provided an enabling environment for research which included a well furnished lab and a cluster of computers. He was also available to review progress of my work at the regular and well organised meetings we used to have on weekly basis.

Besides, I would like to thank my co-supervisor Rui Paulo Maximo Pereira Mateus Esteves for his valuable suggestions and availability for discussion concerning my work.

My thanks also go to my labmates for the lovely time we shared together in the lab and the friendship we developed.

Lastly, my mother has been a solid rock of my life, she is my greatest inspiration, and to her am gratefully thankful. My brothers and friends have been a great part of my life and to them I say thank you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Clustering algorithms have a wide application in different fields of science. For example, they are used for machine learning and data analysis. One such algorithm is the K-means. This algorithm is simple yet powerful and practically efficient. It was first proposed as a serial algorithm that is to run on a single processor. This created a limitation on how much data it could be applied because the data had to be resident in memory. However, with great magnitudes of data being generated by various applications the algorithm had to be adjusted and a parallel version was proposed. The parallel version was adapted to run on distributed systems such as Hadoop that make use of commodity computers.

In a distributed system, the computers are connected by a network and so the capacity of the network, and delays associated with it, becomes a factor. Some of the cases that arise from the network include situations where the data being worked on is larger than the capacity of the network or individual pieces of data that have to be tranferred between computers become too many. In both cases the computer making the final computation would spend considerable time on communication more than computation. However, the solution to these cases is to do what is called *partial aggregation* [13]. It is an approach that is acknowledged to increase efficiency of parallel algorithms. We will therefore explore how *partial aggregation* affects the K-means algorithm.

Hadoop [7] is a software framework for distributed computing for han-

dling large data sets and it is based on MapReduce. The MapReduce [1] approach first maps values to keys in the map phase and outputs $< key, value >$ pairs that are input to the reduce phase that reduces groups of values with similar keys.
The K-means algorithm naturally fits into the MapReduce model because its most intensive part, the distance calculations, can be carried out independently.

1.1 K-means Algorithm

The K-means algorithm's application would be identified in the category of unsupervised learning. The training or input data is not labelled and so the algorithm partitions data into K clusters or groups based on the similarity that exists within the data. The K is predetermined by the user. Besides, the algorithm works by repeatedly calculating a number of K means or averages, hence the name of the algorithm, from the training data. The initial means are input to the algorithm by e.g randomly selecting K elements from the training data. In every iteration the traininig data is assigned to the closest mean called centroid. Closeness is determined by calculating the distance between every element in the training data and all the current centroids. After the training data is partitioned into K groups, a new centroid is calculated per group. The centroid is calculated by determining the mean of the elements in a cluster. Therefore, the new centroids are used in the next iteration to partition the training data in order to form new clusters. When the algorithm converges to some objective function, or if the means stop changing, it stops. At this point, the training data is partitioned into the final clusters. However, different initial centres will produce clusters with different contents therefore the algorithm does not produce unique clusters if the initial centres are changed. Therefore, the algorithm just works to group the data that is more similar together.

1.1.1 Serial Approach

The K-means algorithm was originally designed to run serially on a single processor. This had an implication that the data had to be resident in memory in order to be clustered. The requirement to have the data present in memory is in itself a bottleneck because when the data is too large the algorithm fails to work. The serial approach would further be lendered impractical in the face of large data sets that would need to be clustered. Currently there are applications that produce large datasets that are in magnitudes beyond the capacity of memory on a single machine hence making this approach of data clustering not an option to consider. Besides, a single machine may not suffice to provide the processing power necessary.

1.1.2 Message Passing Interface[MPI] Approach

The MPI standard allows applications to take advantage of distributed memory on the network. Therefore, with this approach the K-means algorithm can work with larger data sets because the data is divided among the nodes in the network cluster. Furthermore, this approach also takes advantage of the multicore processors and therefore the computation time should be expected to be reduced by a factor equal the total number of processor cores. However, the collective memory capacity of the nodes is still a limiting factor.

1.1.3 MapReduce Approach

MapReduce is a programming model and implementation that is originally found in functional languages, e.g Common Lisp and Scheme though it has now been popularised by Google[1]. The software framework and implementation based on this concept is the Hadoop MapReduce[3] that provides the distributed environment for parallel programs. The approach offered by MapReduce would allow K-means to cluster very large datasets of high magnitudes because the data does not need to be resident in memory. Therefore, this approach somehow waives the memory limitation evident in the previous approaches because the training data is kept on local disks that provide cheaper and higher storage capacity.

The MapReduce approach requires first that a user defines a map function that assigns keys to values which then outputs $< key, value >$ pairs as intermediate values. Another function that a user defines is a reduce function that takes the intermediate values from the map function and merges the values with similar keys. The output of the reduce function contains unique $< key, value >$ pairs as final group computations. However, an optional function called a combiner can be defined by the user, in some cases similar to the reduce function, to do some computations on the local computer before any data transfer can take place. It is the combiner function that performs *partial aggregation*.

1.2 Partial Aggregation

In the K-means algorithm, the new centroids are calculated by determining the means which involves mainly aggregation. Training data belonging to the same group maybe residing on different machines within the distributed system. Therefore, partial sums can be done on the local machine per group, identified by similar keys, and then the partial sums along side the count of the summed up data can be sent to the machine calculating the centroids since the centroids have to be global. Hence, every machine in the distributed system can send these values in the form, as below:

```
struct PartialCountAndSum{
    int partialCountG
    int partialSumG
}
```

1.3 Hadoop

Apache Hadoop is a software framework and programming model for developing programs to be executed in parallel using MapReduce[1] on a distributed file system. It provides high availability, reliability and scalability. This makes it favourable for analysing large data sets.

1.3.1 Hadoop Distributed File System[HDFS]

HDFS[2] stores file contents across the datanodes [2] in the network cluster and it works as a file system component of Hadoop. The blocks of each file are replicated so that in an event that a node containing a particular file block becomes inaccessible, the computations should not stop. The storing of blocks of data across multiple nodes allows the file system to store larger files that would otherwise not be able to fit on any single node's local disk in the cluster.

HDFS separates metadata, a description of data, from data itself. It keeps the metadata on a node configured as namenode[2] and the data is itself stored on the nodes configured as datanodes[2]. So the data is basically replicated on several datanodes.

1.4 Motivation

Partial aggregation is known to have a postive effect on the performance of parallel algorithms running on distributed systems [17]. Therefore, the motivation of this work is to establish how *partial aggregation* impacts the K-means algorithm that is designed to run on a distributed system. There is a general acknowledgement that *partial aggregation* improves performance in distributed systems as observed from some work done on aggregation in distributed systems [13], and also most work related to algorithms running on distributed systems [17].

R [19] is a statistical programming language. Therefore, it allows easier expression of data analysis problems. However, the recent past has seen the development of packages that allow problems to be expressed in the MapReduce model from R. We would therefore like to compare the K-means implementations in Java and R. This is particularly because R like Java can connect to distributed systems such as Hadoop though using Streaming [21].

1.5 Goals

1. To evaluate how *partial aggregation* impacts the K-means algorithm running on a distributed system. This will help understand why it is necessary to use it in algorithms running on distributed systems.

