Implementation of MapReduce Framework in a Distributed Environment

Hariharan Venugopal, Nikhita Bahuguna

Abstract— MapReduce is a programming model and an associated implementation for processing and generating large data sets. In this paper we implement MapReduce framework using an effective asynchronous message queuing communication that focuses on the computational aspects of any given problem.

We demonstrate the implementation of MapReduce with two major problems which can be solved in a parallel fashion.

K-means Clustering using Voronoi algorithm

Image Search using Histogram algorithm

Our framework optimizes bandwidth of the network as the data is localized. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

Keywords— MapReduce; data locality; inter-machine communication.

I. INTRODUCTION

In the last twenty years, the main focus of the world economy has shifted from the production and trading of industrial goods to the creation and transfer of information. This process was accompanied by fundamental changes in the economical and social systems and led to the term information age.

The essential condition for this development was the ability to store and transfer information electronically, more precisely with the help of computers. The development of the World Wide Web in 1993 has sped up this process. Today nearly every newspaper, book and scientific paper is published electronically via the Internet. In the last five years, even non-textual media like audio and video documents have become widely accessible.

The existence of such a huge information source has led to a new problem, the filtering of unimportant information from the important one. This is commonly known information overload.

In computer science, the field of information retrieval tries to compensate these negative effects. New tools are developed for filtering and processing the data individually, to get only those pieces which are really important for the user. Most of these approaches accomplish this by processing all available documents.

Modern computers provide high processing powers and Moore's Law can still be applied to the progress in development, but the manipulation of multiple gigabytes of data is still not feasible on a single PC. The computation has to be split between several machines.

A distribution of work among a network of computers is more complicated than developing an application which runs on a single computer. The programmer has to find a way to divide the work - a task that might not be straightforward. Afterwards he has to implement different programs and coordinate their communication. The risk for the occurrence of failures and the time for their correction will increase significantly in such a system. Even if all programs contain no functional errors, it is not assured that they can interact properly.

In spite of all this work, there is only a small advantage for further data manipulation tasks. In general it is very likely that many parts of the system will have to be rewritten when the main algorithm changes.

All these problems lead to the idea of encapsulating the mechanisms for splitting the work between several computers in a library. With such a framework, this problem has to be solved only once and can be used for different data processing tasks. When building a system for manipulating huge amounts of data, the programmer only has to provide his own application code and does not have to mind the pitfalls of a distributed system. A very powerful but simple approach for implementing such a framework is provided by the MapReduce concept.

Our paper implements a distributed MapReduce framework using the programming language Python. It provides the application programmer with a simple but flexible interface for executing his programs. At the moment the MapReduce framework intends to provide distributed environment for programs written in sequential manner. The framework is completely written in python thus, making in available in all platforms which allows the user to setup heterogeneous cluster. It utilizes a brokerless message queuing to ensure that the communication time isn’t an overhead for highly data intensive applications. Currently the framework incorporates examples based on k means clustering which iteratively computes the centroid for a given set of pixels and Histogram Algorithm for image processing.

II. PROGRAMMING MODEL

A. Basic Concept

MapReduce is a patented software framework introduced by Google to support distributed computing on large data sets on clusters of computers. In computer science, functional

programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming gave birth to map and reduce functions which inspired this framework.

MapReduce is basically used where large amount of data is involved, hence this framework finds a great deal of application in areas related to cluster and parallel computing. The map function is applied to each logical record in the given input so that a set of intermediate key/value pair can be computed. The reduce function combines the derived data appropriately by using the key which is same for the values generated.

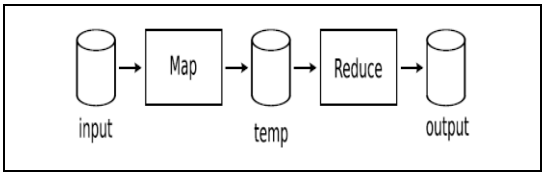

Fig. 1. Basic concept of MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. 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. Many real world tasks are expressible in this model, as shown in the paper. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

"Map" step: The master node takes the input, partitions it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

"Reduce" step: The master node then takes the answers to all the sub-problems and combines them in some way to get the output — the answer to the problem it was originally trying to solve.

