Analysis of MapReduce Algorithms

Harini Padmanaban
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
harini.gomadam@gmail.com

ABSTRACT
MapReduce is a programming model that is designed to handle computations on massive amounts of data in parallel. The basic idea behind this model is the divide and conquer algorithm. The big data on which computations needs to be performed is split among nodes of a cluster, each of which performs the computations on the subset of data[6]. The technique used here is, when handling huge datasets, instead of moving the data across different programs, were are splitting the data among the cluster nodes and make the same program run on each subset of data. In this paper we are going to discuss the need for MapReduce algorithms, the basic concepts behind MapReduce and analyze a few map reduce algorithms and their performance[6].

1. INTRODUCTION
Firstly, let us look into the need for MapReduce algorithms. The availability of data plays a key role in the efficiency of many algorithms. Many real world problems such as classification and clustering problems, speech recognition, sensitivity analysis, information retrieval, and various other machine learning problems, perform better with larger datasets[6]. All these applications are mainly dependent on the training dataset. For example, let us consider classification models. The implementation of these models is as follows. Firstly these models are trained to learn using the training dataset. There are numerous algorithms available for training a model based on different approaches. It has been found that the accuracy results of these classification algorithms is better with the increase in the size of the training dataset. Also if the training dataset is very large, the performance of all these algorithms started converging, meaning that it is that availability of data that determines the efficiency and not the algorithms alone[6]. This motivated the organizations to start accumulating more and more data.

The magnitude of the data accumulated and processed by the organizations is very huge. For example, Google in 2004 was processing around 100TB of data for a single day with map reduce and this grew up to 20PB in four years, facebook is processing around 2.5 peta bytes of used data and ebay has accumulated around 7 peta bytes of user data and growing at around 150 billion new records per day[6].

Handling such massive data is not possible using sequential data processing. The data needs to be distributed across different nodes of a cluster and computations must be performed in parallel. This is where MapReduce comes into picture. It is a programming paradigm that allows for massive scalability across hundreds or thousands of servers in a hadoop cluster[1]. This aim of this paper is to explain about the basic structure of the map reduce programming model and analyze various MapReduce algorithms for different applications.

Overview of the next sections is as follows. Section 2 explains the basics of the MapReduce paradigm. Section 3 discusses various MapReduce algorithms and Section 4 summarizes the overall content.

2. FUNDAMENTALS OF MAPREDUCE
The concept of MapReduce comes from the functional programming model. The two main functions used in functional programming are map and fold. the map function maps a function f to all the items in a given list and the fold function reduces the results produced by the map primitive[6]. In MapReduce, these functions are performed by the mappers and reducers.

2.1 Mappers and Reducers
The map functions take as input set of key values pairs and produces n intermediate set of key value pairs. The reducer combines all the intermediate values associated with the same key that are emitted by the mapper and output a final set of key values pairs[6].

One of the main advantage of the MapReduce paradigm is that, it creates an abstraction between the actual computation and the implementation details of the parallel computation such as starvation, deadlocks race conditions, load balancing etc[6]. This makes it easy for software developers to concentrate on the actual job that needs to be done using the massive data, instead of worrying on the hardware level details.

A programmer usually writes the functionality of map and reduce functions and the framework, takes care of organizing and coordinating those computations.

2.2 Partitioners and Combiners
Partitioners do the job of partitioning the intermediate key space and the associate the key values pairs to the reducers. One of the simple way of partitioning is called the hash partitioning, where the intermediate key is hashed and the mod of it is taken with the number of reduces[6]. This helps in assigning the same numbers of tasks across all the reducers.

Sometimes the output key values pairs of the mappers becomes so large that, it is even bigger than the input. Combiners are basically used to combine these pairs before sending to the reducer. Combiners can be looked as "mini-reducers. They do local aggregation with the outputs receives from the mappers, before sending it to the reducers[6]. Thus it helps in improving the efficiency by reducing the count of the intermediate key values pairs. A combiner neither has access to the key values that are

generated by other mappers nor it processes values associated with the same key as a reducer does.

2.3 Hadoop Distributed File System

Now let us look into where and how the data is stored and how all these computations are performed on the data. The main concept that underlies the hadoop implementation is the data locality. The data to be processed is located on the same server where the computation is performed. While handling large data, storing the data separately would create performance issues as, all the nodes in the cluster might query for data and additional communication overhead might reduce the performance[2].