2. To compare the Java and R implementations of the basic K-means algorithm on Hadoop. The Hadoop system is implemented in Java

and so the Java implementation is expected to run faster because it directly connects to Hadoop while R only connects using Streaming [21].

The above goals define the contribution presented in this thesis.

Chapter 2

Preliminaries

2.1 Data Presentation

In order to manipulate input data in a distributed system, a way to represent it must be sought. Therefore, data presentation is very critical to parallel clustering algorithms. The pattern matrix and proximity matrix described in [4] were used for data presentation in this work. [4] refers to a data point as a pattern or d-place vector where d is the number of attributes or dimensions. So both data points and patterns will be used interchangeably. Given a pattern **x**, it would be described by a vector as: $(x_1, x_2, x_3 ....x_d)$ and each x_h represents an attribute where h = 1,2,3...d. If there are **n** patterns to be represented in a pattern matrix, the pattern matrix to use would be a **n x d** matrix. The value of d can also be looked at as a dimension of the given data point. Therefore, each *i*th row of a *single column* pattern matrix represents a single attribute and every row with d columns represents a pattern, where i = 1,2,3...n.

In the K-means algorithm, the patterns are clustered based on the closeness to cluster centroids. How close a pattern is to a centroid is measured by a distance metric. Therefore, the closeness of patterns to the centroids can be represented by a proxity matrix. Let us take a set of centroids $C = \{C_1, C_2, C_3 ....C_p\}$ where p in the number of cluster centroids and C_j, j=1,2,3...p, is a single cluster centroid, we can generate **n x p** proximity matrix, with n being equal to the number of rows in the pattern matrix

described above. Each C_j can be described by a vector:$(c_1, c_2, c_3.......c_d)$ and therefore the dimension of C_j must be equal to the dimension of the entries of the corresponding pattern matrix.
Therefore, every entry d_{ij} in the proximity matrix, where i=1,2,3....n and j=1,2,3......p, is a distance measure between the ith row entry in the pattern matrix and the jth entry in C. Hence, the number of entries in the proximity matrix is equal to the number of distance computations.

2.2 MapReduce Based K-means Algorithm

Below is a general description of an algorithm that is based on MapReduce as described both by [6] and [5]. It consists of three phases namely *map*,*combine* and *reduce*.

2.2.1 Map Phase

This phase proceeds as below:

1. Calculate the distance between the K centroids and the N input patterns.
2. Based on the calculated distances from step 1, each pattern will be assigned to the nearest centroid.
3. The output of this phase will be pairs of $< centroidID, pattern >$

2.2.2 Combine Phase

In this phase, the partial sums are calculated and an associated counter for each group updated.The input is $< centroidID, pattern >$ pairs from the map phase. The output is pairs of $< centroidID, R >$ where R is a datastructure containing a partial sum and the count of summed up patterns. The phase proceeds as below:

1. Add up pairs according to their corresponding centroid identifiers.

2. Update the counters and partial sums in the R datastructure for each unique centroid identifier.
3. The output is $< centroidID, R >$ pairs.

2.2.3 Reduce Phase

In this phase, the means are calculated per centroid identifier. The input is $< centroidID, R >$ pairs from the combine phase.

1. Add the partial sums per centroid identifier.
2. Calculate the means per centroid identifier using the respective counters.
3. The output is $< centroidID, mean >$ K pairs. Where K is the number of centroids.

2.3 Mean and Distance Calculations

The two calculations involved in the K-Means are; distance between patterns and centroids, and the means for each group. Let X_i represent the patterns in the ith cluster, where i=1,2,3...K and K is the number of clusters. Then let n_i be the number of data points in the ith group. Therefore, to find the mean, m_i, for a group whose data points are represented by X_i, we use the equation as below:

$$\frac{1}{n_i}\sum_{a=1}^{n_i} x_a^i \tag{2.1}$$

Thus the above equation can allow us to calculate the required means and in this case the number of means is K.

However, the calculation of distance would be done according to the equation given below. The distance measure being considered here is euclidean. Otherwise, a different formula would be applicable.
Given that n is the total number of data points to be clustered into K groups. The distance, d, between each data point, x, and a centroid m_i would be

calculated as follows:

$$\sqrt{\sum_{j=1}^{n}(m_i - x_j)^2} \tag{2.2}$$

The above formula can therefore be used to calculate the distance between each single data point and all the K means before the data point is clustered to its closest mean, centroid.

2.4 Decomposition and Independence

Decomposition allows for intractable computational problems to be broken down into smaller and more tractable problems. The solutions to smaller problems can however be merged so that collectively they can form a solution to the original problem. This approach to complex computational problems traces its roots in the principle of divide and conquer. However, for any given problem to be divided into subproblems, the condition that must be fulfilled is that the resultant subproblems have to be independent. This is because the complete solution to the problem must come from the individual subproblems.

The MapReduce model takes on problems that can be decomposed and also whose smaller divisions exhibit independence.To take a comparative example we can look at summation and the fibonacci series. When summing up numbers, the order in which that is done does not matter and therefore summing up many numbers can be divided into groups whose sums can be put together to come up with a final sum. But when we consider the case of fibonacci series it obvious that decomposition cannot work because every *n*th term is dependent on the *n-1*th term. Consequently, fibonacci cannot be implemented in the MapReduce model though K-means can be.

2.4.1 Formalization

The most intensive part of the K-Means algorithm is the calculation of distances. Therefore, distributing these calculations across machines is very necessary to handling large datasets.

We can define distance calculation C by using the triple: C = ($\mathbb{R}^d$, $\mathbb{R}$, sodp

) where $\mathbb{R}^d$ denotes the euclidean space of dimension d and sodp: $\mathbb{R}^d \rightarrow \mathbb{R}$, is a function. Given a , $b \in \mathbb{R}^d$ we can calculate distance *dis* as *dis* = sodp(a , b). However, the computation, *dis'* = sodp(*a'*, *b'*) where a', b' $\in \mathbb{R}^d$ can take place simultaneously because it is independent of the preceeding calculation. Therefore, when the centroids are global, the distance calculations can be parallelised.

In addition to distance calculations, the means of the algorithm also have to be calculated iteratively. The means are calculated by collecting all data points belonging to the same cluster and then finding their mean on group basis. This implies sending data points across the network to the respective reducers that calculate the group means. However, network communication cost can be minimised by doing local sums of the points and then sending across the network only the partial sums and their associated counters, recording the data points added up per group. Therefore, in this case the reducer has to add up the values of the counters and partial sums and then determine the mean per group basis. The partial sums are representing the combine phase of the algorithm described above. The combine phase is realised because of both the associative and commutative properties of addition. These properties demonstrate independence when applying addition to any given terms. Therefore, independence is very important to realising parallelism because it allows for decomposition of complex problems.
We can now look at the two important properties in mathematics that we have just alluded to. Given x, y, z $\in \mathbb{R}$, associativity can be described as; (x + y) + z = x + (y + z) and commutative for two variables as; x + y = y + x or z + y = y + z or x + z = z + x.

2.4.2 Discussion

The above section has shown that the properties of addition, that is associative and commutative, are very important to the parallelising of the computations across machines. Furthermore, the properties allow for partial summations to be executed on the local machines thereby reducing on network traffic. Consequently, the time taken for computation exceeds the communication time and this makes the running time for the algorithm better.

