MSc Project Final Report

K-means Clustering Using Hadoop MapReduce

Grace Nila Ramamoorthy

A thesis submitted in part fulfilment of the degree of

MSc Advanced Software Engineering in Computer Science

Supervisor: Prof. M-Tahar Kechadi

Moderator: Dr. Neil Hurley

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin
September 16, 2011

Table of Contents

Abstract 6

1 Introduction 8

1.1 Purpose of the Study 8

1.2 Layout of the Report 9

2 Background Technical Details 10

2.1 K-means 10

2.2 MapReduce 10

2.3 Hadoop Framework 11

2.4 Choice of Python 13

3 Current Work and Related Research 15

4 Description of Algorithms 17

4.1 MapReduce 17

4.1.1 Start up 17

4.1.2 Mapper 18

4.1.3 Reducer 20

4.2 Sequential Clustering 20

4.2.1 Start up 20

4.2.2 Simple Assignment 21

4.2.3 Centroid Calculator 21

5 Experimental Results and Analysis 23

5.1 Infrastructure Used 23

5.2 Data Points Generation 23

5.3 Experiment A: Mapper and Reducer 24

5.4 Experiment B: With Combiner 27

5.5 Experiment C: Effect of Number of Nodes 32

5.6 Complexity Issues 34

5.6.1 Complexity of the Algorithms 34

5.6.2 Comparison of the Complexity of the Algorithms 35

6 Conclusion 37

Appendix: Complete Python Code 39

List of Figures

2.1 Architecture of MapReduce 11

2.2 Architecture of HDFS 12

4.1 Structure of the Algorithm 18

5.1 Random Generated 2d-Dataset. 23

5.2 Random Generated 3d-Dataset. 24

5.3 Final Clusters of 2d Data. 25

5.4 Final Clusters of 3d Data. 25

5.5 Comparative Plot: Time to Cluster 2d and 3d Points. 25

5.6 Convergence Plot for Multiple Runs. 26

5.7 Comparative Plot 2d: Hadoop MapReduce vs Sequential Clustering. 26

5.8 Comparative Plot 3d: Hadoop MapReduce vs Sequential Clustering. 27

5.9 Comparative Plot: Hadoop MapReduce vs MapReduce with Combiner. . . . 28

5.10 Comparative Plot: Hadoop MapReduce with Combiner vs Sequential Clustering. 31

5.11 Convergence Plot of 2d Points for Multiple Nodes. 32

5.12 Convergence Plot of 3d Points for Multiple Nodes. 33

List of Tables

5.1 Size of the Sample Dataset 23

5.2 Time Taken (seconds) for One Iteration of Hadoop MapReduce for 2d Points 27

5.3 Time Taken (seconds) for One Iteration of MapReduce with Combiner 28

List of Algorithms

1 Algorithm for Startup 17
2 Algorithm for Mapper 19
3 Algorithm for Reducer 20
4 Algorithm for Startup 21
5 Algorithm for SimpleAssignment 22
6 Algorithm for CentroidCalculator 22
7 Algorithm for Mapper with Combiner 29
8 Algorithm for Reducer with Combiner 30

Abstract

Cluster is a group of objects that are similar amongst themselves but dissimilar to the objects in other clusters. Identifying meaningful clusters and thereby a structure in a large unlabelled dataset is an important *unsupervised* data mining task. Technological progress leads to enlarging volumes of data that require clustering. Clustering large datasets is a challenging resource-intensive task and the key to scalability and performance benefits is to use parallel or concurrent clustering algorithms. MapReduce is a framework that performs such resource-intensive tasks in a parallel manner by distributing the computations on large number of nodes. In this study, we examine the applicability of MapReduce in clustering. The goal of this study is to perform k-means clustering using Hadoop MapReduce and compare the results with traditional sequential k-means clustering.

Acknowledgments

My sincere thanks to

. . . my supervisors, Dr.Ilias Savvas and Dr.Tahar Kechadi, for their patience, advice and guidance.

. . . my husband Ram, and children, Daniel and Deborah, for their endless source of encouragement and motivation, not only while doing this project but also during the entire academic year.

. . . An Lekhac for his help while working with the distributed system in UCD.

. . . my classmates, Don Garry and Dawei Yang, for their support during the long summer days while working on the project.

Chapter 1: **Introduction**

Cluster analysis is a study of algorithms and methods of classifying objects. Cluster analysis does not label or tag and assign an object into a pre-existent structure; instead, the objective is to find a valid organisation of the existing data and thereby to identify a structure in the data. It is an important tool to explore pattern recognition and artificial learning.

A cluster is described as a set of similar objects or entities collected or grouped together. All entities within a cluster are alike and the entities in different clusters are not alike. Each entity may have multiple attributes, or features and the likeness of entities is measured based on the closeness of their features. Therefore, the crucial point is to define *proximity* and a method to measure it.

There are many clustering techniques and algorithms in use. K-means is the most common and often used algorithm. K-means algorithm takes an input parameter k, and partitions a set of n objects into k clusters according to a similarity measure. The mean values, or centroids, are a summary measure of the similarity of data objects within the same cluster. First, the algorithm randomly chooses k initial centroids and then iterates over the dataset. In each iteration k-means uses the similarity metric to associate each data object with its nearest centroid. Then, the algorithm computes a set of new centroids by taking the mean of all the data objects in each cluster respectively. The final step is to calculate the change in the centroid positions between the latest iteration and the previous one. The iteration terminates when the change in the centroid position is less than some pre-defined threshold.

With datasets in petabytes, this iteration of data point assignment and centroid calculation could be tedious tasks. One common method used to cluster huge datasets is to filter and reduce the dataset into a sample set and use this small representative set to find the probable cluster centres. The challenge here is in identifying the representative sample set. The choice of sample set directly impacts the final cluster centres. The other method is to distribute the dataset on multiple nodes and perform the calculation of centroids in a parallel mode on chunks of data at a time. This is similar to the Single Program Multiple Data(SPMD) algorithms that can be implemented using threads, MPI or MapReduce. The choice of an appropriate implementation strategy is based on the size of the dataset, complexity of the computational requirements, algorithmic synchronisation, ease of programming and the available hardware profile. The type of applications we target have loosely coupled, easily distributable high volume data that require simple computation.

1.1 Purpose of the Study

The purpose of this study is to explore the possibility of using Hadoops MapReduce framework to identify clusters in large datasets. MapReduce is a programming model to process large datasets. The programs written in this style are highly scalable, automatically parallelised and can be executed on large clusters of commodity hardware.

The objective of the study is to use the MapReduce framework on a large dataset to identify the clusters within that dataset using parallelisation. We will compare the results obtained

using traditional k-means clustering algorithm and the MapReduce clustering algorithm to evaluate any performance benefit of using MapReduce algorithm.

1.2 Layout of the Report

This is a brief outline of the remainder of the report. Chapter 2 gives an overview of the technical aspects of MapReduce and Hadoop; Chapter 3 focusses on current work and existing research; Chapter 4 describes the design and the algorithm we have used in this study; Chapter 5 explains the results and observations; Chapter 6 states the conclusion drawn from the study.

Chapter 2: **Background Technical Details**

In this study, we use Hadoop MapReduce framework to perform k-means clustering. First, we will look at a brief overview of k-means, MapReduce and Hadoop framework.

2.1 K-means

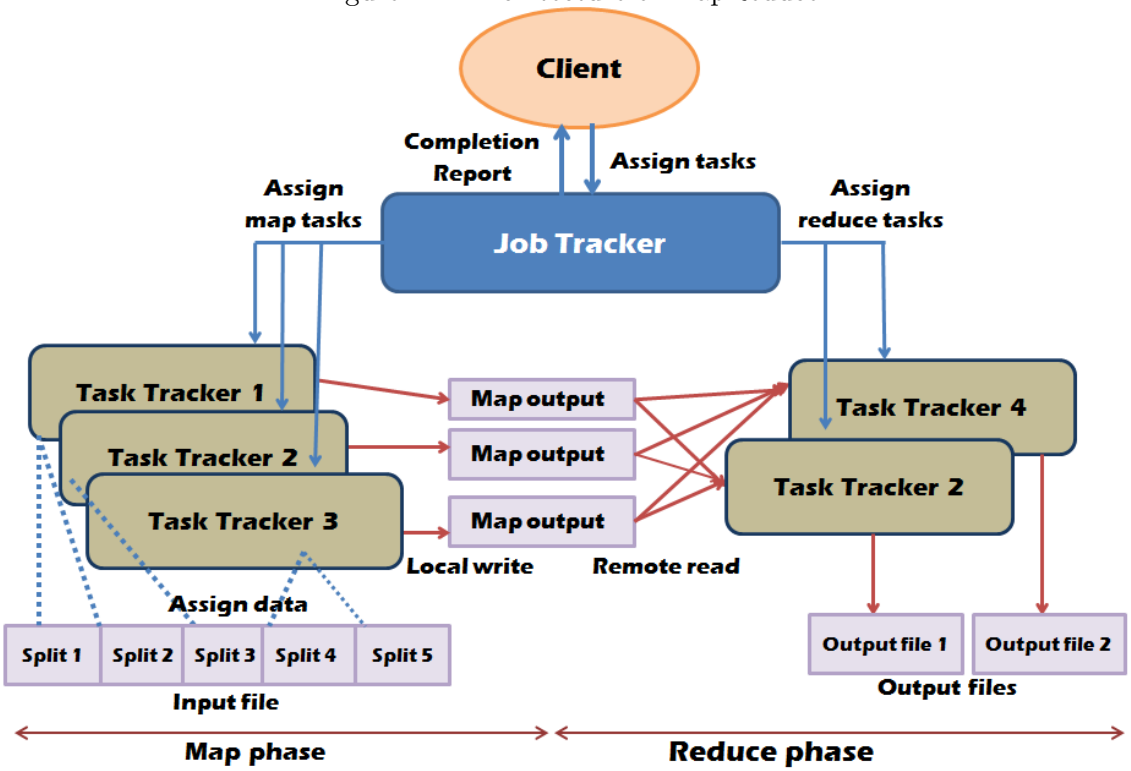
K-means is a common and well-known clustering algorithm. It partitions a set of n objects into k clusters based on a similarity measure of the objects in the dataset. The clusters have a high intra-cluster and a low inter-cluster similarity. As the number of objects in the cluster varies, the centre of gravity of the cluster shifts. This algorithm, starts off with the selection of the k initial random cluster centres from the n objects. Each remaining object is assigned to one of the initial chosen centres based on similarity measure. When all the n objects are assigned, the new mean is calculated for each cluster. These two steps of assigning objects and calculating new cluster centres are repeated iteratively until the convergence criterion is met. Comparing the similarity measure is the most intensive calculation in k-means clustering. For n objects to be assigned into k clusters, the algorithm will have to perform a total of nk distance computations. While the distance calculation between any object and a cluster centre can be performed in parallel, each iteration will have to be performed serially as the centre changes will have to be computed each time.

2.2 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large datasets. A typical MapReduce computation processes many terabytes of data on thousands of machines. MapReduce usually splits the input dataset into independent chunks. The number of splits depends on the size of the dataset and the number of nodes available. Users specify a map function that processes a (*key, value*) pair to generate a set of intermediate (*key, value*) pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. [5]

There are separate Map and Reduce steps. Each step done in parallel on sets of (*key, value*) pairs. Thus, program execution is divided into a Map and a Reduce stage, separated by data transfer between nodes in the cluster. The Map stage takes in a function and a section of data values as input, applies the function to each value in the input set and generates an output set. The Map output is a set of records in the form of (*key, value*) pairs stored on that node. The records for any given key could be spread across many nodes. The framework, then, sorts the outputs from the Map functions and inputs them into a Reducer. This involves data transfer between the Mappers and the Reducer. The values are aggregated at the node running the Reducer for that key. The Reduce stage produces another set of (*key, value*) pairs as final output. The Reduce stage can only start when all the data from the Map stage is transferred to the appropriate machine. MapReduce requires the input as a (*key, value*)

Figure 2.1: Architecture of MapReduce



pair that can be serialised and therefore, restricted to tasks and algorithms that use (*key*, *value*) pairs.

The MapReduce framework has a single master Job Tracker and multiple Task Trackers. Potentially, each node in the cluster can be a slave Task Tracker. The master manages the partitioning of input data, scheduling of tasks, machine failures, reassignment of failed tasks, inter-machine communications and monitoring the task status. The slaves execute the tasks assigned by the master. Both input and output are stored in the file-system. The single Job Tracker can be a single point failure in this framework.

MapReduce is best suited to deal with large datasets and therefore ideal for mining large datasets of petabytes size that do not fit into a physical memory. Most common use of MapReduce is for tasks of additive nature. However, we can tweak it to suit other tasks.

