Implementation of Map-Reduce

INF727 - Systèmes Répartis pour le Big Data

Author :
Olivier Maxwell

Professor :
Rémi Sharrock

Last modified : 4 janvier 2022

Contents

List of Figures

List of Tables

1. Introduction to Map-Reduce

1.1. Origin & concept of Map-reduce

Map reduce, originally developed by Jeffrey Dean and Sanjay Ghemawat at Google was officially released to the public in 2004 with the paper *MapReduce : Simplified Data Processing on Large Clusters*[1]. The Idea behind the project was to create a efficient way to run large calculation on clusters of of computers. At the time google was receiving an increasing amount of data, which common methods of data processing (mainly evolving signing big checks to IBM to use there powerful computers) could no longer handle. The method in the paper soon became the standard of map reduce offering a number of advantage over large scale supercomputers :

- *Scalability,* map reduce was designed with scalability in mind, if more memory or processing power is needed, it is relatively simple to add computer to the cluster.
- *Redundancy,* thanks to the cluster nature of map reduce multiple redundancy protocols are in place in the event a computer of the cluster fails it is possible to continue the operation without having without restarting from nothing.
- *Big data,* thanks to the previously discussed two points, map reduce is optimised for big data analytics.

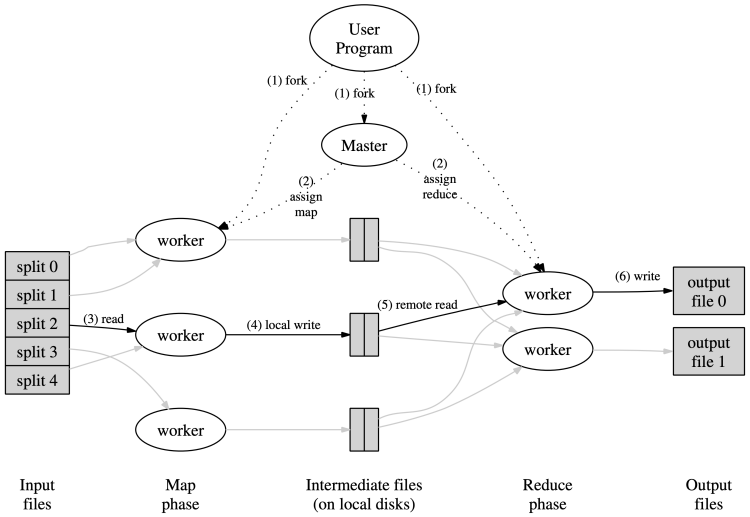

FIGURE 1 – Original Map Reduce *by Jeffrey Dean*

Map reduce is based on 3 main operations :

- **Map**, where each worker of the cluster applies a map function to the local determining which worker node will be processing the data.
- **Shuffle** based on the previous output each node (worker) of the cluster send the data to the corresponding node
- **Reduce** each node applies operations to its local data (received in the Shuffle phase) and therefore reduces the data before outputting its output file.

This entire process is summed up in figure 1 taken form the the original paper by Jeffrey Dean [1]

1.2. Implementation Assignment

This paper will discuss my implementation of map-reduce, the task is to design and implement a bare-bones version of map-reduce in JAVA. The bare bones map-reduce has one task : count the occurrence of each word in a large document. My implementation is fully available on github https://github.com/oli360/My_map_reduce

2. Implementation

The implementation of map reduce discussed in this paper is composed of two main jar programs, one master and one slave (or worker, the two terms are used interchangeably throughout the report). The master code is used to control and manage the workers, instead of describing both jar files in this report, the implementation will be analysing by going through the steps of map reduce and describing the interaction and actions going on within the code.