We have seen how two distance computations can be carried out in parallel since they are independent in nature. The distribution of distance computations is important because this the most intensive part of the algorithm. Therefore, in general the nature of the K-means algorithm allows it to fit into the MapReduce model.

Chapter 3

Literature Review

3.1 Introduction

The K-means algorithm naturally fits into the MapReduce framework, and has therefore attracted a lot of research interest. This is because of the need to make it run in parallel or on distributed systems. This is, in part, because of the wide appeal of the K-means algorithm to different fields of science. However, this is also because of the huge amounts of data that cannot be comprehensively analysed on a single machine without resorting to sampling methods. Sampling methods allow only sampled population of data to be analysed. However, distributed systems are capable of addressing this limitation. The algorithm is used in document clustering, data analysis and classification of data among many other applications handling large amounts of data. In this chapter we make a review of various aspects involved in making the most of this simple yet powerful algorithm. We will get some perspectives on different approaches to distributed computing and other related issues. The choice of which approach to use hinges on the amount of data being considered since the advantages of some architectures are only realised when there are huge margins of data involved.

3.2 Different Approaches To Parallelizing K-means

Message Passing Interface(MPI) is a specifications standard that makes it possible for processes running on different processors to commmunicate using the message passing model. The processors maybe either be connected on the same board or by a network on different machines. There are K-means algorithm implementations that are based on MPI and they therefore take advantage of main memory accessible to the processors. Such algorithms are bound to be faster because the data is stored in memory compared to those that have to access data stored on disk.

MPI was used by [9] to run a parallel K-means algorithm. They evaluated both scaleup and speedup. In the case of scaleup, they demonstrated that it was constant with respect to the number of dimensions of the data points, the number of centroids and the number of data points. For speedup, it was observed that as the number of data points increased there was a corresponding increase. This also implied a good sizeup. Furthermore, [9] showed that the time required to synchronise data points and calculate new centroids is inversely proportional to both speedup and scaleup. They also described a linear relationship between communication cost and the process of calculating new centroids per interation.

Another MPI implementation is demonstrated in [10] where they compare the serial and parallel versions of the algorithm. They observed that with 100,000 - 600,000 number of data points, there was no speedup recorded. This was attributed to the fact that the communication time was more than the computation time. However, between 700,000 - 900,000 number of data points speedup was recorded because of the computation time being greater than the communication time. Their parallel algorithm was not scalable because the number of machines in the cluster defined the possible number of centroids that were considered. There was also another limitation of not using *partial aggregation* because they reported it could further hamper scalability.

CUDA architecture [12] allows the application developers to write distributed applications that can run on GPUs. This approach was used by [11] to implement the K-means algorithm and they recorded fast computation times.

MapReduce [1] is another programming model that distributes computations across commodity machines in a cluster. It is most favourable for applications meant for handling very large data sets that cannot fit in memory.

This model is simple yet very powerful. [5] describe a K-means algorithm based on MapReduce. They concluded that the algorithm was scalable and also that it demonstrated speedup. The implementation was carried out on Hadoop and they made use of a combiner to introduce *partial aggregation*.

FREERIDE [17] is another framework that is also used for writing data mining applications. It takes advantage of both the shared and distributed memory archictures. It provides the programmers with an option to reduce network traffic by doing local reductions or *partial aggregation* of data before the global reduction which involves movement of data between machines can be executed. This framework is viable for both grid and cluster computing. Scalability was demonstrated in this framework as well as speedup.

Phoenix [18] implements MapReduce in a way that takes advantage of the available shared memory in both multicore and multiprocessor architecures. It is another platform that allows programmers to write parallel applications.

3.3 Partial Aggregation In Parallel Computing

Distributed computing is very suitable for data intensive applications, which are very common today and examples being applications used in oil exploration and social networks. This is because it allows the load of computing to be shared among many machines. However, most setups of distributed environments have machines connected by a network. This brings another concern as how to handle large amounts of data being transferred between machines. If the amount of data crossing the network is too much, it will have an impact on the computing time. In the case where the communication time of data is more than the computing time it is difficult to realise the advantage of parallel computing. To deal with this challenge, there is an option to do local aggregation at different levels ranging from machine to rack to cluster level. This consequently helps reduce network traffic.

The MapReduce model [1] has been used widely in analysing large data sets because it allows for parallelizing of computations. There is also work that has been done to address aggregation in this model for example in [13]. *Partial aggregation* has been used in [13] to describe the data reduction at various local levels, at rack and computer, however the conditions for such are also well outlined and formalised. In MapReduce [1] a user defined function, called a combiner, can be used to perform such data reduction. The I/O reduction, which is an optimisation strategy that ensures that I/O requests are minimised, is based on data reduction, here referred to as *partial aggregation*, for example as used in systems like DryadLINQ [14]. Further, [13] suggests that aggregation can be expressed in parallel database systems that support MapReduce. Besides, the iterator and accumulator based interfaces are also discussed as an approach to aggregation in distributed systems.

The middleware in [17] was designed to help programmers write applications for data mining. The data mining algorithms, of which K-means is, have a common general structure and therefore [17] describes a framework that allows developers to do local reduction [17], which is just another terminology used for partial aggregation, in order to minimise network traffic.

3.4 Communication In Distributed Computing

Communication between elements of the distributed systems is very central. However, communication happens at different levels which can be broadly categorised as intra-node and inter-node communication. These two categories are addressed differently depending on the architecture being considered. In architectures like CUDA, the intra-node communication is of more importance but in other frameworks both are very important. In MPI, for instance, it is very important that both levels of communication are optimal because the processes are spread across processors on the same node as well as processors across different nodes. This is necessary for the synchronisation of activities as well as data.

The Hadoop[7] project has a distributed file system called Hadoop Distributed File System(HDFS)[2]. This file system manages data in a distributed environment. In a distributed system, there must be fault tolerance because failures will definitely occur as some machines may become unavailable due to network problems or the local disk on the machine may crash. HDFS addresses such problems by replicating data on different machines so that if one machine is inaccessible the computations will still continue on a different machine with the same data stored on other machines. In order to achieve such fault tolerance, there must be constant communication between machines and also availability of information about where each replica of the data is stored. In the HDFS architecture there are two main elements namely the namenode and the datanode. The namenode maintains information about the datanodes containing respective replicas. This is necessary in the face of failure so that the system remains functional. The data replicas are stored on the datanodes. Each datanode has to perform a handshake with the namenode, to prove its credence, upon startup so as to ensure integrity of the data in the cluster. When the datanode starts running, it will periodically send messages, called heartbeats, to the namenode to signal its availability. In the absense of the heartbeats after a specific period, the namenode determines that the respective datanode is non-functional. Therefore, when the datanode is rendered to be non-functional, the namenode reschedules the jobs that were assigned to that particular datanode and assigns them to the datanodes storing the same data stored on the failed node. The namenode is the focal point of the HDFS architecture because it has to co-ordinate the operations of the datanodes in a cluster. However, this happens to be the single point of failure.