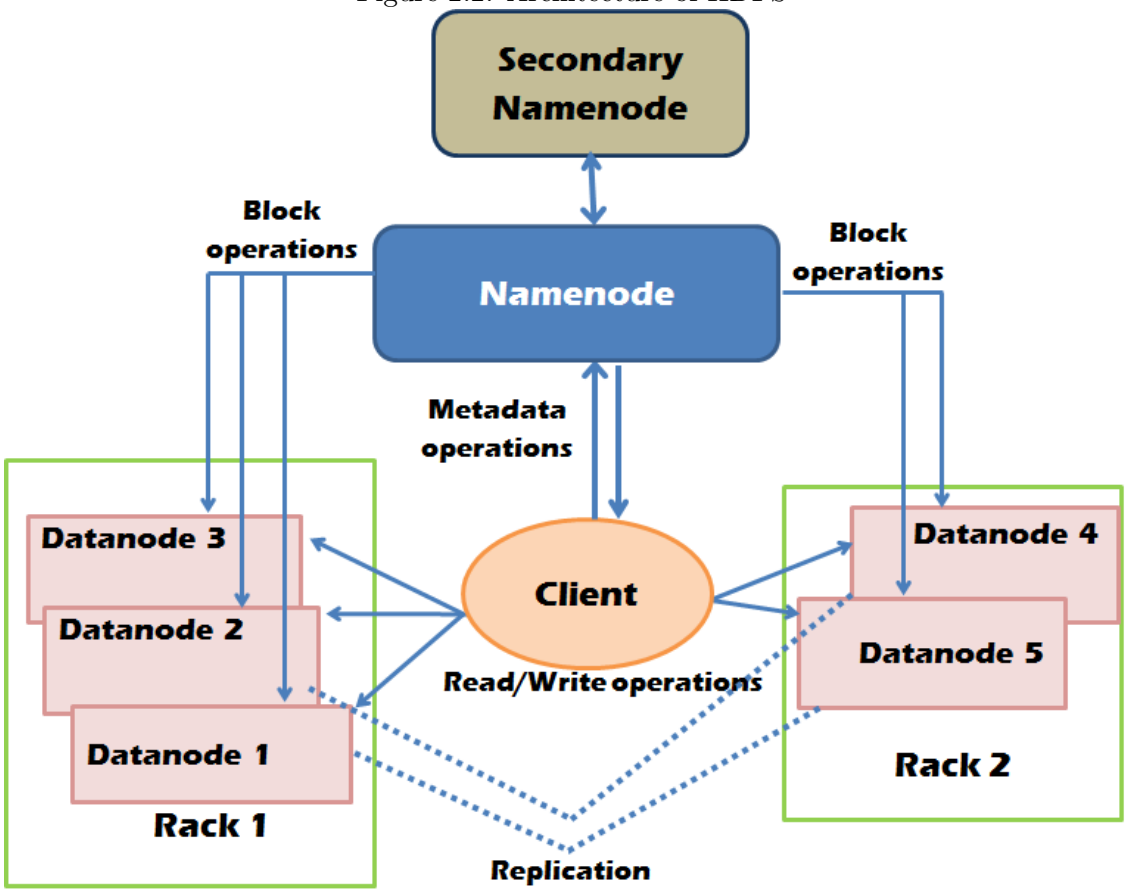
2.3 Hadoop Framework

Hadoop is an open soure software framework that can run large data-intensive, distributed applications and can be installed on commodity Linux clusters. Hadoop comes with its own filesystem called the Hadoop Distributed File System (HDFS) and a strong infrastructural support for managing and processing huge petabytes of data.

Each HDFS cluster consists of one unique server called the *Namenode* that manages the namespace of the file system, determines the mapping of blocks to Datanodes, and regulates file access. Each node in the HDFS cluster is a *Datanode* that manages the storage attached to it. The datanodes are responsible for serving read and write requests from the clients and

performing block creation, deletion and replication instructions from the Namenode.

Figure 2.2: Architecture of HDFS



HDFS is built to deal with large files by breaking them into blocks and replicating them across the network. The common policy is to replicate each file three times. HDFS will place two replicas on two different nodes on the same rack and place the third replica on a different node in a different rack. This ensures reliability even without a RAID backup. However, HDFS is based on the Google File System model that follows the write-once-read-many access model that are not meant for continuous updates of data.

The infrastructural support of Hadoop is built around the assumption that data will be distributed on hardware that *will* fail. Therefore, when tasks are run on multiple nodes concurrently, Hadoop can automatically detect task failures and restart failed tasks on other healthy nodes. When it notices a block or a node is lost, it automatically creates a copy of the missing data from the available replicas. The only single point of failure is the presence of a single Namenode for the entire HDFS file system. Even though HDFS supports an optional secondary Namenode, it does not provide real redundancy. The secondary Namenode connects to the primary Namenode from time to time to take a snapshot of the directory information. The function of the secondary Namenode is to facilitate a quick recovery and restart of the primary Namenode. A real Backup Namenode would enhance the availability of HDFS.

Hadoop places emphasis on high throughput over low latency and is ideal for batch processing. Hadoop also works on the premise that it costs less to move the computation than data. Therefore, HDFS provides interfaces to move the applications closer to the data - either on the same node as the data or the same rack thereby improving I/O bandwidth.

Hadoop is very different from relational databases as Hadoop facilitates applications, such

as text analysis or image processing, that require analysing *all* the available data. Hadoop streaming is a utility that allows users to run any executable as a mapper and /or a reducer. This is based on fault-tolerant, shared-nothing architecture where tasks are not dependent on each other. The only exception is that the reducer waits for the output of the mappers under Hadoop control. However, unlike parallel programming that uses MPI to pass data around, Hadoop data flow is implicit and handled automatically without any coding. For large-scale MapReduce tasks, Hadoop uses a MapReduce execution engine to ensure fault-tolerant, distributed applications on a cluster of heterogeneous commodity machines. Hadoop stores the intermediate results of map tasks locally in the disks where the map computations are executed. It then informs the reducers to retrieve the data for further processing. If a map task fails or a mapper is slow, the task is reassigned to another mapper by the Job Tracker. If a reducer fails then the stored intermediate map results can be used to reassign the reducer task to another node. This improves the robustness of the technology at the cost of increased overhead in communication. This could be a challenge for resource-intensive tasks, such as, k-means clustering.

There are many open source projects built on top of Hadoop. Hive is a data warehouse framework[4] used for adhoc querying and complex analysis. It is designed for batch processing. Pig [3] is a data flow and execution framework that produces a sequence of MapReduce programs. Mahout [1] is used to build scalable machine learning libraries that focus on clustering, classification, mining frequent item-sets and evolutionary programming. Pydoop is a python package that provides an API for Hadoop MapReduce and HDFS.

2.4 Choice of Python

Before we continue, we need to explain the choice of python for this study. Multi-threading is a popular feature used in parallel programming. However, coordinating the sharing of data access between the threads is a complex task that requires semaphores and locks. These special objects need to be used with great care to avoid dead locks that occur when two separate threads hold a lock on some data that the other thread requires. This results in a situation where inadvertently two threads wait for locks to be released by each other. If we can eliminate shared state, then we can avoid the complexity of co-ordination. This is the fundamental concept of functional programming.

In functional programming, data is explicitly passed between functions as parameters or return values and can only be changed by the active function at that moment. As there are no shared states, there are no hidden dependencies. If all the functions are connected to each other using a directed acyclic graph(DAG), the functions in the DAG, that do not share a common ancestor, can be executed in parallel. When there is no hidden dependency from shared state, parallel execution of functions become easier.

MapReduce framework is built around map and reduce concepts of functional programming. Though Python is not a functional programming language, it has a built-in support for both map and reduce concepts. MapReduce framework requires that the reduce function wait for map function to complete. Python has a built-in Global Interpreter Lock(GIL) that assures that only one thread executes Python byte-code at a time. This makes it easier for the interpreter to be multi-threaded but handicaps parallelism by serialising the execution of user-threads. When needed, one can side-step the GIL by using the multiprocessing package which uses sub-processes instead of threads. The multiprocessing package has a Pool class that controls a pool of processes and has a map function of its own that partitions and distributes input to a user-specified function in a pool of worker processes. So we can use

this Pool class to create the required number of worker processes and perform a distributed map and distributed reduce operation.

In addition, the availability of iterators and generators, that defer the production of elements in a sequence until the element is needed, reduces computational expenses and memory consumption. This makes Python a language of choice while dealing with petabytes of data.

Now that we have established the technical background for the study, next, we will look at the current research in this area.

Chapter 3: **Current Work and Related Research**

There have been extensive studies on various clustering methodologies. In this chapter, we will look at some of the studies carried out in the specific area of k-means clustering and MapReduce. Since the development of k-means clustering algorithm in mid 1970's, there have been many attempts to tune the algorithm and improve its performance. There are two important areas that concern most researchers. First, the accuracy of k-means clustering which is dependent on the choice of the number of clusters and the position of initial centroids. Secondly, the iterative nature of k-means algorithm that impacts scalability of the algorithm as the size of the dataset increases. Researchers have come up with algorithms that:

- Improve the accuracy of final clusters
- Help in choosing appropriate initial centroids
- Reduce the number of iterations
- Enhance scalability with increased dataset size or dimensionality
- Handle outliers well

The effectiveness of clustering is dependent on the choice of initial centroids. The right set of initial centroids create clean clusters and reduce the number of iterations. Jin, Goswami and Aggarwal, [12] in their paper, have presented a new algorithm called the Fast and Exact K-means clustering. This algorithm uses sampling technique to create the initial cluster centres and thereafter requires one or a small number of passes on the entire dataset to adjust the cluster centres. The study has shown that this algorithm produces the same cluster centres as the original k-means algorithm. The algorithm has also been tested on distributed system of loosely coupled machines.

In the paper, on Efficient Clustering of High-Dimensional datasets, Andrew McCallum, Kamal Nigam and Lyle Ungar have explained how one can reduce the number of iterations by first partitioning the dataset into overlapping subsets and iterating over only the data points within the common subset. This technique is called Canopy Clustering and it uses two different similarity measures. First, a cheap and approximate similarity measure is used to create the canopy subsets and then a more and accurate measure is used to cluster the points within the subset. This reduces the total number of distance calculations and, thereby, the number of iterations. [9]

While the above studies largely concentrated on tuning the k-means algorithm and reducing the number of iterations, there have been many other studies on the scalability of k-means algorithm. Researchers have identified algorithms and frameworks to improve scalability. In the recent days, more research has been done on MapReduce framework. Zhao, Ma and He, [13] in their paper on Parallel k-means clustering based on mapreduce, have demonstrated that k-means clustering algorithms can scale well and can be parallelised. They suggest that MapReduce can efficiently process large datasets on commodity hardware.

Some researches have compared various MapReduce frameworks available in the market and studied their effectiveness in clustering large datasets. Ibrahim *et.al.* [10] have analysed the performance benefits of Hadoop on virtual machines. Thereby proving that MapReduce is a good tool for cloud based data analysis. Ekanayake *et.al.* [6] have experimented with

Microsoft product DryadLINQ to perform data intensive analysis and compared the performance of DryadLINQ with Hadoop implementations. In another study, Ekanayake *et.al.* [7] have implemented a slightly enhanced model and architecture of MapReduce called the *Twister*. In their paper, they have compared the performance of Twister with Hadoop and DryadLINQ with the aim of expanding the applicability of MapReduce for real scientific applications. According to their study, Twister is well suited for iterative, long running MapReduce jobs as it uses *streaming* to send the output of the map functions directly to reduce function instead of local writes by the mappers and read by the reducer. There were two important observations they made in this study. First, for computation intensive workload, threads and processes did not show any significant difference in performance. Second, for memory intensive workload, processes perform 20 times faster than threads. Jiang, Ravi and Agarwal[11] have done a comparative study of Hadoop MapReduce and Framework for Rapid Implementation of Datamining Engines. According to their study, they have implied that Hadoop is not well suited for modest-sized databases. However, when the datasets are large, there is a performance benefit in using Hadoop.

There are many open source projects that work on MapReduce based algorithms. Apache Mahout is an Apache project that aims to build distributed and scalable machine learning algorithms on Hadoop. [1] It has a ready suite of algorithms for clustering, classification and collaborative filtering that work on top of Hadoop MapReduce framework. Disco is another open source MapReduce runtime developed using Erlang, a functional programming language. [2] While Disco stores the intermediate results of map tasks in local files similar to Google and Hadoop MapReduce architectures, it employs HTTP to access this data for the reduce tasks. It also differs from HDFS in that it does not support a distributed file system but expects the files to be distributed initially over the multiple disks of the cluster.

From all this, we establish that parallel clustering of large datasets is an active area of research interest. Distributing large datasets can improve the performance of resource-intensive clustering. There are a number of studies on the applicability of Hadoop MapReduce for various data and compute intensive applications. Our study is to assess the effectiveness of using Hadoop MapReduce for k-means clustering and comparing the performance benefits of distributing the dataset and the clustering process.