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

map (k1,v1) -> list(k2,v2)

reduce (k2,list(v2)) -> list(v2)

i.e, the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

B. Use of Parallelism

The basic map and reduce functions in the previous section do not use parallelism. The idea is to distribute the computation among several computers. A machine which processes the map phase is called a mapper. The output of all mappers is the input for the reduce phase which is computed by the reducers. The MapReduce framework takes care of providing the mappers and reducers with input data and manages the data exchange between the two phases. To be able to do this, the two basic functions need to be extended to be processed in parallel.

Transforming the map function to parallel execution is straightforward. Since the computations of the result elements in the output list are independent from each other, these can be done in parallel. First, the input list has to be split up and distributed to multiple computers in the system. Instead of one single output list, the map phase produces a set of lists. Each single element contains a valid result.

In contrast to the map function, the basic reduce function from listing is not suitable for parallelism. Because every list element is significant for the overall result, splitting the input would result in getting an incorrect overall result. Therefore, some modifications to the basic concept have to be made. The limitation that the reduce phase only produces one single result is not applicable in a parallel environment. It is acceptable that every reducer produces its own result. The result of the reduce phase is now a set of values and it is free to the user to combine these single results or not.

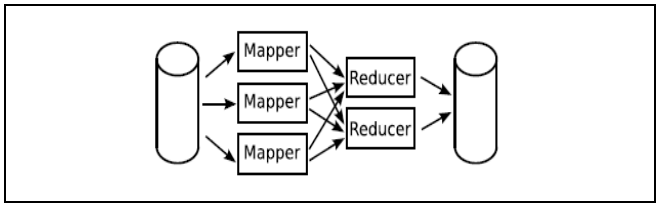

Fig. 2. Use of Parallelism

C. Dataflow

The frozen part of the MapReduce framework is a large distributed sort. The hot spots, which the application defines, are:

- an input reader
- a Map function
- a partition function
- a compare function
- a combiner function
- a Reduce function
- an output writer

Input Reader

The input reader divides the input into appropriate size 'splits' (in practice typically 16MB to 128MB) and the framework assigns one split to each Map function. The input reader reads data from stable storage (typically a distributed file system) and generates key/value pairs. A common example will read a directory full of text files and return each line as a record.

Map Function

Each Map function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other.

If the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and "1" as the value.

Partition Function

Each Map function output is allocated to a particular reducer by the application's partition function for sharding purposes. The partition function is given the key and the number of reducers and returns the index of the desired reduce.

A typical default is to hash the key and modulo the number of reducers. It is important to pick a partition function that gives an approximately uniform distribution of data per shard for load balancing purposes, otherwise the MapReduce operation can be held up waiting for slow reducers to finish.

Between the map and reduce stages, the data is shuffled (parallel-sorted / exchanged between nodes) in order to move the data from the map node that produced it to the shard in which it will be reduced. The shuffle can sometimes take longer than the computation time depending on network bandwidth, CPU speeds, data produced and time taken by map and reduce computations.

Comparison Function

The input for each Reduce is pulled from the machine where the Map ran and sorted using the application's comparison function.

Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user specified Reduce function is commutative and associative.A good example of this is the word counting example. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form <the, 1>. All of these counts will be sent over the network to a single reduce task and then added together by the Reduce function to produce one number. We allow the user to specify an optional Combiner function that does partial merging of this data before it is sent over the network. The Combiner function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Reduce Function

The framework calls the application's Reduce function once for each unique key in the sorted order. The Reduce can iterate through the values that are associated with that key and output 0 or more values.

In the word count example, the Reduce function takes the input values, sums them and generates a single output of the word and the final sum.

Output Writer

The Output Writer writes the output of the Reduce to stable storage, usually a distributed file system.

III. COMMUNICATION ARCHITECTURE

In our proposed framework, we are using a messaging library called ZeroMQ[5] for inter-machine communication.

Zero Message Queue (ZMQ) is a high-performance asynchronous messaging library which allows us to create a new messaging system. It's aimed to use in scalable distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZMQ system can run without a dedicated message broker. The library is designed to have a familiar socket-style API.

The built-in core ZMQ patterns are:

- REQUEST/REPLY - connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern. It is bidirectional, load balanced and state based.
- PUBLISH/SUBSCRIBE - connects a set of publishers to a set of subscribers. This is a data distribution pattern.