Processor speed is very important to executing tasks very fast and moreso in multiprocessor systems when handling distributed tasks. However, how fast the processors communicate in multiprocessor systems with each other is very critical to efficiently coordinate activities. With respect to this, [15] looks at the MPI intra-node communication with particular emphasis on using the user space memory copy scheme though the drawback of this approach is that it could make the processor achieve less computation compared to the NIC-based loopback approach. However, the NIC-loopback based approach cannot distinguish intra-node traffic. The only way it can tell between the inter-node, destined for another node, and intra-node traffic is by looking at the source and destination processes. In their work [16] evaluated different mechanisms of intra-node communication. They used several metrics, among them were latency and bandwidth, to determine efficiency. However, they suggested that in the NIC-loopback based approach bandwidth and latency is a major drawback because the data being transferred has to traverse the I/O bus before it can get to the destination process. The I/O bus is much slower than the system bus.

In Phoenix [18] the buffers in shared-memory are used as a means of communication between processes running on the different processors and/or cores.

Chapter 4

Experimental Evaluation

4.1 Introduction

In this chapter we will look at the results of the experiments that were conducted and then make an analysis. The servers were configured in sets of 3, 5 and 7 on the cluster on which the K-means algorithm was run. The single processor speed was 2.6GHz and of type AMD Opteron 6-Core 4180. Therefore, each server had six processors and each processor had six cores. The servers were running Centos OS and each had 32 GB memory.

For the purpose of the experiments undertaken, the data sets used were integers with two dimensions. The number of data points in the data sets used were 200,000, 400,000, 600,000, 800,000 and 1,000,000. Each data set was run on the three different cluster configurations mentioned above. However, with 1,000,000 data points the algorithm was taking too long to run without the combiner and so the time could not be recorded.

The time taken by the K-means algorithm was recorded in minutes and each value recorded was an average of three repeated trials for a given data set on a particular cluster configuration. The number of iterations for all the experiments was 10, with four centres or clusters (K=4), and none of the data sets used converged within those iterations.

The program used for the experiments was implemented in the R [19] statistical programming language. In addition, Hadoop was used as the framework for distributing the computations on the cluster. However, R needs a programming framwork to connect to the Hadoop framework. In the experiments, the RHadoop [20] programming framework was used to provide the needed connectivity. Its component packages allow access to the MapReduce and the distributed file system of Hadoop. Eventually, the R implementation is compared to the K-means algorithm implemented in Java from the Mahout framework.

4.2 Results

In order to describe how fast the algorithms were running two terms were introduced. The terms were used to compare running times when the number of servers in the cluster was adjusted and when the combiner in the algorithm was introduced. The two terms were Server Speed Ratio(SSR) and Combine Speed Ratio(CSR), and are defined as below.
SSR: This is a factor that indicates how fast the algorithm runs for a given data set on different cluster configurations and particularly when the number of servers is increased. It is a ratio of time taken by the algorithm to run on a given cluster configuration to the time taken on the base cluster configuration, in this case 3 servers, for the same data set.
CSR: This is a factor that indicates how fast the algorithm runs for a given data set on a particular cluster configuration with respect to the presence or absence of the combiner. It is a ratio of time taken by the algorithm when using a combiner to the time taken when the combiner is not used.
The above ratios are as expressed as below: Given a data set N. Let a = *Time taken to cluster N without a combiner* and b = *Time taken to cluster N using a combiner*,
then:

$$CSR = \frac{a}{b}$$

Now let d = *Time taken to cluster N with/without a combiner on 3 servers* and c = *Time taken to cluster N on x servers*, where

$$x \geq 5$$

then:

$$SSR = \frac{c}{d}$$

In order to compare running times between the R and Java implementations, a ratio called Speed Ratio(SR) was introduced. It is expressed as below. Given a data set N and that; a = *Time taken to cluster N using R implementation* and b = *Time taken to cluster N using Java implementation*,
then:

$$SR = \frac{a}{b}$$

The experimental results are presented in two sets. The first set of results was obtained from the algorithm implemented in R, as in tables 4.1 4.2 4.3 4.4 4.5, and the second set was obtained from an algorithm implemented in Java which is part of the Mahout framework, tables 4.6 4.7 4.8 4.9 4.10. In addition, tables 4.11 4.12 4.13 show the comparison, defined as a ratio, of running times between the Mahout based algorithm and the R implementation that was using a combiner.

N=	With Combiner[min]	No Combiner[min]	Combine Speed Ratio
200,000	16.93438	30.43843	1.797433978
400,000	21.2594	67.8969	3.193735477
600,000	25.40931333	144.78928	5.698275986
800,000	29.82498	200.13228	6.710223444
1,000,000	34.28722	Nil	Nil

Table 4.1: Time and CSR on Three Servers

N=	With Combiner[min]	No Combiner[min]	Combine Speed Ratio
200,000	11.07024667	25.63153333	2.315353407
400,000	13.58591333	60.74878	4.47145352
600,000	16.47119667	121.06808	7.350290477
800,000	19.10741333	194.05322	10.15591261
1,000,000	23.51826333	Nil	Nil

Table 4.2: Time and CSR on Five Servers

N=	With Combiner[min]	No Combiner[min]	Combine Speed Ratio
200,000	9.210451667	23.77025333	2.580791279
400,000	10.70146333	57.86763333	5.407450508
600,000	12.86491	116.27462	9.038121526
800,000	14.57861667	185.02874	12.69178992
1,000,000	18.40605333	Nil	Nil

Table 4.3: Time and CSR on Seven Servers

N=	No Combiner	With Combiner
200,000	1.187538397	1.529720205
400,000	1.11766689	1.564811984
600,000	1.195932735	1.542651323
800,000	1.031326767	1.560911437
1,000,000	Nil	1.457897614

Table 4.4: SSR values for Five Servers

N=	No Combiner	With Combiner
200,000	1.280526109	1.83860473
400,000	1.173313925	1.986588127
600,000	1.245235461	1.975086754
800,000	1.081628076	2.045803157
1,000,000	Nil	1.862823028

Table 4.5: SSR values for Seven Servers

N=	Time[min]
200,000	4.900694444
400,000	5.149455556
600,000	5.3558
800,000	5.697561111

Table 4.6: Time on Three Servers[Mahout]

N=	Time[min]
200,000	4.666377778
400,000	4.84155
600,000	5.159672222
800,000	5.426011111

Table 4.7: Time on Five Servers[Mahout]

N=	Time[min]
200,000	3.335461111
400,000	3.530977778
600,000	3.829127778
800,000	4.129677778

Table 4.8: Time on Seven Servers[Mahout]

N=	Server Speed Ratio
200,000	1.050213823
400,000	1.063596484
600,000	1.038011674
800,000	1.050045972

Table 4.9: SSR values on Five Servers[Mahout]

N=	Server Speed Ratio
200,000	1.469270449
400,000	1.458365325
600,000	1.398699733
800,000	1.37966239

Table 4.10: SSR values on Seven Servers[Mahout]

N=	R With Combiner[min]	Mahout[min]	Speed Ratio
200,000	16.93438	4.900694444	3.455506192
400,000	21.2594	5.149455556	4.128475286
600,000	25.40931333	5.3558	4.74426105
800,000	29.82498	5.697561111	5.234692427

Table 4.11: Time and SR on Three Servers

N=	R With Combiner[min]	Mahout[min]	Ratio
200,000	11.07024667	4.666377778	2.372342574
400,000	13.58591333	4.84155	2.806108237
600,000	16.47119667	5.159672222	3.192295161
800,000	19.10741333	5.426011111	3.521447513

Table 4.12: Time and SR on Five Servers

N=	R With Combiner[min]	Mahout[min]	Ratio
200,000	9.210451667	3.335461111	2.761372824
400,000	10.70146333	3.530977778	3.030736529
600,000	12.86491	3.829127778	3.359749464
800,000	14.57861667	4.129677778	3.530206823

Table 4.13: Time and SR on Seven Servers

4.3 Analysis

In this section, we will delve into looking at the relationships observed from the data collected in the tables 4.1 to 4.10. The data was plotted in the graphs shown in the figures 4.1 4.2 4.3 4.4 4.5 4.6 4.7.