Chapter 4: **Description of Algorithms**

Now that we have established that k-means algorithm can be parallelised and MapReduce is a probable parallel framework, in this chapter, we will describe the algorithms that we use for the study.

4.1 MapReduce

K-means is a clustering algorithm used to cluster a set of data objects into a given number of clusters based on the distance measure between the points. Our first step is to generate a random dataset that would represent real-life data objects and create a seed centroid file.

4.1.1 Start up

Algorithm 1 Algorithm for Startup

Require:

- A set of d-dimensional objects X= $\{x_1, x_2, \ldots, x_n\}$
- k-number of clusters where $k < n$
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$
- δ convergence delta

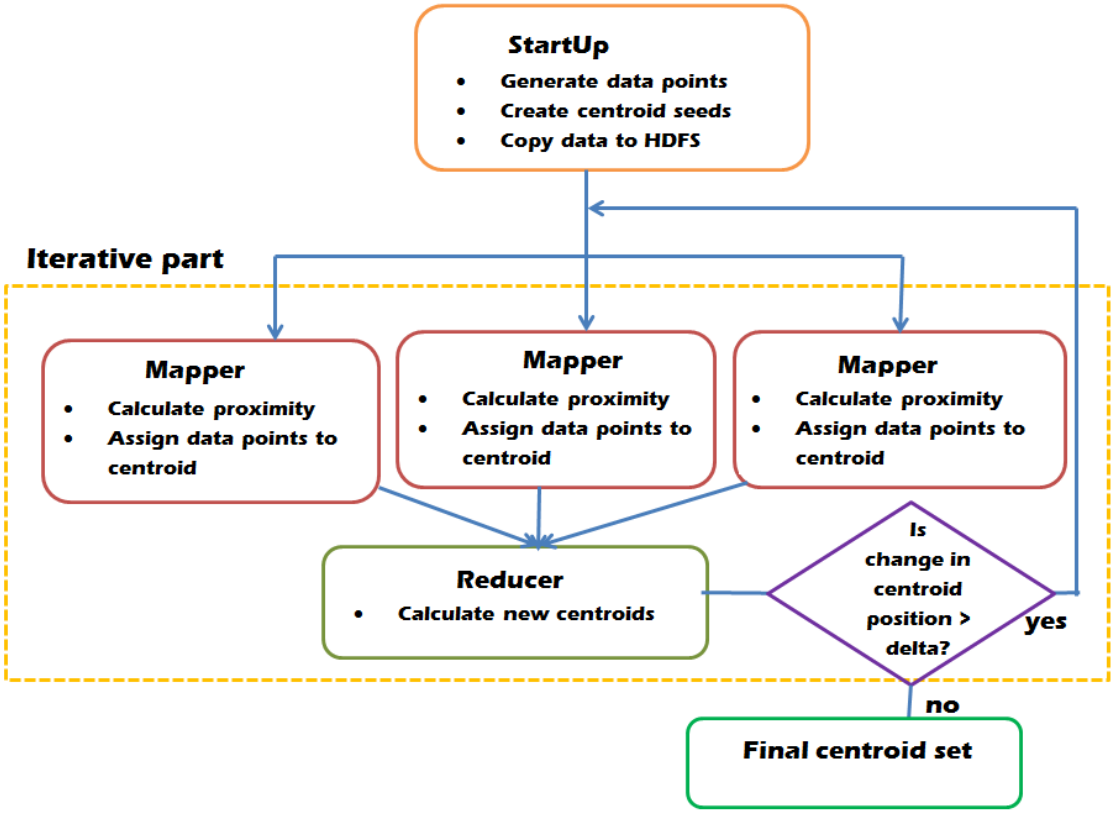
Output: a new set of centroids, number of iterations, final clusters, time taken to converge

1: $load\,(X, C)$
2: $current\_centroids \Leftarrow C$
3: $initialise$ numIter, timetoConverge, finalClusters
4: $startTime \Leftarrow currentTime()$
5: $C' \Leftarrow perform\ MapReduce$
6: $new\_centroids \Leftarrow C'$
7: $numIter \Leftarrow 1$
8: **while** $change(new\_centroids, current\_centroids) > \delta$ **do**
9: $current\_centroids \Leftarrow new\_centroids$
10: $C' \Leftarrow$ perform MapReduce
11: $new\_centroids \Leftarrow C'$
12: $numIter \Leftarrow numIter + 1$
13: **end while**
14: $endTime \Leftarrow currentTime()$
15: $timetoConverge \Leftarrow (endTime - startTime)$
16: $perform\ outlierRemoval$
17: $finalClusters \Leftarrow perform\ finalClustering$
18: $writeStatistics$
19: **return** $current\_centroids, numIter, finalClusters, timetoConverge$

We use a start-up program to initiate the process by generating random objects and inputting the initial set of centroids. Then, we call the mapper algorithm to assign the objects to clusters by calculating the distance between the centroids and the objects. The mapper algorithm calls the reducer to recalculate new centroids based on the current clustering. The start-up program, then, evaluates the change in the centroid positions and compares it with the convergence delta that we have defined. If the change is more than δ, the start-up program iterates through the mapper and reducer functions again. When the change in centroid position is less than δ, we assume convergence. At this point, the algorithm generates the output files and creates the final clusters.

The structure the algorithm is given below:

Figure 4.1: Structure of the Algorithm



4.1.2 Mapper

The input dataset of objects is distributed across the mappers. The initial set of centroids is either placed in a common location and accessed by all the mappers or distributed to each mapper. The centroid list has an identification for each centroid as key and the centroid itself as the value. Each input object in the subset $(x_1, x_2, ..., x_m)$ is assigned to its closest centroid by the mapper. We have used Euclidean distance to measure proximity of points. The distance between each point and each centroid is measured and the shortest distance is calculated. Then, the object is assigned to its closest centroid. When all the objects are assigned to the centroids, the mapper sends all the input objects and the centroids they are assigned to, to the reducer.

Algorithm 2 Algorithm for Mapper

Require:

- A subset of d-dimensional objects of $\{x_1, x_2, \ldots, x_n\}$ in each mapper
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$

Output: A list of centroids and objects assigned to each centroid separated by tab. This list is written locally one line per data point and read by the Reducer program.

1: $M_1 \Leftarrow \{x_1, x_2, \ldots, x_m\}$
2: $current\_centroids \Leftarrow C$
3: $distance\,(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$ where p_i (or q_i) is the coordinate of p (or q) in dimension i
4: **for all** $x_i \in M_1$ such that $1 \leq i \leq m$ **do**
5: $bestCentroid \Leftarrow null$
6: $minDist \Leftarrow \infty$
7: **for all** $c \in current\_centroids$ **do**
8: $dist \Leftarrow distance(x_i, c)$
9: **if** $bestCentroid = null \;||\; dist < minDist$ **then**
10: $minDist \Leftarrow dist$
11: $bestCentroid \Leftarrow c$
12: **end if**
13: **end for**
14: $outputlist << (bestCentroid, x_i)$
15: i +=1
16: **end for**
17: **return** *outputlist*

4.1.3 Reducer

The reducer accepts the key,value pair output from the mappers, loops through the sorted output of the mappers and calculates new centroid values. For each centroid, the reducer calculates a new value based on the objects assigned to it in that iteration. This new centroid list is emitted as the reducer output and sent back to the start-up program.

Algorithm 3 Algorithm for Reducer

Require:

Input: (key, value) where key = bestCentroid and value = objects assigned to the centroids by the mapper.

Output: (key, value) where key = oldcentroid and value = newBestCentroid which is the new centroid value calculated for that bestCentroid.

```
1: outputlist ⇐ outputlists from mappers
2: v ⇐ {}
3: newCentroidList ⇐ null
4: for all y ∈ outputlist do
5:     centroid ⇐ y.key
6:     object ⇐ y.value
7:     v[centroid] ⇐ object
8: end for
9: for all centroid ∈ v do
10:    newCentroid, sumofObjects, numofObjects ⇐ null
11:    for all object ∈ v[centroid] do
12:        sumofObjects += object
13:        numofObjects += 1
14:    end for
15:    newCentroid ⇐ (sumofObjects ÷ numofObjects)
16:    newCentroidList << (newCentroid)
17: end for
18: return newCentroidList
```

The start-up algorithm 1 calculates the change in centroid positions, compares it with convergence δ and loops into map and reduce programs, if needed.

4.2 Sequential Clustering

To study the effectiveness of using MapReduce in clustering algorithms, we used a simple sequential clustering algorithm to cluster the same dataset we created for MapReduce. We compared the results of this sequential clustering method with the statistics collected using MapReduce.

The clustering process was repeated five times using both algorithms with different centroid seed points on each dataset. This ensured that we had consistent result for the study.

4.2.1 Start up

The same data points and the initial centroids generated for mapReduce algorithm is used here. We use the Start-up program to start the process and call the SimpleAssignment algo-

rithm to perform the assignment of objects to clusters by calculating the distance between the centroids and the objects. When all objects are assigned to the centroids, the SimpleAssignment program sends all the assigned objects and the centroids they are assigned to, to the CentroidCalculator program. The CentroidCalculator recalculates the new centroids based on the current clustering and sends the list back. The Start-up program, then, evaluates the change in the centroid positions and compares it with convergence δ and loops into simple clustering, if needed.

Algorithm 4 Algorithm for Startup

Require:

- A set of d-dimensional objects X= $\{x_1, x_2, \ldots, x_n\}$
- k-number of clusters where $k < n$
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$
- δ convergence delta

Output: A new set of centroids, number of iterations, final clusters and the time taken to converge

1: $current\_centroids \Leftarrow C$
2: $initialise\ numIter, timetoConverge, finalClusters$
3: $startTime \Leftarrow currentTime()$
4: $C' \Leftarrow$ call $SimpleAssignment$
5: $new\_centroids \Leftarrow C'$
6: $numIter \Leftarrow 1$
7: **while** $change(new\_centroids, current\_centroids) > \delta$ **do**
8: $current\_centroids \Leftarrow new\_centroids$
9: $C' \Leftarrow$ call $SimpleAssignment$
10: $numIter \Leftarrow numIter + 1$
11: $current\_centroids \Leftarrow C'$
12: **end while**
13: $endTime \Leftarrow currentTime()$
14: $timetoConverge \Leftarrow (endTime - startTime)$
15: $perform\ outlierRemoval$
16: $finalClusters \Leftarrow$ perform $finalClustering$
17: $writeStatistics$
18: **return** $current\_centroids$, numIter, $finalClusters$, $timetoConverge$

4.2.2 Simple Assignment

We use Euclidean distance to measure proximity of points. The distance between each point and each centroid is measured and the shortest distance is calculated. Then, the object is assigned to its closest centroid.

4.2.3 Centroid Calculator

This takes the list of centroids and their assigned objects as a parameter from the SimpleAssigment algorithm and calculates the new centroid value. It loops through each centroid, then calculates a new value with the objects assigned in that iteration and outputs a list of new centroid values.

Algorithm 5 Algorithm for SimpleAssignment

Require:

- A set of d-dimensional objects of $\{x_1, x_2, \ldots, x_n\}$
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$

Output: A dictionary list consisting of each centroid and the objects assigned to them. This list is passed as an argument to the CentroidCalculator program.

1: $M_1 \Leftarrow \{x_1, x_2, \ldots, x_n\}$
2: $current\_centroids \Leftarrow C$
3: $distance\,(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$ where p_i (or q_i) is the coordinate of p (or q) in dimension i $v \Leftarrow \{\}$
4: **for all** $x_i \in M_1$ such that $1 \leq i \leq n$ **do**
5: $bestCentroid \Leftarrow null$
6: $minDist \Leftarrow \infty$
7: **for all** c in $current\_centroids$ **do**
8: $dist \Leftarrow distance(x_i, c)$
9: **if** $bestCentroid = null \;||\; dist < minDist$ **then**
10: $bestCentroid \Leftarrow c$
11: $minDist \Leftarrow dist$
12: **end if**
13: **end for**
14: $v[bestCentroid] \Leftarrow (x_i)$
15: i +=1
16: **end for**
17: call $CentroidCalculator(v)$

Algorithm 6 Algorithm for CentroidCalculator

Require:

Input: List of centroids and their assigned objects
Output: List of new centroids

1: $v = \{\}$
2: $v = outputlist$ from $SimpleAssignment$
3: $newCentroidList \Leftarrow null$
4: **for all** $centroid \in v$ **do**
5: $newCentroid, sumofObjects, numofObjects \Leftarrow null$
6: **for all** $object \in v[centroid]$ **do**
7: $sumofObjects$ += $object$
8: $numofObjects$ += 1
9: **end for**
10: $newCentroid \Leftarrow (sumofObjects \div numofObjects)$
11: $newCentroidList$ << $(newCentroid)$
12: **end for**
13: **return** $newCentroidList$

Chapter 5: **Experimental Results and Analysis**

5.1 Infrastructure Used

We used a cluster of 4 nodes. We used one node as Namenode and Job Tracker and that had Intel Pentium 4 CPU, 3.40GHz and 1.72 GB memory. Three other nodes were Datanodes and Task Trackers. Two of the Task Trackers had Intel Pentium 4, 2.8GHz and 2GB memory. One Task Tracker node had Intel Pentium 4, 2.8GHz and 1GB memory. All four nodes ran Red Hat version 4.1.2-48 and 32 bit Cent OS operating system 5.5. The software used were Java Runtime version 1.6.0_16, Python 2.4.3 and Hadoop 0.21.0

5.2 Data Points Generation

We generated the required number of d -dimensional data-points in the Start-up program. After trying different ranges and types of data, we chose to generate data-points in the range of 1 to 900. We generated 15 million 2d and 3d points. We then used the required number of data points randomly from the generated set.