- UPSTREAM / DOWNSTREAM - connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.
- PAIR - communication exclusively between peers

Our proposed framework is adaptable to REQUEST/ REPLY, PAIR and PUBLISHER/SUBSCRIBER message patterns which can be implemented as per the nature of the problem.

The REQUEST/ REPLY socket type allows only an alternating sequence of send(request) and subsequent recv(reply) calls. Each request sent is load-balanced among all services, and each reply received is matched with the last issued request.

When a ZMQ::REQ socket enters an exceptional state due to having reached the high water mark for all services, or if there are no services at all, then any send() operations on the socket shall block until the exceptional state ends or at least one service becomes available for sending; messages are not discarded.

- Compatible peer sockets - ZMQ::REP
- Direction - Bidirectional
- Outgoing routing strategy - Load-balanced
- Incoming routing strategy - Last peer
- ZMQ::HWM option action - Block

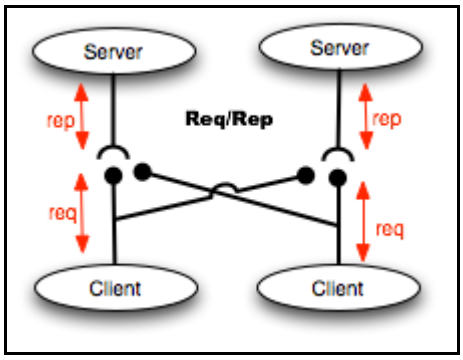

Fig. 3. Request/Reply Pattern

```
import zmq
context = zmq.Context.instance()

sock = context.socket(zmq.REQ)
sock.bind('tcp://*:8080')

sock.send(message)
response = sock.recv()
```

Fig. 4. Code Snippet of Request Message in Master

```
import zmq
context = zmq.Context.instance()

sock = context.socket(zmq.REP)
sock.connect('tcp://localhost:8080')

while True:
    message = sock.recv()
    response = handle_message(message)
    sock.send(response)
```

Fig. 5. Code Snippet of Response Message in Worker

IV. IMPLEMENTATION

The MapReduce framework utilizes cluster architecture of 1 level network topology. Typically there are N number of slave nodes and an uplink to a core switch or router.It is designed to run on commodity hardware. Commodity hardware implies average configuration systems which have medium fault tolerant components. Very low end machines have a lower quality component which leads to high failure rate and running such systems in large numbers leads to greater maintenance cost. On the other hand, large database machines are not recommended either, since they don't score very well on price performance curve.

To manifest maximum performance from the MapReduce framework, it's essential that the user complies by the network limitations. The data transaction between the master and the slave nodes must be kept minimum at the same time there must be full utilization of the bandwidth. This completely depends upon the partition function written by the user. It's up to the chunk of data or granules which are broadcasted to various slave nodes. If the granule size is very large then communication time becomes a great overhead and may also lead to loss of data since the hardware cannot handle such large chunks of data. On the other hand, if the granule size is very small, it may require larger IO cycles to communicate the data and the bandwidth of the network won't be effectively utilized.

- Setting Up Clusters :

The MapReduce framework can scale to N number of systems. The framework can easily be configured to run the job.

- Installing Python :

All the systems including master must have python interpreter , preferably of the same version. Python 2.6 is the most stable release of all versions. The following command can check the version of python installed.

$python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.

- Setting up the slave system :

Initially the user has to run the daemon process on each of the slave system in order to be enlisted in the cluster. While running the daemon process on the slave node , the user has to mention the IP address of the master node and the authentication key in order to establish connection. All the slave systems connect to the master node and are waiting to be authenticated.

Once the master node run the master process , it automatically incorporates all those systems which have valid authentication key.

- Initialization of master system :

The master process is initiated at the master node. All the slave nodes currently running the daemon process with a valid authentication key are enlisted. A dictionary of nodes are created where in the slave`s IP is the key and the status of the slave is its value. A slave node can either be idle or in active state. In the client list tab , the user can check on various slave systems connected to the master node and their respective states.

- Proposed Framework Properties :

The chunk size largely depends on users partition function. Although the chunk of data transmitted must comply with the hardware limitations. The partition input must be specified partition folder , which is the sub folder in master folder. All slave communications with the master occurs at port 4000, rest all IP`s are configured and communicated at run time.