In HDFS, the huge data that needs to be processed is divided into chunks or blocks and these block are large in themselves. The blocks are then distributed across various nodes in the cluster[6]. In order to account for failures, the data is also replicated and stored across multiple nodes. Then computations are separately carried out on each of these nodes simultaneously.

The HDFS uses a master slave architecture. There is one master node called the name node which keeps track of the metadata such as, file to block mapping, location of blocks etc., and datanodes where the actual divided blocks of data to be processed resides. If an applications needs to access a block of data, it first reaches out for the name node, to get information about the location of the data block[6]. Having got the details of the location of the block, the application then contacts the data node directly in order to perform the computations. One very important thing to note here is that, the accessing of data or transfer of the data does not happen via name node. The name provides the application, with just the meta data of the block. Further communication between the data block, is done directly with the corresponding datanode[6]. The architecture of the HDFS and the communication between the client application and nodes is shown in figure 1 that was reproduced from reference [6].

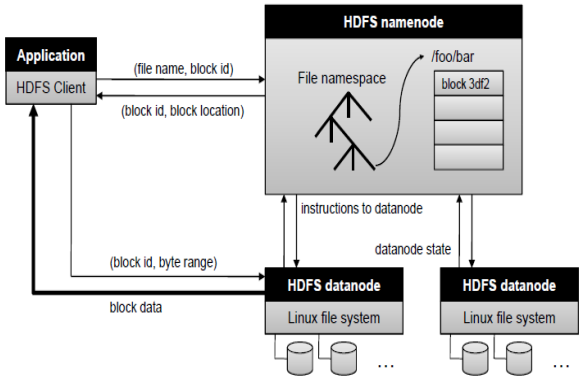

Figure 1: Architecture of HDFS [6]

Also in order to ensure availability and reliability, the HDFS replicates 3 separate copies of each data block and stores it across different data nodes[6].

2.4 Hadoop Architecture

The Hadoop architecture consists of three major components viz., the name node which acts as the master and stores the meta data of the split data. The job tracker node, which handles the coordination of the map and reduce tasks. and the slave nodes on which the actual data is stored. The slave nodes run two other programs called the task tracker and the data node daemon[6]. The task tracker keeps track of the individual map and reduces tasks carried out in that node and the datanode daemon, helps in serving the distributed data in hadoop.

A map reduce job has two main components viz., the map tasks and the reduce tasks. The number of map tasks created depends in the total number of files, the number of blocks into which the data is divided and the reduce tasks can be specified by the programmer. The programmer can also specify the number of map tasks in order to give an idea to the framework[6]. The task tracker in each of the data nodes sends information about he tasks it is running to the job tracker node. The architecture of Hadoop is clearly explained in figure 2, reproduced from reference [6], with 3 slave nodes.

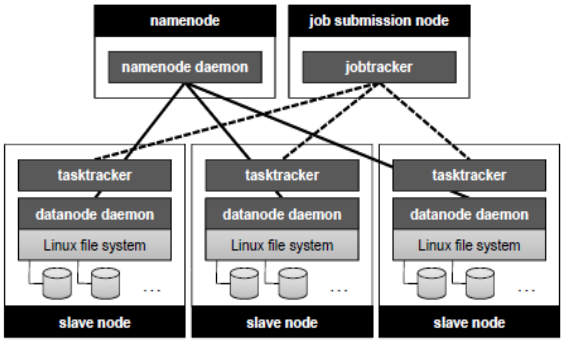

Figure 2: Hadoop Architecture [6]

3. MAPREDUCE ALGORITHMS

Having understood about the hadoop architecture and basic map reduce concepts, let us look into some map reduce algorithms that involve huge data and understand how the parallelism achieved through MapReduce helps in improving the efficiency.

3.1 Information Retrieval

The concept of information retrieval mainly deals with processing and storing huge amounts of data in a specific format and then querying the store data or get some information. The two important tasks in information retrieval are indexing the data (process and store the data) and searching (querying the stored data). Let us consider how MapReduce algorithms can be used for each of these tasks.