Table 5.1: Size of the Sample Dataset

Data Points (in Million)	1	5	10	11	12
2d (size in MB)	7.5	37.5	75	82.5	90
3d	11	55	110	120	132

The distribution of the datasets before clustering is shown below:

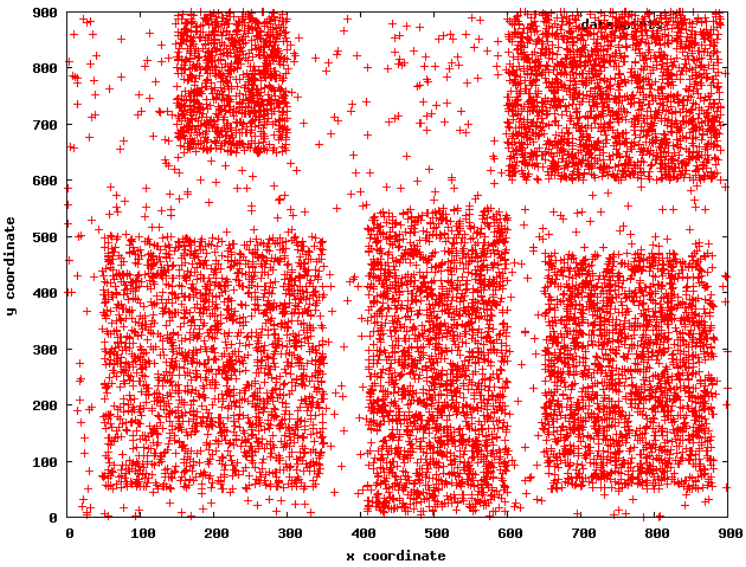


Figure 5.1: Random Generated 2d-Dataset.

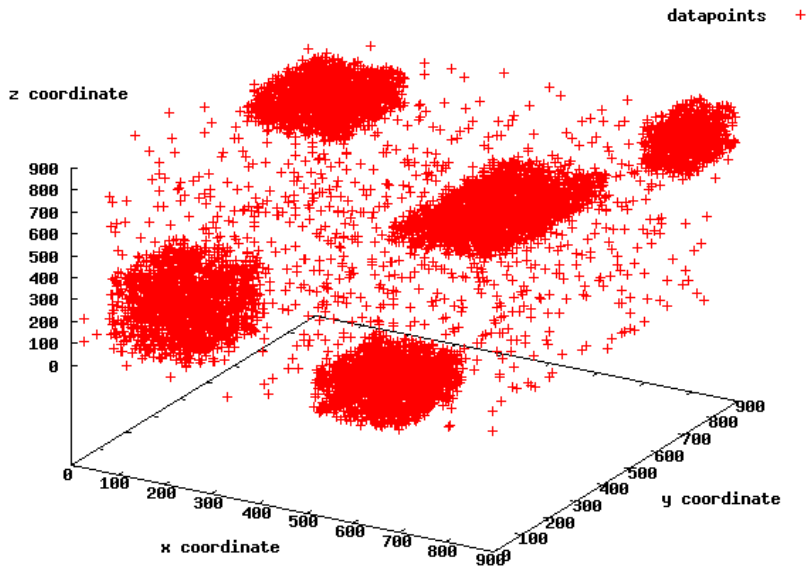


Figure 5.2: Random Generated 3d-Dataset.

Next, we had to identify appropriate initial set of seed centroids. We considered various methods, from random choice of centroids to canopy clustering, to generate centroids. As the quality of the final clusters in k-means is dependent on the initial number and choice of centroids, we decided against random choice. Instead, we chose a more dependable approach and used the same cluster coordinates as the data points to generate 5 different sets of seed centroids in 2d and 3d. This worked well for our study as our aim was not to optimise the choice of initial set of centroids but to analyse the effectiveness of MapReduce to cluster.

5.3 Experiment A: Mapper and Reducer

Once we had generated the data points and the initial set of centroids, we initialised the required statistical parameters, such as, iteration counters and timers. Next,we copied the required files into Hadoop Distributed File System(HDFS). Then, we called the Hadoop MapReduce iteratively until the set of centroids converged. We started with 500 data points of about 5KB in size and gradually increased the number of points up to 13 million about 125MB in size. We clustered each dataset five times using the five input centroid seed sets. Repeating the experiment five times helped us to reliably monitor the time taken to cluster, ensure we could replicate the result with stability and identify any unexplainable circumstance.

The final clustered data points are shown below:

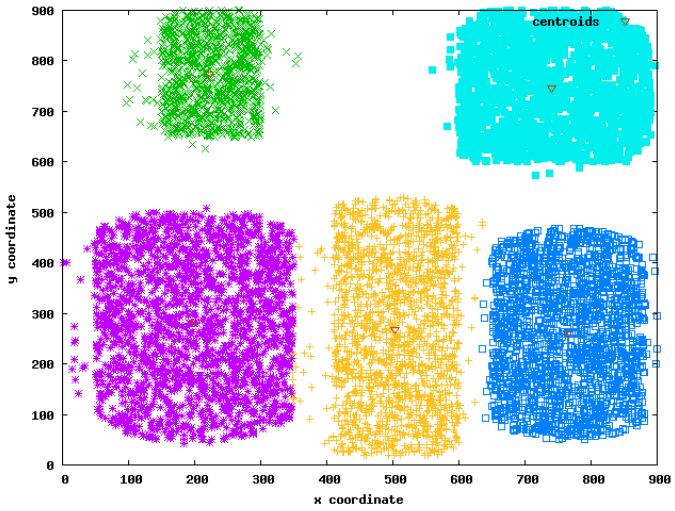


Figure 5.3: Final Clusters of 2d Data.

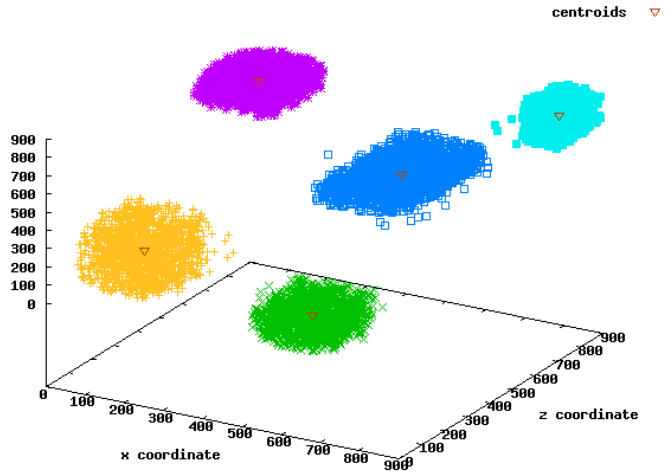


Figure 5.4: Final Clusters of 3d Data.

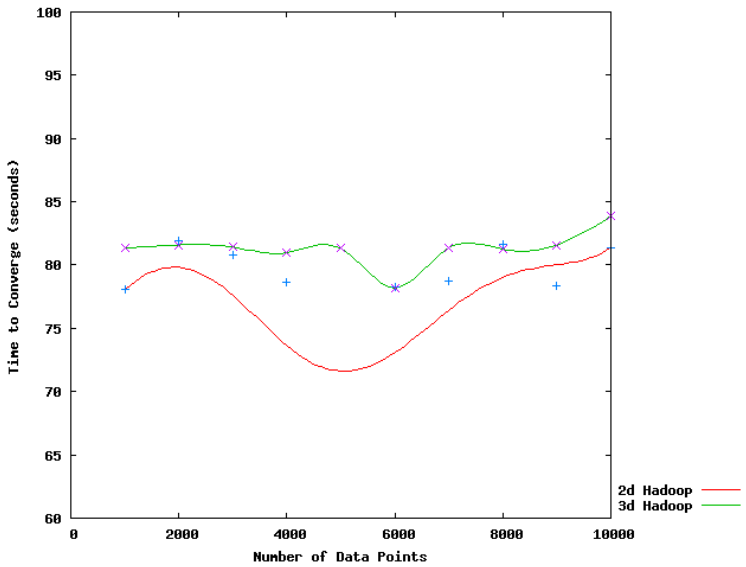


Figure 5.5: Comparative Plot: Time to Cluster 2d and 3d Points.

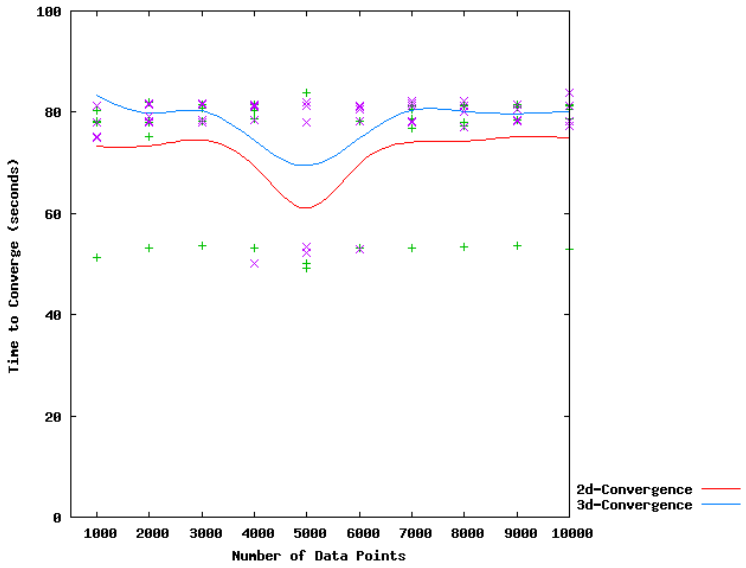


Figure 5.6: Convergence Plot for Multiple Runs.

The algorithm worked well for any dimensional data. To test the robustness of the algorithm, we ran the clustering algorithm on each dataset with 5 slightly different sets of centroid seeds. We can notice from the figure 5.3 that there is some time difference in converging the same dataset with slightly different centroid input sets. We used 2d, 3d and 4d points for our study. At the end of each run, we gathered the statistical data for analysis.

While our purpose was to identify the suitability of using Hadoop MapReduce for k-means clustering, we also needed to study its effectiveness. Therefore, we used a simple sequential clustering program to cluster the same set of data points without MapReduce. In this method, we stored the data points and centroid seeds in local files. We applied a simple sequential clustering algorithm, algorithm 5, to assign the data points to the closest centroid. We used a centroid calculator, algorithm 6, to find the new centroid for the current cluster. We noted the time taken to converge, number of iterations and the final clusters. We used the results from both types of run for our analysis.

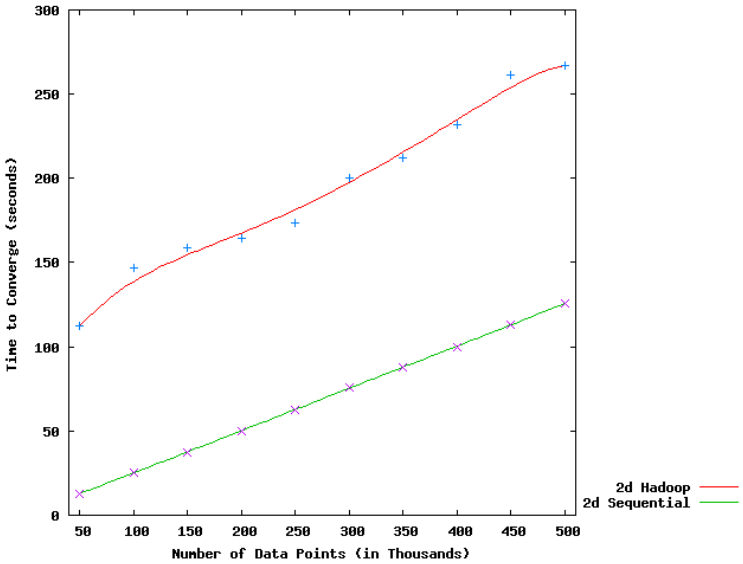


Figure 5.7: Comparative Plot 2d: Hadoop MapReduce vs Sequential Clustering.