A. Activity Diagram for Master

The master follows the following sequence of steps:

1. The master first authenticates the slave nodes.
2. The master then accepts input from the programmer i.e the location of the input file and the mapper, partition and reduce functions.
3. It then partitions the input file according to the user input into chunks of data.
4. The master communicates the generic map() function to all the slaves.
5. It then communicates the partitioned chunks of data.
6. It then waits to receive the intermediate map() results from the slaves and in the mean time updates the status of the tasks which will be reflected in the GUI
7. On receiving the intermediate map() results from the slaves, the master combines them and proceeds to perform reduce() on the combines results
8. It again updates the status to reflect the state of the tasks.
9. After the completion of reduce(),a final output is obtained which is written to a output file and displayed in a tab of the GUI.

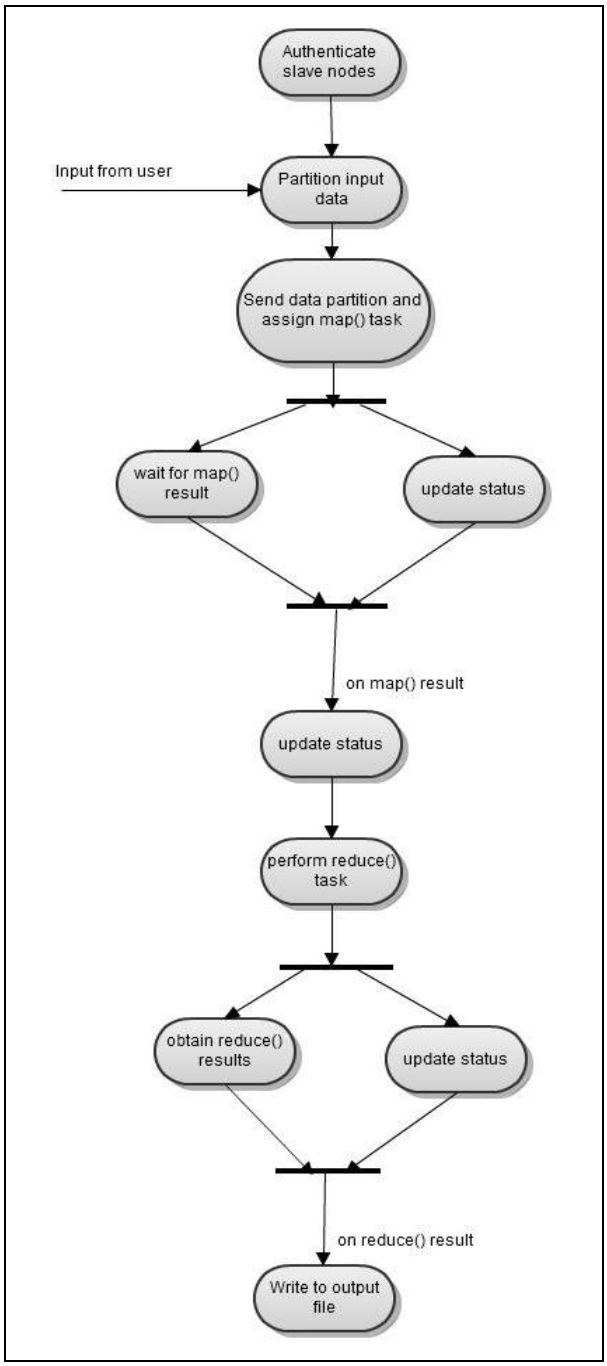

Fig. 6. Activity Diagram for Master

B. Activity Diagram for Slave

The slave follows the following steps:

1. The slave initially polls to connect to the master system
2. On obtaining an active connection to the master, the slave authenticates itself.
3. The slave receives a piece of code i.e the map() function, which specifies the computation to be carried out on the partitioned data chunk.
4. It then receives a data chunk from the master, which is to be processed, via the messaging queue
5. It then executes the map() function on the partitioned data chunk it received and produces intermediate results
6. These intermediate results are then communicated back to the master