Searching:

Given a set of files where each file contains some text, the goal is to find the files that contain the given pattern. An implementation of this is as follows[5]:

[(Filename, Text) ,(Pattern)] is mapped to [filename, 1/0] where 1 is emitted when the pattern exists in the file and 0 otherwise. The reducer simply collects the filename corresponding to the emission of 1. Figure 3, that is modified and reproduced form [5] gives a scenario for this implementation.

In the process of searching, the pattern in the query must be checked for in all the data files (documents) and the documents

need be ranked bases on their relevance to the query. To achieve this we need to compute certain metrics from the data in documents. This metric is called as the TF-IDF score of the documents.

TF-IDF refers to term frequency–inverse document frequency. It is a metric used to quantify the relative importance of a word to a document in a collection or corpus [3]. It increases with the increase in number of occurrences of that term in the document.

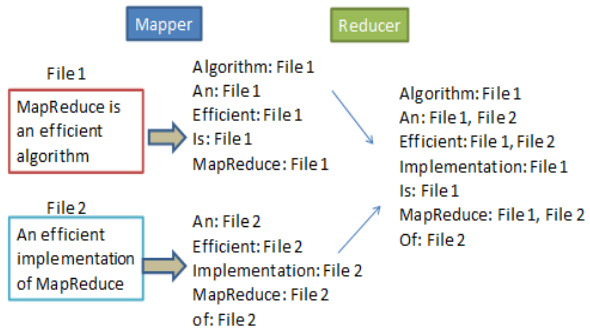

Figure 3: MapReduce for Searching [5]

The term frequency for a term t in document d is the ratio of the number of occurrences of the term in the document and the total number of words in the document as given below [4] .

$$tf(t,d) = \frac{n_t(d)}{N(d)}$$

The IDF is calculated as given below where N is the total number of documents in the corpus.

$$idf(t,D) = log\frac{N}{|\{d: t \in d\}|}$$

In order to implement this using MapReduce framework, we need to compute the following quantities[4].
1. For each document, the number of times the term t occurs: $n_t(d)$

2. For each document, the total number of terms: N(d)

3. Number of documents the term t occurs: $|\{d: t \in d\}|$

4. Total number of documents N

For each of the above steps the map reduce implementation is given below.

STEP 1: Compute term frequency

The map function for this step takes as input key, the document id and the values as the text document itself. for every term the occurs in the document, emits a values of 1 with the key as the tuple of (word, document id). The reducer then accumulated the counts of all the ones for each word and output the key value pair of word and count. The mapper and reducer function input outputs are abbreviated below as discussed in [5].

Mapper:
Input: ((document id, document terms))
Output: ((word, document id),1)
Reducer: Accumulates the word count for document id
Output: ((word, document id),count)

STEP 2: Compute total number of terms in a document.

The mapper takes as input the output of step 1 reducer and outputs the term count for each of the documents. The Reducer in turn sums the individual counts of words belonging to the same document and at the same time also retains the word-specific count. The functions are as given below as discussed in [5].

Mapper:
Input: ((term, document id),count)
Output: (document id,(term, count))
Reducer: takes as input metrics corrponding to same document id
Output: ((term,document id), (count, totalCount))

STEP 3: In the strp, the goal is to generate the number of documents in the corpus that the term appears in. The mapper takes as input the reducer output of the previous step. The map task emits a one every time it finds the term in the document along with the metrics of the term. The reduces then accumulates all the ones emitted by the map task corresponding to the same term. The functions are given below as discussed in [5].

Mapper:
Input: ((term, document id),(count, totalCount))
Output: (term, (document id, count, totalCount, 1))
Reducer:
Output: ((term, document id), (count, totalCount, totalDocs)

STEP 4: Now we have all the quantities that are needed to compute the TF-IDF. Assuming total number of documents (N) is known, TF-IDF is given by,

$$\left(\frac{count}{totalCount}\right) * \log\left(\frac{N}{totalDocs}\right)$$

This computation can be performed by the mapping function and we can use the results of the mapping functions without any use for reducer as the metric need not be reduced further[5].