We noticed that the simple clustering took much less time and overheads for smaller datasets.

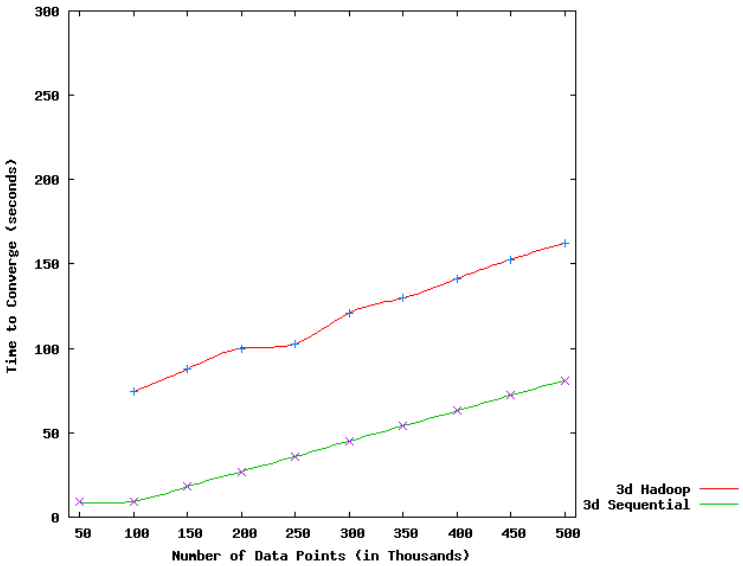


Figure 5.8: Comparative Plot 3d: Hadoop MapReduce vs Sequential Clustering.

However as the size of the dataset increased, the time to converge increased as well. We also noticed that the time taken for Hadoop clustering increased with the size of the dataset.

We analysed the time taken for various jobs in the Hadoop framework to tune the performance. The table below shows the time taken for various tasks in each iteration of MapReduce. Each dataset took 5 iterations to converge.

Table 5.2: Time Taken (seconds) for One Iteration of Hadoop MapReduce for 2d Points

Data Points (in Million)	1	2	3	4	5
Average Map Time (in seconds)	14	28	43	56	69
Shortest Map Time	2	3	4	4	5
Shuffle Task Time	54	84	105	157	207
Average Reducer Time	14	27	30	50	61
Total Time for a MapReduce Cycle	84	148	213	284	340
Number of Spilled Records (in Millions)	2	4	9	12	15
Number of Killed Tasks	0	1	1	1	2
Time Taken to Converge (in seconds)	431	742	1060	1414	1728

We noticed a large number of spilled records and an increase in the time taken to shuffle. There were more number of redundant map and reduce tasks that were started and killed when the data points increased to over 2 million. To reduce the number of duplicated tasks and the shuffling time, and to improve the performance of MapReduce, we added a combiner.

5.4 Experiment B: With Combiner

In the MapReduce framework, the output of the map function is written locally to the disk and the reduce function reads the output from the mappers. When the mappers are ready to write, the output is sorted and shuffled. Our initial experiments showed that the time taken for data shuffling and sorting increased as we increased the number of data points from 50000 to 5 million. The time taken to shuffle 50000 points was about 4 seconds which rose up to

30 seconds for 500000 points and 207 seconds for 5 million points. Sorting and shuffling the map output was time intensive. Therefore, we modified the mapping algorithm slightly to combine the map outputs in order to reduce the amount of data that the mappers write and the reducer reads.

So we created a combiner that reads the mapper outputs and calculates a local centroid in each mapper. The reducer, then, reads the output from the mappers and calculates global centroid. This improved the performance of Hadoop clustering.

Reducing the amount of write in the mapper and the read in the reducer, improved the Hadoop clustering performance. Combining the mapper output locally improved the performance as shown below:

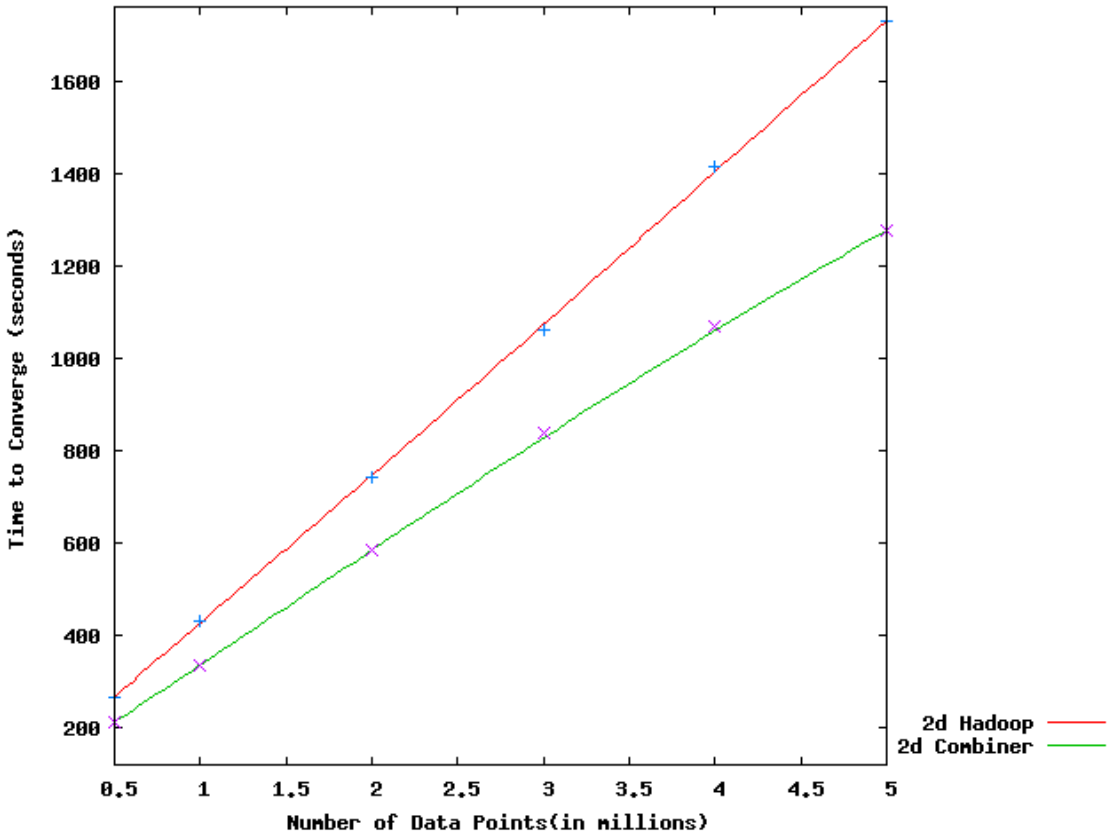


Figure 5.9: Comparative Plot: Hadoop MapReduce vs MapReduce with Combiner.

Table 5.3: Time Taken (seconds) for One Iteration of MapReduce with Combiner

Data Points (in Million)	1	2	3	4	5
Average Map Time (in seconds)	14	26	38	50	63
Shortest Map Time	2	2	4	4	5
Shuffle Task Time	49	98	129	164	182
Average Reducer Time	2	1	1	2	1
Total Time for a MapReduce Cycle	63	114	166	209	257
Number of Spilled Records	10	10	10	10	10
Number of Killed Tasks	0	0	1	1	1
Time Taken to Converge (in seconds)	345	588	831	1062	1270

The algorithms for the Mapper with Combiner and the new Reducer are given below:

Algorithm 7 Algorithm for Mapper with Combiner

Require:

- A subset of d-dimensional objects of $\{x_1, x_2, \ldots, x_n\}$ in each mapper
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$

Output: a list of centroids, new local centroids; written locally and read by the reducer.

$M_1 \Leftarrow \{x_1, x_2, \ldots, x_m\}$
$current\_centroids \Leftarrow C$
$distance\,(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$ where p_i (or q_i) is the coordinate of p (or q) in dimension i
$outputlist \Leftarrow null$
$v \Leftarrow \{\}$
for all $x_i \in M_1$ such that $1 \leq i \leq m$ **do**
    $bestCentroid \Leftarrow null$
    $minDist \Leftarrow \infty$
    for all c in $current\_centroids$ **do**
        $dist \Leftarrow distance(x_i, c)$
        if $bestCentroid = null \;||\; dist < minDist$ **then**
            $bestCentroid \Leftarrow c$
            $minDist \Leftarrow dist$
        end if
    end for
    $v[bestCentroid] \Leftarrow (x_i)$
    i +=1
end for
for all $centroid \in v$ **do**
    $localCentroid, sumofLocalobjects, numofLocalObjects \Leftarrow null$
    for all $object \in v[centroid]$ **do**
        $sumofLocalObjects$ += $object$
        $numofLocalObjects$ += 1
    end for
    $localCentroid \Leftarrow (sumofLocalObjects \div numofLocalObjects)$
    $outputlist$ << $(centroid, localCentroid)$
end for
return outputlist

Algorithm 8 Algorithm for Reducer with Combiner

Require: Input: (key, value) where key =oldCentroid and value = newLocalCentroid assigned to the centroid by the mapper. Output: (key, value) where key = oldcentroid and value = newBestCentroid which is the new centroid value calculated for that bestCentroid.

1: $outputlists$ from mappers
2: $v \Leftarrow \{\}$
3: $newCentroidList \Leftarrow null$
4: **for all** $y \in outputlist$ **do**
5: $centroid \Leftarrow y.key$
6: $localCentroid \Leftarrow y.value$
7: $v[centroid] \Leftarrow localCentroid$
8: **end for**
9: **for all** $centroid \in v$ **do**
10: $newCentroid, sumofLocalCentroids, numofLocalCentroids$
11: $newCentroid \Leftarrow (sumofLocalCentroids \div numofLocalCentroids)$
12: $newCentroidList << (newCentroid)$
13: **end for**
14: **return** $newCentroidList$

Adding the combiner reduced the time taken to shuffle and the time at the reducer. It reduced the number of spilled records and improved the overall time taken for a cycle.

The graph below shows a comparative plot of the time taken for up to 12 million points using Hadoop Combiner and sequential clustering. We noted that Hadoop MapReduce performed better than sequential clustering from 8 million data points about 60MB size. By 12 million points and 90MB size, we noticed that Hadoop completed clustering 150 seconds ahead of sequential clustering.

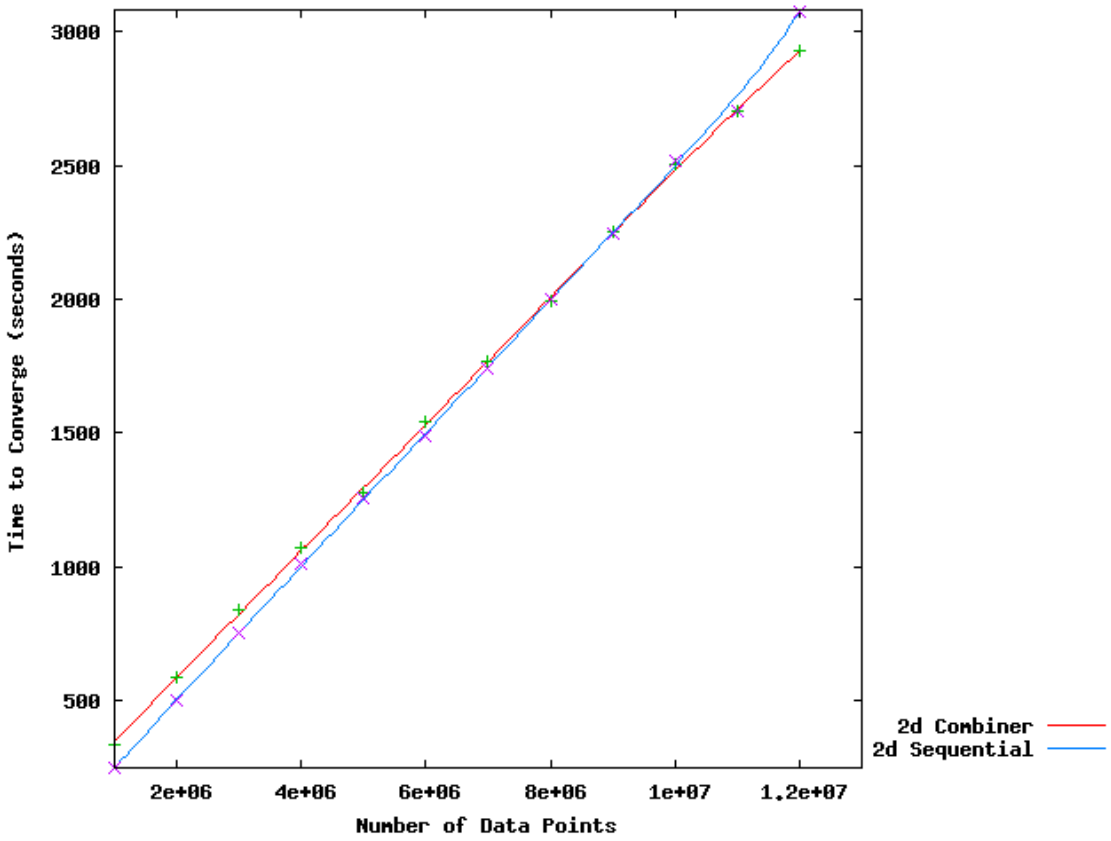


Figure 5.10: Comparative Plot: Hadoop MapReduce with Combiner vs Sequential Clustering.