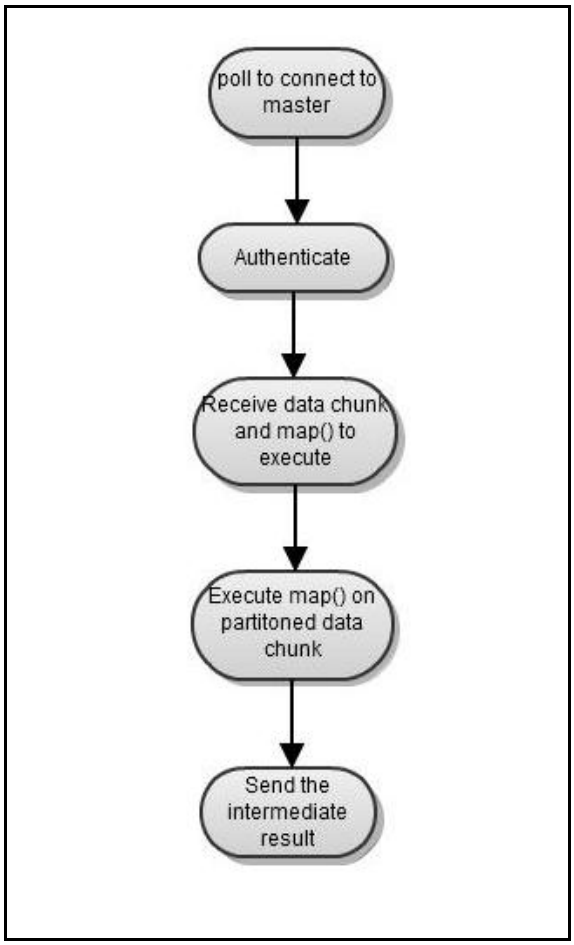

Fig. 7. Activity Diagram for Slave

C. The Graphical User Interface

The master node will have a GUI using which the programmer can interact with the MapReduce framework. The GUI consists of several tabs. Each tab has a specific functionality. In one tab the user can enter the map() function,the partition() function and the reduce() function. In another tab,the user can check the status of the functions assigned to the slave nodes. Another tab provides the user with the provision of viewing the final results. We have built our GUI using Tkinter of Python.

Tkinter is a Python binding to the Tk GUI toolkit. It is the standard Python interface to the Tk GUI toolkit and is Python's de-facto standard GUI, and is included with the standard Windows install of Python. Tkinter is implemented as a Python wrapper around a complete Tcl interpreter embedded in the Python interpreter. Tkinter calls are translated into Tcl commands which are fed to this embedded interpreter, thus making it possible to mix Python and Tcl in a single application. Other bindings, such as PerlTk, are generally implemented directly on the Tk C library.

D. Implementation of K-means clustering using Voronoi algorithm

K-means clustering[8] is a method of cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. This results in partitioning of the data space into Voronoi cells.

1) Kmeans Algorithm

Given a set of observations (x1, x2, …, xn), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets ($k \leq n$) S = {S1, S2, …, Sk} so as to minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

where μi is the mean of points in Si.

The most common algorithm uses an iterative refinement technique. Due to its ubiquity it is often called the k-means algorithm; it is also referred to as Lloyd's algorithm, particularly in the computer science community.
Given an initial set of k means m1,…,mk ,the algorithm proceeds by alternating between two steps:

- **Assignment step***: Assign each observation to the cluster with the closest mean (i.e. partition the observations according to the Voronoi diagram generated by the means).*

$$S_i^{(t)} = \{x_p : \|x_p - m_i^{(t)}\| \leq \|x_p - m_j^{(t)}\| \; \forall \, 1 \leq j \leq k\}$$

Where each Xp goes into exactly one Si(t) even if it could go in two of them.

- ***Update step:*** *Calculate the new means to be the centroid of the observations in the cluster.*

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j$$

2) Kmeans Using Mapreduce Framework

Our proposed framework implements MapReduce style parallel algorithm for Kmeans Clustering. Each map function gets a portion of the data. These data items do not change over the iterations, and it is loaded once for the entire set of iterations. The variable data is the current cluster centers calculated during the previous iteration and hence used as the input value for the map function.

All the map functions get this same input data (current cluster centers) at the beginning of each iteration and compute a partial cluster centers by going through its data set. A reduce function then consolidates the output of the map function with the most optimized centroids and also partitions the clusters such that, it represents the Voronoi diagram generated by the means.