3.2 MapReduce Paradigm in Digital Communication Systems

In wireless communication systems, the performance is evaluated with the metric Packet Error Rate. For example, to arrive at a packet error rate of 10^-3 typically requires 100/10^(-3)=10^5 simulations with random channel and noise. This is time prohibitive to run in a serial fashion. Since each of the runs is independent of each other, the time can be reduced if run in a parallel fashion. If Ts is the time taken to run one packet, the serial fashion requires NTs while distributed implementation such as MapReduce can leverage parallel processing it to reduce the running time by a factor of L where L is the number of maps.

3.3 Calculating Distributed Sum

Consider the case of distributed sum. Let N be the total number of terms. If done serially, the complexity in terms of additions is $C(serial) = (N-1)$. Now if this can be done parallel with M parallel units, the total number of additions in one parallel unit is $\frac{N}{M} - 1$. So the total number of additions, including the sum across parallel units, is

$$C(parallel) = \frac{N}{M} - 1 + M - 1$$

The optimal value for M here is $M = \sqrt{N}$. Thus, with one stage parallelization, we have reduced the complexity from O(N) to O($\sqrt{N}$). Now, here it is assumed that all the parallel units are synchronized at both the input and the output so that there is no latency. In such a case, the running time is proportional to complexity. The complexity can be further reduced by incorporating multiple intermediate parallelization. For example, if we have two stages, with M mappers in the first stage and R in the next, the resulting complexity can be written as

$$C(parallel) = \frac{N}{M} - 1 + \frac{N}{MR} - 1 + R - 1$$

Without any constraint on the number of intermediate hops, the optimal allocation will be N/2 mappers in the first stage, N/4 in the second stage, and so on. This results in log2(N) stages. So the total time complexity will be O(log (N)). Thus we have reduced the complexity from O(N) to O(logN). However, a practical issue is the synchronization and the latency. Since the second stage can operate only when the first stage has completed operations for all the mappers, there is a delay accumulation at each level.

3.4 Sorting Algorithm

Consider sorting algorithm as follows. The input to the sort algorithm is a set of files with each line consisting of a value. The mapper key is a tuple of file name and the line number. The mapper value is the text content of the line[5]. One of the important property of Hadoop (partitioning and shuffling) that can be leveraged for sorting algorithms is that, the key value pairs output by the mappers are sorted and given to the reducers. This is represented using figure 4 that has been reproduced from [5]. Thus the mapper task keeps emitting the values associated with the keys. The mapper input tuple are then forwarded to reducers. Due to the above mentioned property , the mapper to reducer forwarding is such that the smallest key is forwarded to the smallest reducer index and so on, the resulting combining is automatically sorted [5].

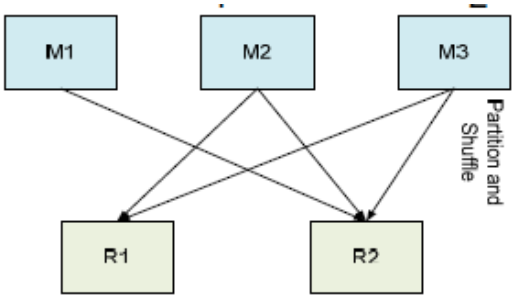

Figure 4: Sorting with MapReduce [5]

3.5 Graph Algorithms

Firstly, for a graph problem to be considered for MapReduce algorithm, the magnitude of such a graph should be very huge. Some of the fields that requires analysis of such huge graph are social networks, web pages, location maps etc. A graph is generally represented in terms of nodes and edges or links connecting the nodes. In case of social media, the nodes ay represent the individuals and edges represents the connections between them. The nodes and edges of a social network carry other some information with them such as, associated with each node(individual) is the age, sex, height, country etc., and associated with each edges are information such as type of relationship, strength of the relationship etc. If we take hyperlinks into consideration, the world wide web is full of web pages that are connected to one another. These hyper links themselves from the nodes of the graph and the weight of edges can represent number of hops required to reach another web page from the current page[6]. If we consider Google maps, that provides the route for all locations, that in itself is a very huge graph, with nodes representing locations. All these graphs discussed above are very huge and in order analyze and derive metrics from these graphs, sequential processing will never work out given the magnitude. Hence MapReduce comes into picture. This is because to store such huge graphs and processing them, is not manageable in single system.