5.5 Experiment C: Effect of Number of Nodes

While tuning the performance of Hadoop improved the clustering time, we also had to study the effect of the number of nodes on the convergence time. So we used the same set of random points and performed MapReduce with just one node. In such a case the entire data is mapped and reduced on the same node. We noted the time to converge and increased the number of nodes to two and performed the same experiment. We studied the effect up to four nodes. The time taken to converge is shown below:

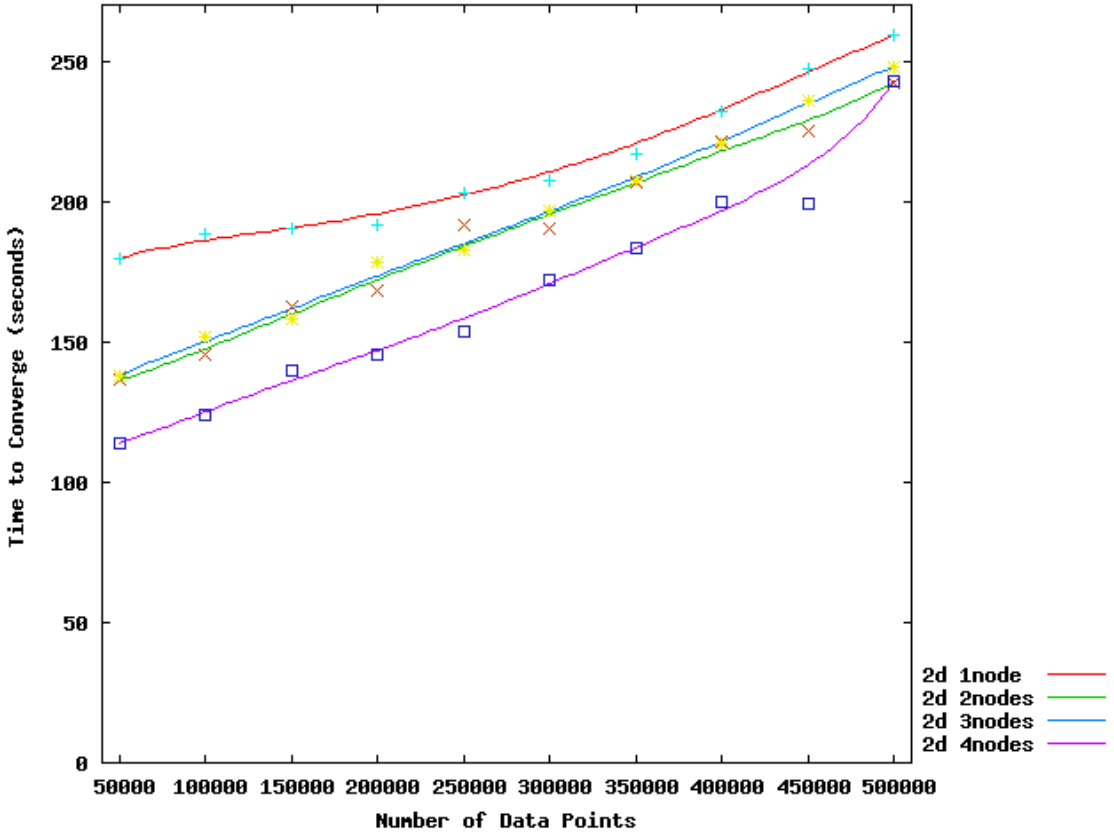


Figure 5.11: Convergence Plot of 2d Points for Multiple Nodes.

As we increased the number of nodes, there was a marked performance benefit and convergence was faster. For 100000 3d-data points, with one node in action, it took about 110 seconds to converge; with 4 nodes, it took only 71 seconds. We noted up to 35 percent improvement in convergence speed. Therefore, increasing the number of nodes and distributing the data improves Hadoop MapReduce performance.

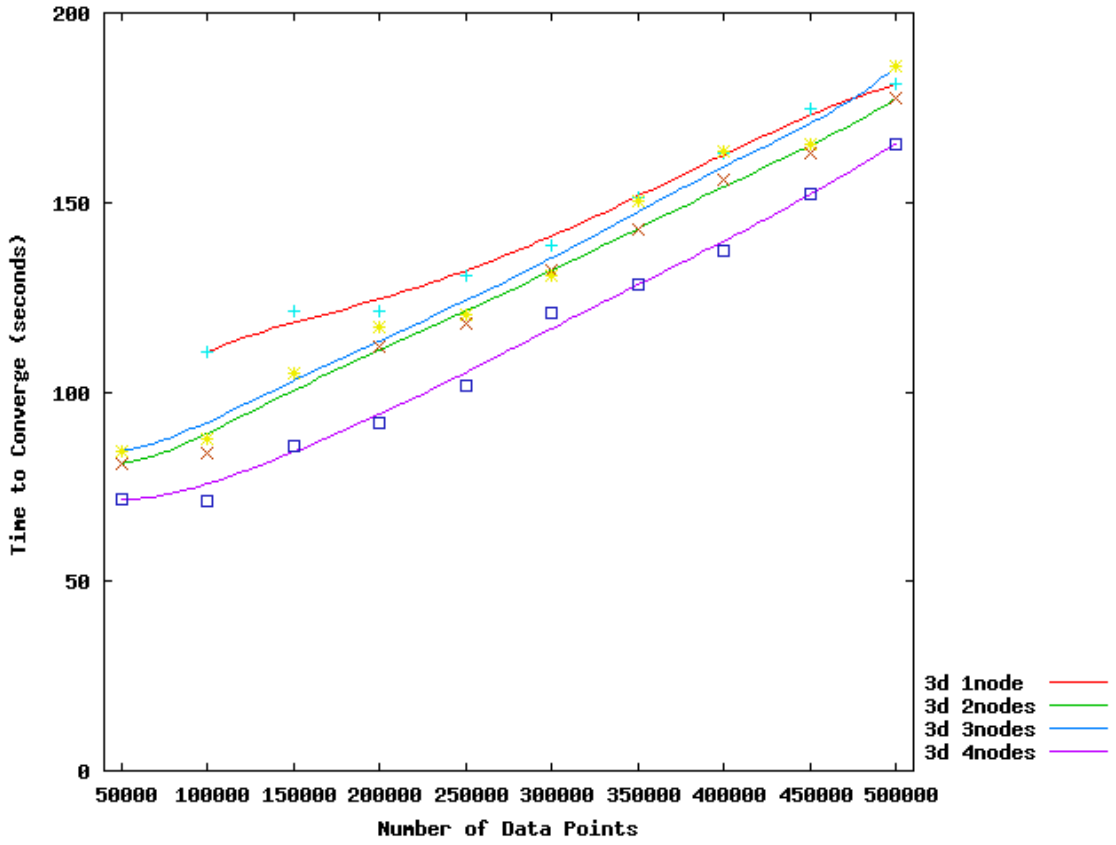


Figure 5.12: Convergence Plot of 3d Points for Multiple Nodes.

5.6 Complexity Issues

In this section, we will show and prove theoretically the experimental results obtained.

5.6.1 Complexity of the Algorithms

Complexity of the SequentialAalgorithm:

SimpleAssignment: In sequential clustering, we iterate through the dataset of m points for each centroid point in the k clusters, calculating the distance between the data point and each centroid. So, the complexity of SimpleAssignment is:

$$SA = k * m \tag{5.1}$$

CentroidCalculator: When the data points are assigned to a centroid, we iterate through each centroid point. We cumulate and count the data points assigned to each centroid and calculate the new centroid.

$$CC = m \tag{5.2}$$

We repeat the SimpleAssignment and CentroidCalculation for n number of iterations until convergence. Therefore, the total time complexity of the sequential algorithm according to the equations 5.1 and 5.2 is as follows:

$$SQ = n(km + m) = nm(k + 1) \tag{5.3}$$

where k is the number of clusters, m is the number of data points and n is the number of iterations.

Complexity of MapReduce:

Mapper: While using MapReduce for clustering, we split the dataset between x available mappers. In each mapper, we iterate through m/x data points and calculate the minimum distance for each centroid in the k clusters. So, the complexity of the mapper is expressed as follows:

$$CM = \frac{k * m}{x} \tag{5.4}$$

Reducer: In the reducer, we iterate through the output from x mappers assigning each point m/x and its associated centroid into a dictionary type variable for further processing. We then, iterate through each centroid k, cumulating and counting the data points m/k assigned in that cluster and calculate a new centroid.

$$CR = \frac{xm}{x} k \frac{m}{k} = m^2 \tag{5.5}$$

We repeat the MapReduce process n times till convergence. So, according to the equations 5.4 and 5.5 the total complexity time for the MapReduce algorithm is:

$$MR = n(\frac{km}{x} + m^2) \tag{5.6}$$

where k is the number of clusters, m is the number of data points, n is the number of iterations and x is the number of nodes assigned for MapReduce (n is far less than m).

Complexity of MapReduce with Combiner:

Mapper: In the mapper, we iterate through all the m/x data points for each centroid in the k cluster to calculate the minimum distance. Therefore, the complexity now is as follows:

$$CMC = \frac{km}{x} \tag{5.7}$$

In the combiner, we iterate through m/x data points and their centroids calculating a local centroid. Thus, the combiner's complexity is:

$$CCC = \frac{m}{x} \tag{5.8}$$

Reducer: In the reducer, we iterate through k outputs from x mappers to calculate the global centroid:

$$CRC = kx \tag{5.9}$$

We repeat the entire process for n iterations till convergence. Thus the total complexity of the MapReduce with Combiner algorithm according to the equations 5.7, 5.8, and 5.9 is as follows:

$$MRC = n(\frac{km}{x} + \frac{m}{x} + kx) \tag{5.10}$$

where k is the number of clusters, m is the number of data points, n is the number of iterations and x is the number of nodes assigned for MapReduce. Once again, n is far less than m.

5.6.2 Comparison of the Complexity of the Algorithms

In this section, we will compare the complexities of the three algorithms we have used and prove their ranking according to their complexity.

Sequential vs. MapReduce:

According to the equations 5.3, and 5.6, we compare the sequential and the MapReduce algorithm:

$$\frac{SQ}{MR} = \frac{mn(k+1)}{n(\frac{km}{x} + m^2)} = \frac{m(k+1)}{m(\frac{k}{x} + m)} = \frac{k+1}{\frac{k}{x} + m} < 1 \tag{5.11}$$

since $m >>> k+1$.

Therefore, the sequential algorithm outperforms the simple MapReduce algorithm as was demonstrated with the experimental results as well.

MapReduce vs. MapReduce with Combiner:

From equations 5.6 and 5.10 we can see that:

$$\frac{MR}{MRC} = \frac{n(\frac{km}{x} + m^2)}{n(\frac{km}{x} + \frac{m}{x} + kx)} = \frac{km + xm^2}{km + m + kx^2} > 1 \tag{5.12}$$

since $km + xm^2 >>> km + m + kx^2$.

This proves the experimental results where the MapReduce with the Combiner performs better than the simple MapReduce algorithm.

MapReduce with Combiner vs. Sequential:

Finally, from the equations 5.10 and 5.3, comparing the sequential and the MapReduce with Combiner algorithms, we have the following:

$$\frac{MRC}{SQ} = \frac{n(\frac{km}{x} + \frac{m}{x} + kx)}{mn(k+1)} = \frac{\frac{m(k+1)}{x}}{m(k+1)} + \frac{kx}{m(k+1)} = \frac{1}{x} + \frac{kx}{m(k+1)} < 1 \tag{5.13}$$

This equation could be less than 1 if $x > 1$, since $m >>> kx$

Both simple MapReduce and MapReduce with Combiner have a set up overload. It is then proved both theoretically and experimentally that the MapReduce with Combiner algorithm can outperform the sequential algorithm which in turn outperforms the simple MapReduce one.

In addition, the experimental results show that the MapReduce with Combiner scales efficiently. This can also be proved theoretically based on the equation 5.10. It is clear that the algorithm speeds up when the number of processing nodes x increases.