The k-means algorithm takes as input the number of clusters to generate, k, and a set of observation vectors to cluster. It returns a set of centroids, one for each of the k clusters. An observation vector is classified with the cluster number or centroid index of the centroid closest to it.

A vector v belongs to cluster if it is closer to centroid i than any other centroids. If v belongs to i, we say centroid i is the dominating centroid of v. The k-means algorithm tries to minimize distortion, which is defined as the sum of the squared distances between each observation vector and its dominating centroid. Each step of the k-means algorithm refines the choices of centroids to reduce distortion. The change in distortion is used as a stopping criterion: when the change is lower than a threshold, the k-means algorithm is not making sufficient progress and terminates. One can also define a maximum number of iterations.

Since vector quantization is a natural application for k-means, information theory terminology is often used. The centroid index or cluster index is also referred to as a "code" and the table mapping codes to centroids and vice versa is often referred as a "code book". The result of k-means, a set of centroids, can be used to quantize vectors. Quantization aims to find an encoding of vectors that reduces the expected distortion.

3) Results

Figure 8 shows the output of the final reduce function containing twenty five most optimized centroids for each cluster. The different colored clusters represent the Voronoi diagram generated by the means.

Table I shows the comparative result of the K-means algorithm. It was observed that execution of k-means is much faster using our proposed framework(parallel) than when done sequentially

TABLE I. KMEANS CLUSTERING RESULTS

S/ No	K clusters	3D Coordinates	Computation Time(in Seconds)	
			Sequential	***Parallel***
1	75	300	0.423585891724	1.07124185562
2	75	3000	5.35187411308	1.84882116318
3	100	3000	6.18906092644	2.0945289135
4	75	30000	88.3964231014	11.5744128227
5	100	30000	116.474918127	15.3820641041
6	75	300000	1271.8319509	159.91038394

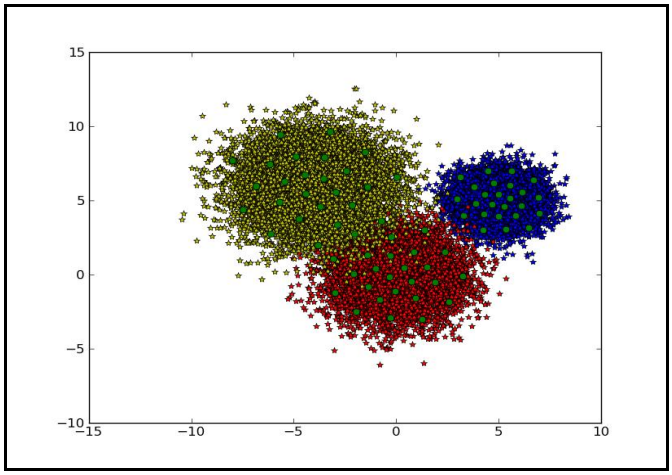

Fig. 8. k clusters are created in 3D coordinates by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.

E. Image Segmentation Histogram Algorithm

An image histogram[9] is a visual representation of the brightness of an image. In a digital image, each pixel is assigned a brightness value ranging from 0 to 255 for the colors red, green and blue (RGB). An image histogram is created by plotting the frequency at which each RGB value occurs in an image. Histogram of an image refers to the histogram of the intensity values of the pixels. It displays the no of pixels in an image for a particular intensity level.

Histogram Equalization is an important step in Image preprocessing and Image Segmentation.

1) *Histogram Algorithm*

To get a measure of how similar two images are, you can calculate the root-mean-square (RMS) value of the difference between the images. If the images are exactly identical, this value is zero. The following function uses the difference function, and then calculates the RMS value from the histogram of the resulting image.

```
import ImageChops
import math, operator

def rmsdiff(im1, im2):
    "Calculate the root-mean-square difference between two images"

    h = ImageChops.difference(im1, im2).histogram()

    # calculate rms
    return math.sqrt(reduce(operator.add,
        map(lambda h, i: h*(i**2), h, range(256))
    ) / (float(im1.size[0]) * im1.size[1]))
```

Fig. 9. Code Snippet of Histogram Algorithm

2) *Histogram Using MapReduce Framework*

Our proposed framework partitions the entire data set of images as per the criteria written by the user. Each partitioned chunk is then sent to the slave systems for further computation. The master also sends the histogram of the input image that is to be searched, to all the connected slaves.