Figure 4.1 shows a plot between data points and the *Server Speed Ratio*. When the combiner was used in the algorithm, the *Server Speed Ratio* was fairly constant and otherwise the ratio was decreasing as the number of data points increased. The former behaviour was observed even in the Mahout algorithm that also uses a combiner as observed in figure 4.2. Therefore, when a combiner is used it can be suggested that regardless of the number of data points under consideration the running time of the algorithm is changed by a constant when the number of computing elements is adjusted. However, in the absence of a combiner as the number of data points increased the *Server Speed Ratio* dropped because the communication time between computing elements increased [9].

In figure 4.3 is a plot between the *Combine Speed Ratio* and data points. The graph reveals that the introduction of a combiner in the algorithm significantly reduces the running time and that this effect is linearly proportional to the number of data points. As already defined, *Combine Speed Ratio* is simply a ratio of the running time taken with a combiner to when the combiner is absent, for a particular number of data points, by the algorithm. However, this graph could only be done for the R implemented algorithm because there was an option to run the algorithm with and without a combiner while the Mahout algorithm could only run with a combiner in place.

When the algorithm was run without a combiner, the running times obtained were plotted against the data points and the figure 4.4 depicts the graph obtained. There seems much more like a linear relationship between time and the number of data points. However, in figure 4.5 the running times of the algorithm, with a combiner used, were plotted against the number of data points. By observation, it could be seen that the graph in figure 4.4 had lines that were steeper than the graph in figure 4.5. The lines in the graph in figure 4.5 were more gentle and it was observed that as the number of processing elements [servers] increased the lines became gentler. The contrast between the figures 4.4 and 4.5 was well demonstrated in the plot of *Combine Speed Ratio* and data points as shown in figure 4.3. This is because *Combine Speed Ratio* simply compares the running times in figures 4.4 and 4.5. Besides, 4.3 demonstrates that CSP is linearly proportional to data points. When the combiner is absent the algorithm does not gain much in speed as is the case when the combiner is present. The difference between running times increases as data points increase.

The figure 4.6 shows a plot using the Mahout algorithm similar to figure 4.5, graph of time against data points. The main difference is that the R based algorithm depicted in 4.5 has steeper lines than those in 4.6. In figure 4.5, as the number of processing elements, i.e. servers, increased there was a significant reduction in running time hence the corresponding reduction in the steepness of the lines. However, the reduction in steepness in 4.6 was not so much as the number of servers increased because the change in running time was relatively smaller. However, the ratios between the running times in figure 4.5 to the running times in figure 4.6 were plotted against data points as in the graph of figure 4.7. The relationship established in figure 4.7 was linear.

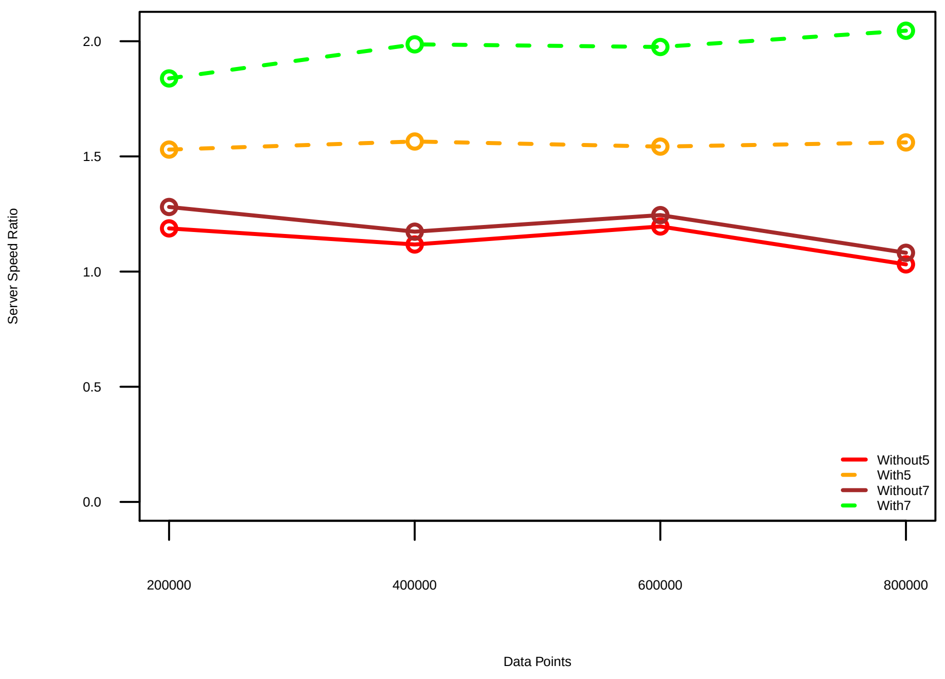

Figure 4.1: SSR for Five and Seven Servers. WithX means using a combiner and WithoutX means the combiner is not used on X servers.

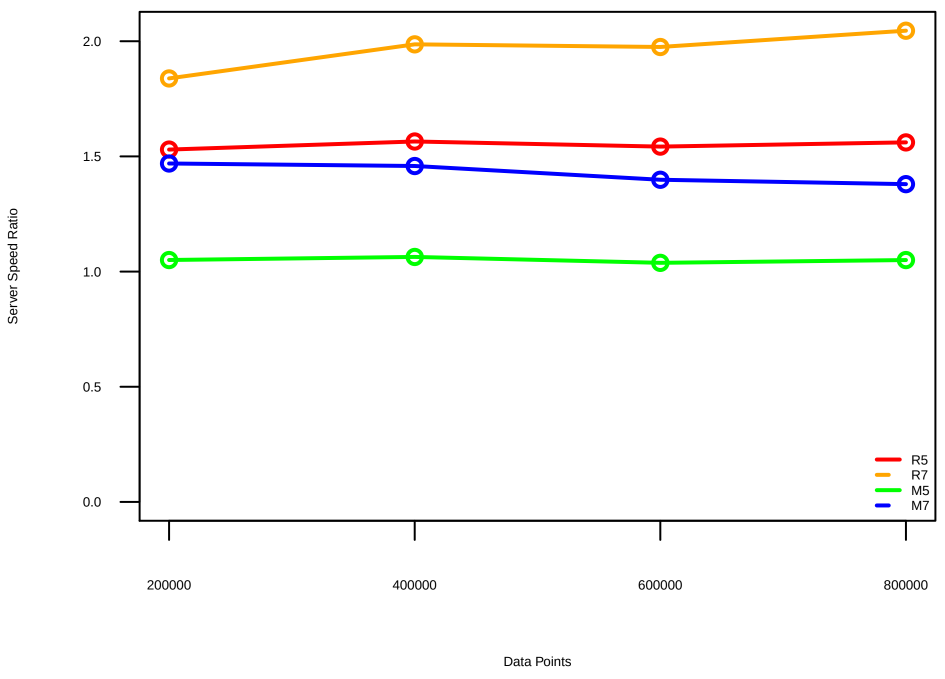

Figure 4.2: SSR for Five and Seven Servers using the combiner. RX repesents R implementation and MX represents Mahout implementation and X is the number of servers.