Breadth First Search Algorithm

In order to MapReduce graph algorithms, we need to find suitable graph representations to store in the HDFS that facilitates efficient processing. Generally, these algorithms are iterative in nature. As far as this algorithm is concerned, it moves level by level for each iteration through the graph starting with a single node. In the next level it processes all nodes connected to the first node and in the second level, all the nodes connected to those nodes are processed[6]. This does not suite MapReduce programming model. Passing the whole graph to the mappers consumes a lot of memory and is wasteful. Hence it is essential to find appropriate representations for the graphs.

The usual representation of graphs as references from one node to other will not work out as it is has linked list implementations and cannot be serialized[5]. Also the representation of graph in the form of adjacency matrix is also not suitable as the graph is so huge with lots of zeros or ones. They are also composed of unnecessarily long rows to process most of which would be zeros. Now consider spare matrix representation that has the list of all the adjacent nodes for every node along with their weight[5]. This representation is simple and can be passed as arguments to the mappers or reducers easily as these only have non zero elements as part of the list. Thus it eliminates the necessity for storing huge matrices most of the values of which is zero.

Let us consider one of the famous graph problems, finding the shortest path from the source to all other nodes in the graph. This problem is solved using Dijktra's algorithm. It is a sequential algorithm. The algorithm works as follows. Initially, the distance from the source to all the nodes are initialized to infinity except for the distance to the source itself which is zero. The algorithms maintains a priority queue, and distance to the nodes in the queue is calculated staring with the node with minimum distance. In the first iteration it will be the source node itself[6]. Hence the source node is removed from the queue and the distance to all the nodes in the adjacency list of the retrieved node is calculated and the source node is marked as visited. In this manner the algorithm proceeds with each level of the graph. This is repeated until the priority queue become empty. At that point shortest path from source to all the nodes would have been computed[6].

Now this algorithm can be implemented in MapReduce using the parallel breadth first search algorithm is the. Let us assume that in this graph the distance between all the nodes equals 1. Let n

represent the node of the graph and N denotes the details corresponding to the node such as the distance to node from source and the adjacency list of that node. The initialization is same as in the dijkstra's algorithm. All the nodes are given as inputs to the mappers and for every node that the mapper processes, it emits a key values pair corresponding to the nodes in the adjacency list [6]. The key is the node itself and the values is one added to current distance to the node. This is because, we can say that, if a node can be reached with a distance of x then we can reach a node connected to that node with a distance of x+1. Now the reducers will get the key value pair that was output by the mappers i.e.(node, distance_to_node)[6]. From this the reducer needs to choose the shortest distance corresponding to each node and update it in the adjacency list. This process continues for the next iteration. With every iteration we will get the shortest distance to the nodes form the source in the next level. So if the diameter of the graph (or) the maximum distance between any two nodes fo the graph is D then we need to repeat this process for D-1 iterations to find the shortest distance from the source to all the nodes, assuming that the graph is fully connected[6].

One main difference between the Dijkstra's algorithm and the MapReduce version is that, the former, gives that path to the source form the node, but the latter only gives the shortest distance. In order to overcome this we also need to emit the path to the node in the mappers and update it[6].

Now if the edges have different weights then we need to alter the adjacency list to accommodate the link weights as well. And while finding the distance we need to add the link weight instead of adding one to the previously calculated distance.

The pseudo code for the map reduce functions is as below.

```
Map(node n, node N)
        d ← N.Distance
        Emit(node n, N)
        for all node m in AdjacencyList (N) do
        emit(node m, d + 1) .

Reduce(node m, [d1, d2,...])
        dmin ← ∞
        M ←null
        for all d 2 counts [d1, d2,...] do
                if IsNode(d) then
                        M ← d .
                else if d < dmin then
                        dmin ← d
        M.Distance ← dmin
        emit(node m, node M)
```

Pseudo Code for Parallel Breadth First Search, reproduced from reference [6]

3.6 Machine Learning and MapReduce

Machine learning is an important application for MapReduce since most of the algorithms related to machine learning perform data intensive applications. Some of the common algorithms in the field of machine learning include, Naïve Bayes Classification and K-Means Clustering.

Naïve Bayes Classification

Here the input to the algorithm is a huge data set that contains the values for the multiple attributes and the corresponding classifier. The algorithm needs to learn the correlation of the attributes with the classifier and exploit this information to classify test cases that contains the attribute information without the classifier. A typical example of training and testing data set for the naive bayes model is given in figure 5 and figure 6 respectively.