2.1. Preparing map reduce

At the starting point of map reduce we have one master device which contains :

- *Master.jar* a master jar file
- *Slave.jar* a slave jar file
- *IP_address.txt* a text file containing the IP address of working and available potential worker machines
- *Input.txt* a (hopefully large) text file, the goal being to count the words contained within it

The map reduce process is started by running the *Master.jar* file with java, it will then

1. Split the input.txt file based on a size selected by the user
2. Fetch the ip address of worker nodes
3. Deploy the *Slave.jar* files on the worker nodes along with a random split of the text file and the IP address of the different worker nodes needed for the operation,the amount of worker nodes depend on the size of chunks and the original size of the input file, it is governed the equation by equation 1

$$NumberOfNodes = \frac{OriginalDataSize}{SplitSize} \tag{1}$$

During the deployment of the jar files the first issue arose, when sending the files the directory was not always existent. Therefore the *cp* command sent to the worker node would throw an error, this was handled by waiting for a response by the worker node and, if a directory error occurred, a *mkdir* command would be send before the sending the files. The drawback from this method is it not being optimum since the algorithms waits for a response from each node before sending the file to the next node. However since this step is not integral to map reduce not much time was spent on it. In a more realistic situation all nodes would have the same working directory. This scenario is simulated in the implementation with a *deployJar* variable if true, the algorithm assumes all nodes are empty (only true if all nodes are completely clean). The process ensure all files are well transferred by looking at the console and waiting for the process to finish. The code below shows this

```
public static void checkSlaveResponse(List<Slave> slaves){
    for (Slave slave:slaves) {
        try {
            System.out.println(slave.helper.getSysOut());
        } catch (IOException e) {
            slave.alive = false;
            e.printStackTrace();
        }
    }
}
```

Once all the nodes have the right directories and jar file, the master node will initiate the map function,

2.2. Map

The purpose of the map function is to send all the word to the right node to be counted, this is done using a hash function to map each word to a corresponding node. This procedure is initiated by the master,

with the command *ssh omaxwell-20@tp-4b01-21 java -Xmx6000M -jar /tmp/omaxwell-20/SLAVE.jar 0 S00.txt 1*. The command takes three parameters, the phase to run, the text file to map, and the number of nodes in the cluster. The master send the command to all nodes and then waits for the response, this ensures the nodes run concurrently and not sequentially. Once all the Mapping is completed all nodes should have created a number of text files to send to other nodes of the cluster, for example in the case of 4 nodes in the cluster, node 1 will have 3 text files to send to nodes 2,3 and 4. The master waits for all nodes to have finished this process by waiting the system output of the nodes. It then moves on to the shuffle.

2.3. Shuffle

The shuffle consists of sending all the text files to there required destination, this is done by running the same jar file on the local nodes however with the parameter two, this makes all nodes send all the files in there *export* directory to the respective node. This part does not contain any special optimisation since it depends mostly on the network and in the case of this assignment this subject out of bounds. However the master runs the jar file on all the nodes and then waits for a response, ensuring all nodes are working at the same time. The nodes also send the files in a parallel manner.

One issue met during this phase is when working with large clusters (over 20 nodes), during this phase all nodes are sending *scp* commands to all other nodes which would sometime saturate the computer resulting in multiple *Connection closed by remote host* no solution was found for this issue other than multiple attempts at running the algorithm multiple times. Once The master has waited for all nodes to have completed the Shuffle is looks a the console log to ensure all transfer where successful and starts the reduce.

2.4. Reduce

At the start of the reduce all nodes have received the words they will need to count in multiple text files. The master will then run the jars with parameter three, this will start a word count on all nodes, resulting in all nodes creating a *reduced.txt* file contain the words count for the words they where assigned. The reduced file is created by opening all the local files and counting the words contained within them, then outputting a final reduced file with the count.

2.5. Finishing map reduce

The final step of map reduce involves retrieving all the reduced files from all the nodes and concatenating them to make one large word count file. This is done by retrieving the files sequentially using *scp* commands Reading them in java and outputting a final large file with all words sorted.

3. Optimisation

Over the course of the assignment multiple aspect aspects of the map reduce implementation was optimised, this section will discuss some of these optimisations.