Chapter 6: **Conclusion**

In this study, we have applied MapReduce technique to k-means clustering algorithm and clustered over 10 million data points up to 109.8MB in size. We compared the results with sequential k-means clustering. We have shown that k-means algorithm can be successfully parallelised and clustered on commodity hardware. MapReduce can be used for k-means clustering. The results also show that the clusters formed using MapReduce is identical to the clusters formed using sequential algorithm.

Our experience also shows that the overheads required for MapReduce algorithm and the intermediate read and write between the mapper and reducer jobs makes it unsuitable for smaller datasets. However, adding a combiner between Map and Reduce jobs improves performance by decreasing the amount of intermediate read/write. We also noted that the number of nodes available for map tasks affect performance and more the number of nodes, the better is the performance. Therefore, we believe, that MapReduce will be a valuable tool for clustering larger datasets that are distributed and cannot be stored on a single node.

References

[1] Hadoop Page on Mahout. `http://mahout.apache.org/`, 2011.

[2] Hadoop Page on Disco. `http://discoproject.org/`, 2011.

[3] Hadoop Page on Pig. `http://pig.apache.org/`, 2011.

[4] Hadoop Page on Hive. `http://hive.apache.org/`, 2011.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] J. Ekanayake, T. Gunarathne, G. Fox, A.S. Balkir, C. Poulain, N. Araujo, and R. Barga. Dryadlinq for scientific analyses. In *2009 Fifth IEEE International Conference on e-Science*, pages 329–336. IEEE, 2009.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[8] D. Gillick, A. Faria, and J. DeNero. Mapreduce: Distributed computing for machine learning, 2006.

[9] S. Guha, R. Rastogi, and K. Shim. Cure: an efficient clustering algorithm for large databases. In *ACM SIGMOD Record*, volume 27, pages 73–84. ACM, 1998.

[10] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi. Evaluating mapreduce on virtual machines: The hadoop case. *Cloud Computing*, pages 519–528, 2009.

[11] W. Jiang, V.T. Ravi, and G. Agrawal. Comparing map-reduce and freeride for data-intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[12] R. Jin, A. Goswami, and G. Agrawal. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems*, 10(1):17–40, 2006.

[13] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. *Cloud Computing*, pages 674–679, 2009.

Appendix: Complete Python Code

Program Structure

Python Code for Mapper Reducer

StartUp Program

```
#!/usr/bin/env python
import sys, math, random
import os
import shutil
import time
# starting point for the system
#This program generates the required number of data points and calls convergence
#check and mapreduce.

def main(args):
    DP = open("datapoints.txt","w").close()
    num_points=0
    pointx = ''

    while num_points <10000000:
        num_iter =0
        num_points +=1000000
       num_gen = 1000000
        # Create num_points random Points in n-dimensional space
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                          int(num_points*.5), 2,1, 900, 1,900
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                         int(num_points*.1), 2,50, 350, 50,500
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                         int(num_points*.1), 2,410, 600,10,550
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                      int(num_points*.1), 2,600, 890, 600,900
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                        int(num_points*.1), 2,100, 500,650,900
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
        num_gen, coords, lowerx, upperx, lowery, uppery =
                                         int(num_points*.1), 2,650, 880,50,470
        pointx = makeRandomPoint(num_gen, lowerx, upperx, lowery, uppery)
```

```
while num_iter <5:
  num_iter + =1
  filename = "centroid2d_" + ‘num_iter‘ +".txt"
  shutil.copy(filename,"centroidinput.txt")
  datafilename = "datapoints" + ‘num_points‘ + ".txt"
  shutil.copy(datafilename,"datapoints.txt")
  kmeans.main(num_points)
  # final clustering of the datapoints
  os.system("python mapperfinal.py")
  #Renaming and finalising of files
  newfilename = "statistics_" + ‘num_points‘ +".txt"
  os.rename("statistics.txt", newfilename)
  newname = "datapoints" + ‘num_points‘ +".txt"
  shutil.copy("datapoints.txt", newname)
  newname = "cluster1_" + ‘num_points‘ +".txt"
  os.rename("cluster1.txt", newname)
  newname = "cluster2_" + ‘num_points‘ +".txt"
  os.rename("cluster2.txt", newname)
  newname = "cluster3_" + ‘num_points‘ +".txt"
  os.rename("cluster3.txt", newname)
  newname = "cluster4_" + ‘num_points‘ +".txt"
  os.rename("cluster4.txt", newname)
  newname = "cluster5_" + ‘num_points‘ +".txt"
  os.rename("cluster5.txt", newname)
  newname = "cluster6_" + ‘num_points‘ +".txt"
  os.rename("cluster6.txt", newname)


#    DP = open("datapoints.txt","w").close()
num_points=0
pointx = ''
while num_points <10000000:
   num_points +=1000000
   num_gen = 1000000
   num_iter = 0
#        # Create num_points random Points in n-dimensional space
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                        int(num_points*.4), 2,1, 900, 1,900,1,900
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery, ppery,lower,upper)
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                      int(num_points*.1), 2,50, 300, 50,450,50,600
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                   int(num_points*.1), 2,410, 600,10,550,100,200
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                 int(num_points*.1), 2,600, 890, 0,200,10,300
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)

   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
```

```
                                    int(num_points*.1), 2,100, 500,650,900,600,700
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                     int(num_points*.1), 2,650, 880,50,470,800,900
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)
   num_gen, coords, lowerx, upperx, lowery, uppery,lower,upper =
                                    int(num_points*.1), 2,800, 900,750,900,800,900
   pointx =
   makeRandom3dPoint(num_gen, lowerx, upperx, lowery,uppery,lower,upper)
while num_iter <5:
   num_iter +=1
   print num_iter
   filename = "centroid3d_" + `num_iter` +".txt"
   shutil.copy(filename,"centroidinput.txt")
   datafilename = "datapoints3d" + `num_points` + ".txt"
   shutil.copy("datapoints.txt", datafilename)
   kmeans3d.main(num_points)
   # final clustering of the datapoints
   #Renaming and finalising of files
   newfilename = "statistics3d_" + `num_points` +"_"+ `num_iter` +".txt"
   os.rename("statistics3d.txt", newfilename)
   newname = "cluster3d1_" + `num_points` +".txt"
   os.rename("cluster3d1.txt", newname)
   newname = "cluster3d2_" + `num_points` +".txt"
   os.rename("cluster3d2.txt", newname)
   newname = "cluster3d3_" + `num_points` +".txt"
   os.rename("cluster3d3.txt", newname)
   newname = "cluster3d4_" + `num_points` +".txt"
   os.rename("cluster3d4.txt", newname)
   newname = "cluster3d5_" + `num_points` +".txt"
   os.rename("cluster3d5.txt", newname)
   newname = "cluster3d6_" + `num_points` +".txt"
   os.rename("cluster3d6.txt", newname)

def makeRandomPoint(num_points, lowerx, upperx, lowery, uppery):
    datapoints = ''
    coordX = ''
    coordY = ''

  for i in range(num_points):
    coordX=random.randint(lowerx, upperx)
    coordY=random.randint(lowery, uppery)
    datapoints +=`coordX`+','+`coordY`
    datapoints += ' '
    DP = open("datapoints.txt","a")
    # Write all the lines for datapoints at once:
    DP.writelines(datapoints)
    DP.close()

    return datapoints

def makeRandom3dPoint(num_points, lowerx, upperx, lowery, uppery,lowerz,upperz):
```

```
    datapoints = ''
    coordX = ''
    coordY = ''
    coordZ = ''
    for i in range(num_points):
      coordX=random.randint(lowerx, upperx)
      coordY=random.randint(lowery, uppery)
      coordZ=random.randint(lowerz, upperz)
      datapoints +=`coordX`+','+`coordY`+','+`coordZ`
      datapoints += ' '
      DP = open("datapoints.txt","w")
      # Write all the lines for datapoints at once:
      DP.writelines(datapoints)
      DP.close()

    return datapoints

if __name__ == "__main__": main(sys.argv)
```

StartUp Program: Call to MapReduce

```
#!/usr/bin/env python
import sys, math, random
import os
import shutil
import time

# First step into the algorithm.
# checks convergence and calls mapReduce iteratively

def main(args):
  # maximum delta is set to 2.
  maxDelta = 3
  oldCentroid = ''
  currentCentroid = ''
  mrtime = 0
  num_iter = 1
  statistics =''
  statplot =''
  # copies the  generated datapoints file and seed centroid file
  #from local folder to hdfs and starts mapReduce
  os.system('bin/hadoop dfs -put ~/hadoop/datapoints.txt datapoints.txt')
  statp = open("stat_plot.txt","a")
  STAT = open("statistics.txt","w")
  start = time.time()
  while maxDelta >2:
    #print num_iter
    # check for delta
    cfile = open("centroidinput.txt", "r")
    currentCentroid = cfile.readline()
    cfile.close()
```

```
if oldCentroid != '':
  maxDelta = 0
  #   remove leading and trailing whitespace
  oldCentroid = oldCentroid.strip()
       # split the centroid into centroids
  oldCentroids = oldCentroid.split()
  #   remove leading and trailing whitespace
  currentCentroid = currentCentroid.strip()
       # split the centroid into centroids
  currentCentroids = currentCentroid.split()
     # centroids are not in the same order in each iteration.
     #So each centroid is checked against all other centroids for distance.

   for value in currentCentroids:
    dist = 0
    minDist = 9999
    oSplit = value.split(',')

    for c in oldCentroids:
    # split each co-ordinate of the oldCentroid and the currentCentroid
      cSplit = c.split(',')
    # To handle non-numeric value in old or new centroids
      try:
        dist = (((int(cSplit[0]) - int(oSplit[0]))**2) +
                ((int(cSplit[1]) - int(oSplit[1]))**2))**.5
        if dist < minDist:
          minDist = dist
         #print minDist, value, c
        except ValueError:
           pass
    if minDist > maxDelta:
      maxDelta = minDist
  else:
    statistics += '\n seed centroid: ' +`currentCentroid`
    statistics += '\n num_iteration: ' +`num_iter`+';  Delta: '+`maxDelta`
    #checks the new delta value to avoid additional mapreduce iteration

  if maxDelta > 2:
    os.system('bin/hadoop dfs -put
            ~/hadoop/centroidinput.txt centroidinput.txt')
    mrstart = time.time()
    os.system('bin/hadoop jar
           ~/hadoop/mapred/contrib/streaming/hadoop-0.21.0-streaming.jar
           -D mapred.map.tasks=4 -file ~/hadoop/mapper1.py -mapper
           ~/hadoop/mapper1.py -file ~/hadoop/reducer1.py -reducer
           ~/hadoop/reducer1.py -input datapoints.txt -file centroidinput.txt
           -file ~/hadoop/defdict.py -output data-output')
    mrend = time.time()
    mrtime += mrend -mrstart
    #old_centroid is filled in for future delta calculation
    cfile = open("centroidinput.txt", "r")
    oldCentroid = cfile.readline()
    cfile.close()
    # output is copied to local files for later lookup
```

```
    #and the hdfs version is deleted for next iteration.
    os.system('bin/hadoop dfs
                    -copyToLocal /user/grace/data-output ~/hadoop/output')
    os.rename("output/part-00000", "centroidinput.txt")
    shutil.rmtree('output')
       num_iter += 1
    os.system('bin/hadoop dfs -rmr data-output')
    os.system('bin/hadoop dfs -rmr centroidinput.txt')
    end = time.time()
    elapsed = end -start
    print "elapsed time ", elapsed, "seconds"
    statistics += '\n  Time_elapsed: '+ `elapsed`
    statistics += '\n New Centroids: '+ `currentCentroid`

     # Write all the lines for statistics at once:
     STAT.writelines(statistics)
     STAT.close()
     # Write all the lines for statplot incrementaly:
     statplot = `args` + '  '+ `mrtime` + '  '   +`num_iter` + '  '+ `elapsed` +'\n'
     statp.writelines(statplot)
     os.system('bin/hadoop dfs -rmr datapoints.txt')

if __name__ == "__main__": main(sys.argv[1])
```