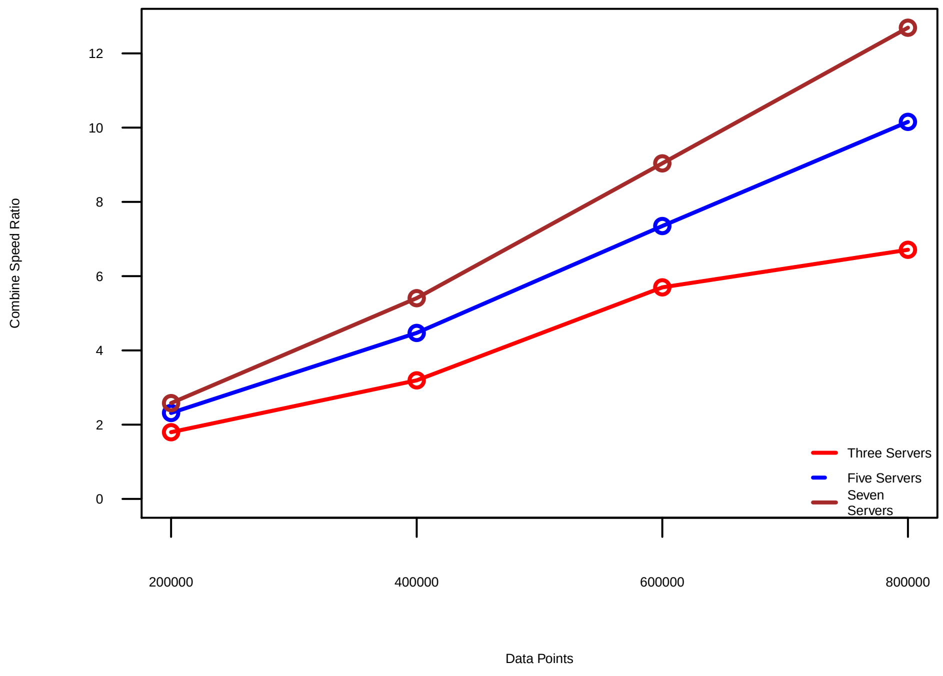

Figure 4.3: CSR on Three, Five and Seven Servers. This is for R implementation.

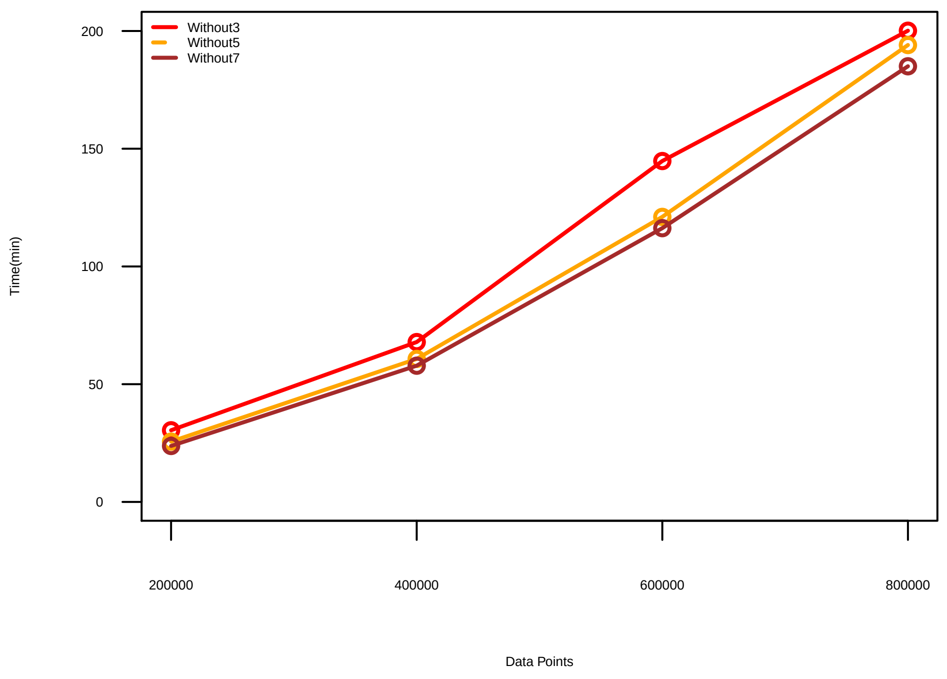

Figure 4.4: Time for R implementation without a combiner . The X in WithoutX is the number of servers.

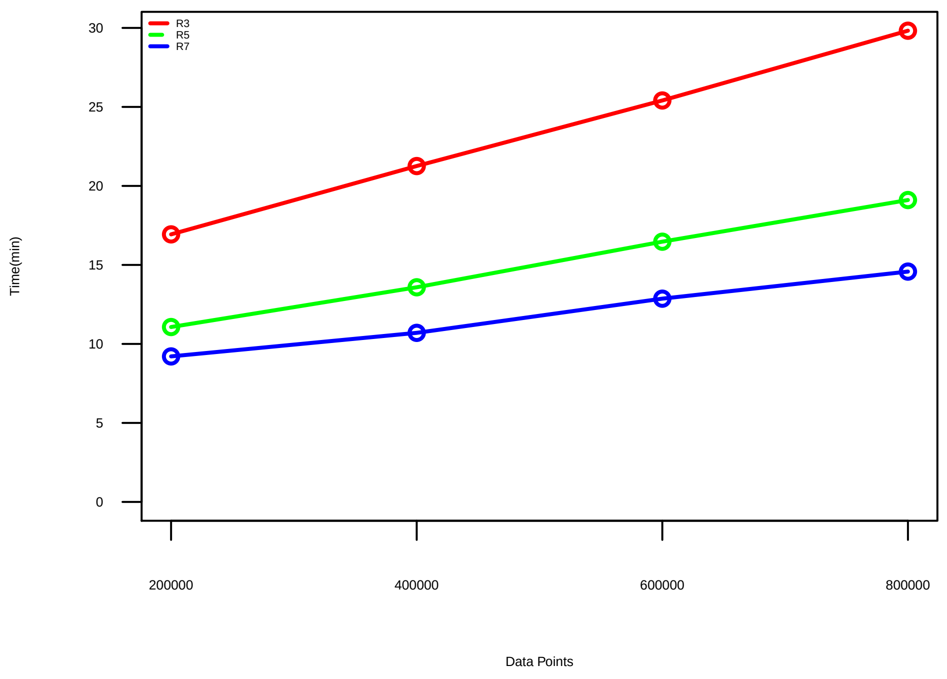

Figure 4.5: Time for R implementation using a combiner. The X in RX is the number of servers.