The mapper code is then executed by each slave. The slave constructs histogram for all the images present in the data chunk and subtracts it from the histogram sent by the master. The result is then used for the computation of the root mean square(rms) value of the image. If the computed rms value is less than equal to the rms value specified by the user, the index value of that image is sent back to the master as the result of the mapper code exceution. This process is done iteratively on each image by the slave till the entire data chunk is exhausted.

The master then runs the reducer on the output of all the slave systems, extracts the images from the data set based on the index values and displays the images equivalent to the input image.

3) *Results*

A sample image pic B in Fig 10 is searched in a large data set using histogram algorithm in our proposed framework and pic A, pic B, pic C and pic D are the images found. All the images in the database are of size 150 X 175 pixels.

In Table II shows the comparative result of the Histogram algorithm. It was observed that execution of algorithm is much faster using our proposed framework(parallel) than when done sequentially

TABLE II. IMAGE SEARCH RESULTS

S/No	No. of Images	Computation Time (in Seconds)	
		Sequential	***Parallel***
1	100	0.91832613945	0.721187013626
2	1000	6.69655799866	4.0562620163
3	2000	12.1568629742	7.49930500984

Pic A Pic B Pic C Pic D

Fig. 10. A Sample Image

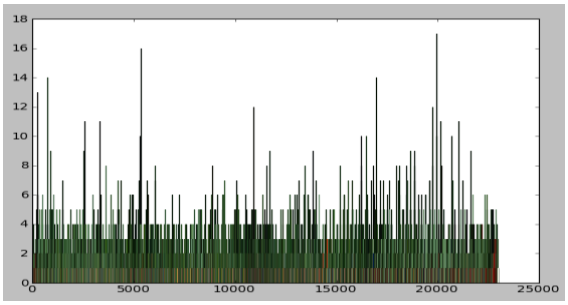

Fig. 11. Histogram of a sample image.

V. CONCLUSION

The MapReduce programming model has been successfully used for many different purposes. This success can be attributed to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems.

Also, we observe from the above implementation scenarios that, MapReduce can be successfully used to achieve faster and efficient results in the areas of network coverage optimization where Voronoi algorithm is implemented by means of K-Means clustering.

MapReduce is closer to the functional-programming, message-passing, data-copying paradigm. Although it allows explicit parallelism among the cluster systems, it`s imperative for the user to understand this paradigm in order to fit the problem into this structure. This further involves various parameters of the paradigm to be optimized for efficient processing of the problem. Even though the proposed framework can be scalable to large number of heterogeneous systems, the user still has to decide on how to chunk size to be broadcasted to slave systems. Lesser chunk size will lead to ineffective usage of the bandwidth and larger chunk size will make the communication time an overhead. Besides the proposed framework does not implicitly decide on the degree of parallelism, for a given data set, there exist a threshold level for the amount of which the data can be parallelized, beyond which further parallelism will decrease the speed of execution. The framework must be able to predict the number of systems to be involved for a given data sets and the computation performed on them.

VI. REFERENCES

[1] MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat jeff@google.com, sanjay@google.com Google, Inc.

[2] The Holumbus Framework Distributed computing with MapReduce in Haskell Stefan Schmidt Prof. Dr. Uwe Schmidt

[3] Hadoop:The Definitive Guide by Tom White

[4] Twister: A Runtime for Iterative MapReduce by Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox.

[5] ZeroMQ: An Introduction by Nicholas Piel.

[6] ØMQ - The Guide by Pieter Hintjens.

[7] The ØMQ Reference Manual by Martin Sustrik and Martin Lucina.

[8] An Efficient k-Means Clustering Algorithm: Analysis and Implementation By Tapas Kanungo.

[9] Image segmentation by histogram thresholding using hierarchical cluster by Damir Krstini´c⋆, Ana Kuzmani´c Skelin‡, Ivan Slapni˘car .

[10] Implementing the k-means Clustering Algorithm on Hadoop MapReduce by Jesse Cohen .

[11] Building Wavelet Histograms on Large Data in MapReduce By Jeffrey Jeste, Ke Yi, and Feifei Li.