Mapper Program

```
#!/usr/bin/env python
import sys, math, random
# mapper for 2d points
class Point:
    # Instance variables
    # self.coords is a list of coordinates for this Point
    # self.n is the number of dimensions this Point lives in (ie, its space)
    # self.reference is an object bound to this Point
    # Initialise new Points
    def __init__(self, coords, reference=None):
        self.coords = coords
        self.n = len(coords)
        self.reference = reference
    # Return a string representation of this Point

    def __repr__(self):
        return str(self.coords)

def main(args):
  # variable to hold map output
  outputmap = ''
  #centroid details are stored in an input file
  #cfile = " ~/hadoop/centroidinput.txt"
  cfile = "centroidinput.txt"
  infilecen = open(cfile,"r")
```

```
  centroid = infilecen.readline()
  #print centroid

  for point in sys.stdin:
   # remove leading and trailing whitespace
    point = point.strip()
    # split the line into words
    points = point.split()
    #yield point.split()
    # remove leading and trailing whitespace
    centroid = centroid.strip()
    # split the centroid into centroids
    centroids = centroid.split()
   #outfile = open("mapoutput1.txt","w")
   # increase counters

   for value in points:
    dist = 0
    minDist = 999999
    bestCent = 0

    for c in centroids:
     # split each co-ordinate of the centroid and the point
     cSplit = c.split(',')
     vSplit = value.split(',')
    # To handle non-numeric value in centroid or input points
     try:
       #dist = abs(int(cSplit[0]) - int(vSplit[0]))
       dist = (((int(cSplit[0]) - int(vSplit[0]))**2) +
                ((int(cSplit[1]) - int(vSplit[1]))**2))**.5
      #print dist
      if dist < minDist:
       minDist = dist
       bestCent = c
      except ValueError:
       pass
       #outfile.writelines(outputmap)
       print '%s\t%s' % (bestCent, value)
      # outfile = open("mapoutput1.txt","w")
      # outfile.writelines(outputmap)
      # outfile.close()

def makeCentroids():
  cfile = open("centroidinput.txt", "r")
  seed = cfile.readline()
  cfile.close()
 #seed = "43,211 130,62 387,253 454,734 545,920"
  FILE1 = open("centroidpoints.txt","w")
  # Write all the lines for seed = "15,40 110,10 25,9 69,678 240,78"
  FILE1.writelines(seed)
  FILE1.close()
  return seed
if __name__ == "__main__": main(sys.argv)
```

Reducer Program

```
#!/usr/bin/env python

from operator import itemgetter
#from collections import defaultdict
from defdict import *
import sys

# reducer for 2d points
def main(args):
  point2centroid = defaultdict(list)
  # input comes from STDIN

  for line in sys.stdin:
     # remove leading and trailing whitespace
    line = line.strip()
  # parse the input we got from mapper.py into a dictionary
    centroid, point = line.split('\t')
    point2centroid[centroid].append(point)
    #print point2centroid.items()
    pointX= pointY =0
    newCentroid= oldCentroid=''

    for centroid in point2centroid:
     sumX= sumY= count= newX=newY =0
     oldCentroid  += centroid
     oldCentroid += ' '
     for point in point2centroid[centroid]:
       pointX, pointY = point.split(',')
       sumX += int(pointX)
       sumY +=int(pointY)
       count +=1
    newX=sumX/count
    newY=sumY/count
    newCentroid +=`newX`+','+`newY`
    newCentroid+= ' '
    print newCentroid

if __name__ == "__main__": main(sys.argv)
```

WrapUp Programs

```
#!/usr/bin/env python
import sys, math, random
import clustering
import removingoutliers
#from collections import defaultdict
from defdict import *
```

```
# this program performs the final data point assignment
#and calls removingoutliers.py  to remove outliers

def main(args):
  point2centroid = defaultdict(list)
  # variable to hold map output
  infiledat = open("datapoints.txt", "r")
  datap = infiledat.readline()
  dpoints = datap.split()
  cfile = "centroidinput.txt"
  infilecen = open(cfile,"r")
  centroid = infilecen.readline()
  #print centroid

  for point in dpoints:
   #remove leading and trailing whitespace
    point = point.strip()
    #split the line into words
    points = point.split()
    #remove leading and trailing whitespace
    centroid = centroid.strip()
    #split the centroid into centroids
    centroids = centroid.split()
    #increase counters

    for value in points:
      dist = 0
      minDist = 999999
      bestCent = 0

      for c in centroids:
        #print c
        # split each co-ordinate of the centroid and the point
        cSplit = c.split(',')
        vSplit = value.split(',')
        # To handle non-numeric value in centroid or input points
        try:
           dist = (((int(cSplit[0]) - int(vSplit[0]))**2) +
              ((int(cSplit[1]) - int(vSplit[1]))**2))**.5
          #print dist
           if dist < minDist:
              minDist = dist
              bestCent = c
          except ValueError:
             pass
    #print '%s\t%s' % (bestCent, value)
     point2centroid[bestCent].append(value)
 #clustering.main(point2centroid)
 removingoutliers.main(point2centroid)
 #print point2centroid

if __name__ == "__main__": main(sys.argv)
```

WrapUp Program: Removing Outliers

```
#!/usr/bin/env python

from operator import itemgetter
#from collections import defaultdict
from defdict import *
import sys
import clustering
from statlib import stats

# this program removes outliers based on abs median value
#and calls clustering.py to cluster the datapoints
# Finds the final adjusted centroid
def main(args):
  point2centroid = defaultdict(list)
  point2centroidfinal = defaultdict(list)
  point2centroidreject = defaultdict(list)
  point2centroid = args

  for centroid in point2centroid:
   # print 'centroid',centroid
    centroidX, centroidY = centroid.split(',')
    distlist=finallist=pdevlist=rejectlist=[]
    newX =newY=sumX=sumY=count=ctrrej=0

    for point in point2centroid[centroid]:
        #print point
        pointX, pointY = point.split(',')
        #datapoint += pointX + '  ' + pointY +'\n'
        dist = (((int(centroidX) - int(pointX))**2) +
                                    ((int(centroidY) - int(pointY))**2))**.5
        distlist.append(dist)

    pmed = stats.medianscore(distlist)
    #print 'pmed',pmed
    for p in distlist:
      pdev = abs(p-pmed)
      pdevlist.append(pdev)
    med_pdev = stats.medianscore(pdevlist)
    #print "med_pdev", med_pdev

    for p in distlist:
        if med_pdev >0:
            test_stat = abs(p-pmed)/med_pdev
            #print test_stat
            if test_stat<2:
               finallist.append(p)
            else:
                ctrrej +=1
                rejectlist.append(p)

    for point in point2centroid[centroid]:
```

```
        #print point
        pointX, pointY = point.split(',')
        dist = (((int(centroidX) - int(pointX))**2) +
                                    ((int(centroidY) - int(pointY))**2))**.5
        if dist in rejectlist and dist >100:
            point2centroidreject[centroid].append(point)
        else:
            point2centroidfinal[centroid].append(point)
            sumX += int(pointX)
            sumY +=int(pointY)
            count +=1
    newX=sumX/count
    newY=sumY/count
    newCentroid = `newX`+','+`newY`
    #print newCentroid
    point2centroidfinal[newCentroid]=point2centroidfinal[centroid]
    del point2centroidfinal[centroid]
  clustering.main(point2centroidfinal)

if __name__ == "__main__": main(sys.argv)
```

WrapUp Program: Final Clustering

```
#!/usr/bin/env python

from operator import itemgetter
#from collections import defaultdict
from defdict import *
import sys

# this program performs the final cluster and creates files for each cluster
def main(args):
  point2centroid = defaultdict(list)
  point2centroid = args
  datapoint = ''
  centroidpoint=''
  iter = 0

  for centroid in point2centroid:
    #print centroid
    iter +=1
    filename = "cluster"+`iter`+'.txt'
    cfile = open(filename,'w')
    datapoint = ''
    centroidX, centroidY = centroid.split(',')
    centroidpoint += centroidX + '  ' +centroidY +'\n'

    for point in point2centroid[centroid]:
        #print point
```

```
        pointX, pointY = point.split(',')
        datapoint += pointX + '  ' + pointY +'\n'
    cfile.writelines(datapoint)
    cfile.close()
  filename="cluster6.txt"
  #print filename
  cfile= open(filename, 'w')
  cfile.writelines(centroidpoint)
  cfile.close()

if __name__ == "__main__": main(sys.argv)
```

Python Code for MapReduce with Combiner

Mapper with Combiner Program

```
#!/usr/bin/env python
import sys, math, random
from defdict import *
# mapper with combiner for 2d points
def main(args):
  # variable to hold map output
  outputmap = ''
  point2centroid = defaultdict(list)
  cfile = "centroidinput.txt"
  infilecen = open(cfile,"r")
  centroid = infilecen.readline()
  #print centroid

  for point in sys.stdin:
    point = point.strip()
    # split the line into words
    points = point.split()
    centroid = centroid.strip()
    # split the centroid into centroids
    centroids = centroid.split()

for value in points:
  dist = 0
  minDist = 999999
  bestCent = 0

  for c in centroids:
    # split each co-ordinate of the centroid and the point
    cSplit = c.split(',')
    vSplit = value.split(',')
    # To handle non-numeric value in centroid or input points
    try:
      #dist = abs(int(cSplit[0]) - int(vSplit[0]))
```

```
      dist = (((int(cSplit[0]) - int(vSplit[0]))**2) +
                    ((int(cSplit[1]) - int(vSplit[1]))**2))**.5
      #print dist
      if dist < minDist:
        minDist = dist
        bestCent = c
    except ValueError:
      pass
    #print '%s\t%s' % (bestCent, value)
point2centroid[bestCent].append(value)
pointX= pointY =0
 # print point2centroid
newCentroid= oldCentroid=''

for centroid in point2centroid:
        sumX= sumY= count= newX=newY =0

        for point in point2centroid[centroid]:
          pointX, pointY = point.split(',')
          sumX += int(pointX)
          sumY +=int(pointY)
          count +=1
        newX=sumX/count
        newY=sumY/count
        newCentroid =`newX`+','+`newY`
        print '%s\t%s' % (centroid, newCentroid)

if __name__ == "__main__": main(sys.argv)
```

Reducer Program

```
#!/usr/bin/env python

from operator import itemgetter
#from collections import defaultdict
from defdict import *
import sys

# reducer wiht combiner for 2d points
def main(args):
  point2centroid = defaultdict(list)
  # input comes from STDIN

  for line in sys.stdin:
     # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py into a dictionary
    oldCentroid, tempCentroid = line.split('\t')
    point2centroid[oldCentroid].append(tempCentroid)
    #print point2centroid.items()
    pointX= pointY =0
```

```
  newCentroid= ''

  for centroid in point2centroid:
    sumX= sumY= count= newX=newY =0

    for point in point2centroid[centroid]:
     pointX, pointY = point.split(',')
     sumX += int(pointX)
     sumY +=int(pointY)
     count +=1
    newX=sumX/count
    newY=sumY/count
    newCentroid +=`newX`+','+`newY`
    newCentroid+= ' '
    print newCentroid

if __name__ == "__main__": main(sys.argv)
```

Python Code for Sequential Clustering

Simple Assignment Program

```
#!/usr/bin/env python

from operator import itemgetter
import sys, math, random
from defdict import *
import reducer1simple
# simple assignment of points to centroids without hadoop

def main(args):
   point2centroid = defaultdict(list)
   infiledat = open("datapoints.txt", "r")
  datap = infiledat.readline()
  points = datap.split()
  cfile = "centroidinput.txt"
  infilecen = open(cfile,"r")
  centroid = infilecen.readline()
  #for point in sys.stdin:
  for point in points:
    # remove leading and trailing whitespace
    point = point.strip()
    # split the line into words
    points = point.split()
    #remove leading and trailing whitespace
    centroid = centroid.strip()
    # split the centroid into centroids
    centroids = centroid.split()
        # increase counters
```

```
for value in points:
  dist = 0
  minDist = 999999
  bestCent = 0

  for c in centroids:
    # split each co-ordinate of the centroid and the point
    cSplit = c.split(',')
    vSplit = value.split(',')
    # To handle non-numeric value in centroid or input points
    try:
      #dist = abs(int(cSplit[0]) - int(vSplit[0]))
      dist = (((int(cSplit[0]) - int(vSplit[0]))**2) + ((int(cSplit[1]) -
            int(vSplit[1]))**2))**.5
      #print dist
      if dist < minDist:
        minDist = dist
        bestCent = c
    except ValueError:
      pass
     point2centroid[bestCent].append(point)
     reducer1simple.main(point2centroid)

if __name__ == "__main__": main(sys.argv)
```

Centroid Calculator Program

```
#!/usr/bin/env python

from operator import itemgetter
#from collections import defaultdict
from defdict import *
import sys

# centroid calculator program without hadoop
def main(args):
  point2centroid = defaultdict(list)
  point2centroid = args
  pointX =0
  pointY =0
  newCentroid=''
  oldCentroid=''

  for centroid in point2centroid:
    sumX =0
    sumY=0
    count=0
    newX =0
    newY=0
    oldCentroid  += centroid
```

```
    oldCentroid += ' '
    for point in point2centroid[centroid]:
      pointX, pointY = point.split(',')
      sumX += int(pointX)
      sumY +=int(pointY)
      count +=1
    newX=sumX/count
    newY=sumY/count
    newCentroid +=`newX`+','+`newY`
    newCentroid+= ' '
    cfile= open("centroidinput.txt", 'w')
    cfile.writelines(newCentroid)
    cfile.close()

if __name__ == "__main__": main(sys.argv)
```