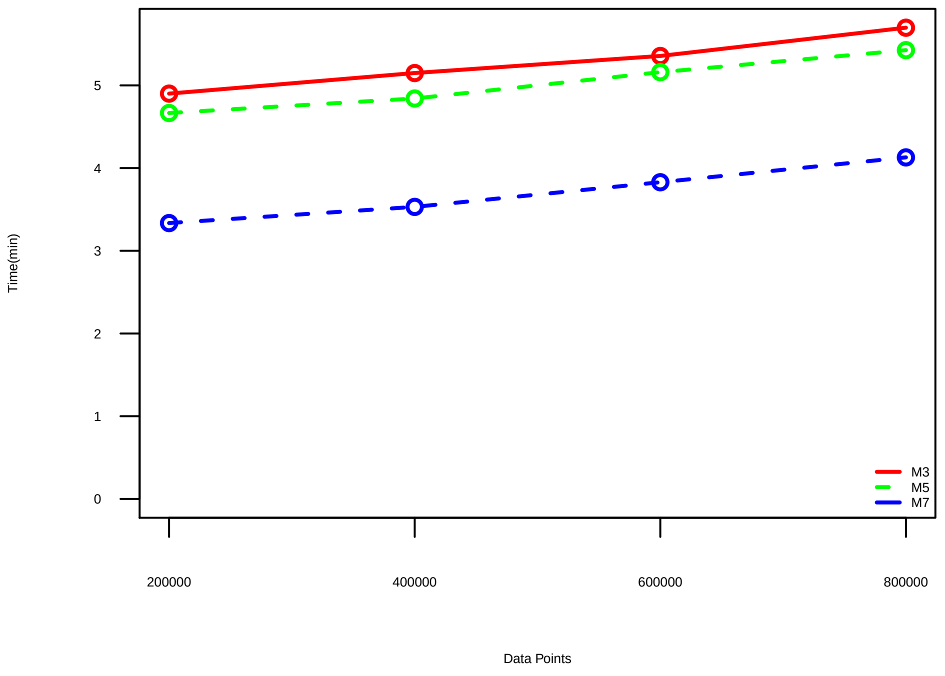

Figure 4.6: Time for Mahout implementation using a combiner. The X in MX is the number of servers.

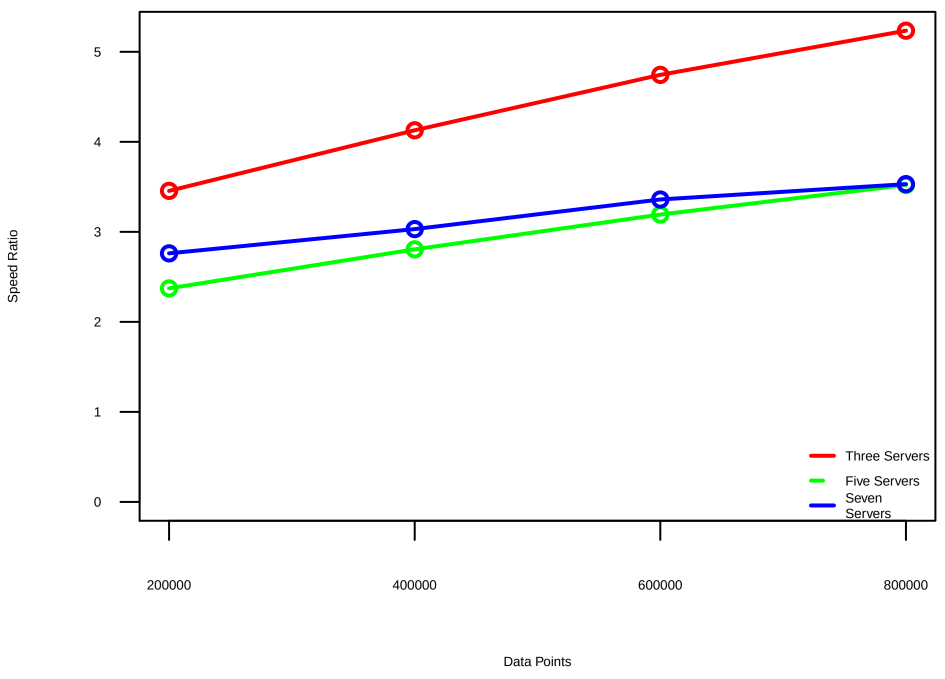

Figure 4.7: Graph between SR and data points.

4.4 Related Work

In most implementations of parallel algorithms there has been an acknowledgement of the positive impact of *partial aggregation* such as in [17]. However, the work presented here is very specific and focuses on the evaluation of how *partial aggregation* impacts K-means running on a distributed system. The other part of this work that draws a comparison between the R and Java implementations could not be related to any other works done so far as involving the distributed algorithms.

However, a relationship can be established between this work and other works with respect to *speedup* , *scaleup* and *sizeup* since mostly these are used to measure performance on distributed systems. The *parallel KMeans* based on MapReduce in [5] used a combiner which effects *partial aggregation.* The algorithm was reported to demonstrate the three qualities positively. There is also another parallel implementation of K-means using MPI as described in [9], which uses a function that does *partial aggregation* at machine level. They also demonstrated *speedup* , *scaleup* and *sizeup* in their implementation by varying dimensions and data set sizes. In addition, [10] show another MPI implementation of a parallel K-means algorithm. On the contrary, the algorithm never used *partial aggregation* and so it was limited as to how much input data it could be applied. The algorithm was reported to had had its scalability negatively affected if the *partial aggregation* had been used. The algorithm's number of cluster centres was tied to the number of servers.

In [14] *partial aggregation* is discussed in the context of DryadLink, which is a system for managing distributed programs, and how it affects the choice of *execution plans* [14] during runtime. Different approaches were employed to evaluate *partial aggregation* in the system and a comparison was made with equivalent other distributed systems such as MapReduce.
However, *local reduction* is another term used for *partial aggregation* in [17]. [17] describes a framework for distributed programming in which *partial aggregation*'s advantage is highlighted as having a great impact on reducing load on the network. That did not just apply to computers in a cluster but also to computers in different geographical locations.

4.5 Future Work

The number of servers available for experiments was small and therefore for large data sets the R implementation was taking too long. Hence, many servers would be required for the algorithm's performance to be tested on larger data sets. The clustering algorithms have a common structure [17] that would benefit from *partial aggregation* therefore it would be interesting to see how other algorithms get impacted.

Chapter 5

Concluding Remarks

In this thesis, we looked at the implementation of a parallel K-means algorithm in a distributed environment. We had two goals to look at. First, we wanted to establish how *partial aggregation* impacts the performance of the algorithm. The second goal was to compare the performance of R and Java implementations of the algorithm.