Table 5: Naive Bayes Training Data

Attr. 1	Attr 2	Attr 3	Attr 4	Attr 5	Class
a	1	4	0	'High'	2
b	3	2	1	'Medium'	3
d	2	3	0	'High'	1
b	1	3	1	'Low'	2

Table 6: Naive Bayes Testing Data

Attr 1	Attr 2	Attr 3	Attr 4	Attr 5	Class
d	3	3	0	'High'	?
b	1	4	1	'Medium'	?

For example, for any attribute A_i, the algorithm needs to compute

$$\Pr(A_i = S_m^{[i]} \mid C = c_k) = \frac{Count\ of\ all\ rows\ with\ A_i = S_m^{[i]}, C = c_k}{Total No\ of\ rows\ with\ C = c_k}$$

This needs to be repeated for all the states of A_i and for all the attributes and for all states of the class variables. Let N be the number of attributes, C be the number of classes and S be the number of states for a given attribute. Then the complexity of the above operation is O(CNS). However, the above operations are highly amenable for parallel implementation. It has been shown in [7] that the complexity can be roughly reduced by a factor of P with a MapReduce framework. Here P is the number of cores in the processor. The software implementation is as follows. Divide the input data set into multiple subgroups. Each subgroup handles certain tuples of (Attribute, State, Class).

K-Means Clustering

This is another important clustering algorithm used commonly in machine learning. In K-means clustering [8], the following operations take place

Input: Random points in a multi-dimensional space

Initialization: Generate K points in the input space that are to be used as cluster centroids.

Iteration:

Associate each of the points to the nearest cluster

Re-compute the cluster centroids

Output: Cluster index for each of the points

As can be seen from the above steps, the algorithm is amenable to MapReduce architecture where cluster computation and cluster association can be done in parallel. In [7], it has been shown that the complexity can be reduced by a factor of P where P is the number of cores.

4. CONCLUSION

Parallel computations have stated dominating today's world. Given any problem, the best solution needs to achieve the results efficiently. Solving a problem is no longer enough. In order to survive in this competitive world, everything should be performed time efficiently. To achieve that, we need to move toward parallel computing. The programmers can no longer be kept blind about the underlying hardware architecture. In order to scale up and

utilize the parallel infrastructure, the must know how to utilize the potential of such architecture and code efficiently. The MapReduce paradigm helps the programmers to achieve this by providing an abstraction to the programmer and hides the implementation details of the code. But the programmers still need to decide on which operation should be parallelized. That is, it is the duty of the programmer to decide on what the map and reduce functions do. From there on, hadoop takes care of the implementation of parallelism making it easy for developer. Dynamic programming is also another algorithmic concept that helps in solving highly complex problems.

By analyzing the various MapReduce algorithms, we could realize the potential of parallelization. There are many more MapReduce algorithms solving various problems. So we need to clearly identify which problems fit into the MapReduce model and efficiently utilize the parallel infrastructure.

5. REFERENCES

[1] What is MapReduce? Retrieved April 5, 2014, form, https://www-01.ibm.com/software/data/infosphere/hadoop/MapReduce/

[2] What id Hadoop Distributed File System (HDFS)? Retrieved April 5, 2014, from, https://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/

[3] TF-IDF. Retrieved April 10, 2014, form, http://en.wikipedia.org/wiki/Tf%E2%80%93idf

[4] Buttcher, S., Clarke, C.L.A., Cormack G.V., (2010) Information Retrieval: Implementing and Evaluating Search Engines.

[5] MapReduce Algorithms. Retrieved April 10, 2014, from http://courses.cs.washington.edu/courses/cse490h/08au/lectures/algorithms.pdf

[6] Lin, J., Dyer, C. (2010). Data-Intensive Text Processing with MapReduce. Retrieved February 10, 2014,from http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

[7] Chu, C.-T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G., Ng, A., and Olukotun, K. 2006. Map-Reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems Conference (NIPS)*. Vancouver, Canada.

[8] Retrieved April 5, 2014, form, http://en.wikipedia.org/wiki/K-means_clustering