We showed that the presence of a combiner reduced the running time of the algorithm by a factor that was linearly proportional to the size of the data set. Furthermore, we showed that the combiner makes the algorithm scale well with the distributed system. Therefore, *partial aggregation* allowed the algorithm to run in shorter time when the number of servers in the cluster increased.

In addition it became obvious that the Java implementation was much faster than the R counterpart. The factor by which it was faster was linearly proportional to the data set size.

Appendix A

Code

The code for K-means provided in this chapter was adapted from the code in [22]. The code was written in R. Below is the description of the parameters used in the program:

1.**dpoints**: hdfs path to the file with data points.
2.**disfunc**: function for calculating distance between data points.
3.**nofcents**: number of clusters.
4.**centroids**: the initial centres otherwise the first four data points are considered as the initial centres.
5.**dmns**: the dimension of the data points.
6.**iterations**: is the number of iterations.

Below is the code:

```
library(rmr)
library(rhdfs)
kmeansfun =
function(dpoints, disfunc, nofcents = 4, centroids = NULL, dmns = 2 ) {
from.dfs( mapreduce (input = dpoints,
input.format=make.input.format( “csv” , sep = “,” ),
map = if ( is.null ( centroids ) ) {
function( k , v )
if( k <= nofcents )
keyval( k , v )
}
else {
function( k , v ) {
dist = disfunc( centroids , v )
keyval( which.min( dist ) , v ) } },

#Combine function
combine = function( k , v )keyval( k , data.frame( lapply( do.call
( rbind , v ) , function(x){ colSums( as.matrix( x ) ) } ) , as.numeric( length(
v ) ) ) ) ),

#Reduce function reduce = function( k , vv ){
keyval ( k , redc( vv , dmns ) ) } ) , to.data.frame = T )}

#This function calls the above function implementing the KMeans
#according to the number of iterations. If the means converge, it will
#stop. Otherwise it runs the number of iterations provided.
kmeansitn = function( dpoints, iterations = 10, disfunc = distancef , cen-
troids = NULL ) {
start <- Sys.time()
newCentroids = kmeansfun( dpoints , disfunc )
for(i in 1:iterations) {
temp <- newCentroids
newCentroids = kmeansfun(dpoints , disfunc , centroids = newCentroids )
testd <- setdiff( as.matrix( temp ) , as.matrix( newCentroids ) )
testl <- length( testd )
print( i ) ; print( temp )
if( all( testl == 0 ) ){
print( “Running Time:” ) ;print( Sys.time() - start )
print( “Means not changing” )
break
}
}
newCentroids
```

```
print( “Running Time:” ) ; print( Sys.time() - start )
}

#distance calculation function
distancef = function( cs , dps ) {
tcs <- cs[ -1 ]
temp <- rbind( tcs , dps )

    d <- as.matrix( dist( temp, method= “euclidean” ) )
d[ nrow( tcs ) + 1:nrow( dps ) , 1:nrow( tcs ) ]

    }

#Function part of the reduce function.
redc = function( xv , dmns ){

    rs <- data.frame( lapply( do.call( rbind , xv ) , function( x ){
mat <- as.matrix( x );
colSums( mat ) } ) );
sweep( rs[ 1 : dmns ] , 1 , rs[ dmns + 1 ] , “/” )
}
```

Appendix B

Execution

To run the code in the previous chapter, the Rhadoop [20] framework is required. Two of the packages that are part of the framework are rhdfs and rmr. The rmr package allows programmers to express their problems in the form of MapReduce model in R while the rhdfs package allows access to the HDFS. The Hadoop distribution used was Cloudera's CDH3

The necessary information on how to install the packages together with their dependencies can be accessed from [20]. And below is an example of how the program can be run:
1. Provide the path to the file with the data points. The file must be uploaded to hdfs e.g input = "/usr/local/hadoop/hadoop_tmp/csv".
2. Then call the function that repeatedly executes the function defining kmeans like this: kmeansitn(input).
The kmeansitn function accepts more parameters that are described in the previous chapter. Also the user can define their own function for calculating distance which can be used as a parameter in kmeansitn.

Bibliography

[1] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, In Proc. of the 6th Symposium on Operating Systems Design and Implementation, San Francisco CA, Dec. 2004.

[2] K. Shvachko, H. Kuang, S.Radia, R. Chansler, The Hadoop Distributed File System, IEEE, 2010.

[3] Hadoop MapReduce. Web, 2012. http://hadoop.apache.org/mapreduce/

[4] A. K. Jain, R. C. Dubes, Algorithms For Clustering Data. Prentice Hall, 1988.

[5] W. Zhao, H. Ma, Q. He, Parallel K-Means Clustering Based on MapReduce, In: CloudCom 2009. LNCS, vol. 5931, pp. 674-679 (2009).

[6] Mahout. Web, 2012. https://cwiki.apache.org/confluence/display/MAHOUT /K-Means+Clustering

[7] Hadoop. Web. http://hadoop.apache.org/

[8] A. Silvescu, V. Honavar, Independence, Decomposability and functions which take values into an Abelian group, In:Proceedings of the AIMATH'2006, 2006.

[9] I. S. Dhillon, D. S. Modla, A Data-Clustering Algorithm Distributed Memory Multiprocessor, Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence, 1999, pp. 245-260.

[10] S. Kantabutra and A. L. Couch, Parallel K-Means Clustering Algorithm on NOWs, NECTEC Technical Journal, Vol.1, No.6, 2000, pp. 243-248.

[11] R. Farivar, D. Rebolledo,E. Chan and R. H. Campbell, A Parallel Implementation of K-Means Clustering on GPUs, In:Proceedings of the Internation Conference on Parallel and Distributed Processing Techniques and Applications, July, 2008, pp. 340-345.

[12] CUDA. Web, 2009. http://developer.download.nvidia.com/compute/cuda/docs/ CUDA_Architecture_Overview.pdf

[13] Y. Yu, P. K. Gunda and M. Isard, Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations, In Proceedings of SOSP, 2009.

[14] Y. Yu, P. K. Gunda, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson and J. Currey, DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language, In Proceedings of the 35th SIGMOD International Conference on Management of data, 2009.

[15] L. Chai, A. Hartono and D. K. Panda, Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters, In Proceedings of IEEE International Conference on Cluster Computing, 2006.

[16] D. Buntinas, G. Mercier and W. Gropp, Data Transfers between Processes in an SMP System: Performance Study and Application to MPI, In Proceedings of International Conference on Parallel Processing, 2006.

[17] L. Glimcher, R. Jin and G. Agrawal. Middleware for data mining applications on clusters and grids. Journal of Parallel and Distributed Computing, 68(1):37-53, 2008.

[18] C. Ranger, R. Raghuraman, A. Penmestsa, G. Bradski and C. Kozyrakis, "Evaluating MapReduce for Multicore and Multi-processor Systems,"In Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture, 2007, pp. 13-24.

[19] R. Web, 2012. http://www.r-project.org/

[20] RHadoop. Web, 2011. http://blog.revolutionanalytics.com/2011/09/mapreduce-hadoop-r.html

[21] Streaming. Web, 2007. http://hadoop.apache.org/common/docs/r0.15.2/streaming.html

[22] Kmeans. Web, 2012. https://github.com/RevolutionAnalytics/RHadoop/tree/master/rmr/pkg/tests