3.1. Parallelising run functions

The first key area of optimisation was parallel processing, this involves making all nodes run at the same time instead of one after the other, this is a key area of map-reduce, this was achieved by running the jar file concurrently and then waiting for the response from the first node and hen on, this way all nodes are working at the same time, this improves time by approximately n times , n being the amount of nodes, (for small amount of nodes, as defined by Amdhal's law)

3.2. File split

The second area where optimisation is possible is during the file split , the original method used was iterating through the file and splitting when a certain amount of words was reached in java, this method required java to iterate through the entire file which in some cases was over 1 GB. This could take some time, the split method was therefore modified to use the split function however since MAC OS only has a dialled down version the complete version of split, called gsplit was installed from from coreutils using

brew, this resulted in under 2 second split times. which stayed approximately constant no matter the number of split, shown in figure 2.

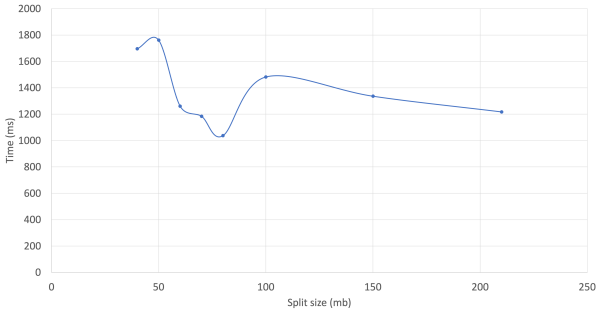

FIGURE 2 – Split time vs Split Function

3.3. Word count

The third and most improvement in execution time was word handling within the java code. The original method involved writing each word to a file and reopening said file to add each additional word resulting in a heavy amount of HHD read and write times. With a bit of research a google class called *ArrayListMultimap* was imported it behaves similarly to a dictionary however instead of a key value, it is a key list, allowing for easy adding of words for a certain key.

The algorithm would use this class to split all words and write the content of all keys to text files to be later transferred to the right node.

4. Final results

4.1. Performance

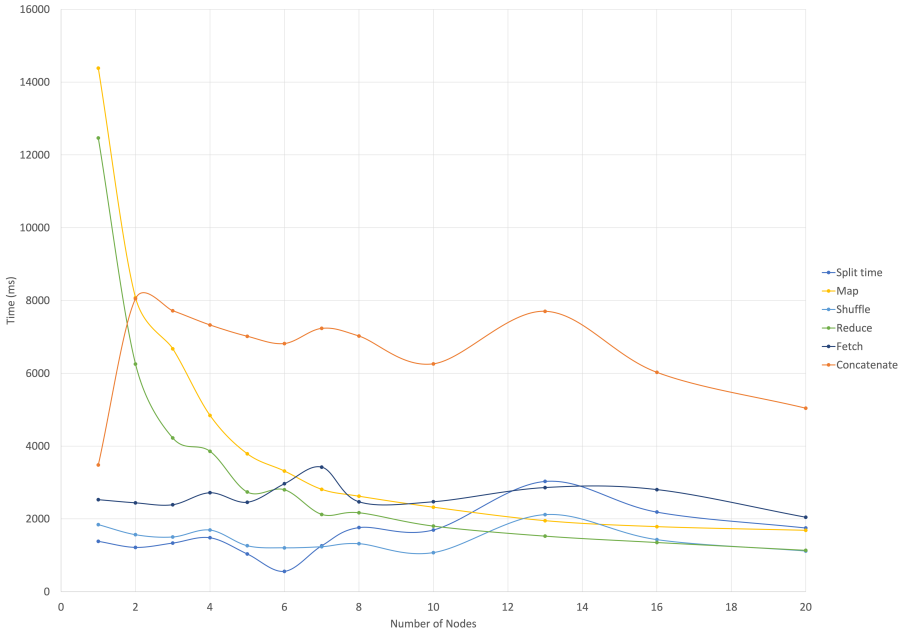

FIGURE 3 – Time taken for each step of implementation

With these optimisation in place it was possible to count the occurrence of words in the amazon web crawl in 54s with 10 nodes, however this is taking into account the deployment of the jar file along with sending the splits from my personal device to the devices in the lab. Only sending the files took over 42s. If

we only look at the time after at the split and ignore the time taken to retrieve the files, this implementation could do in only 4 seconds.

If we look at the time taken by the different step of the implementation shown in figure 3 it is quite clear that the reduce and and map phases accelerate the most with the increased amount of nodes. The information is also shown in figure 4, which represents the speed up based on the amount of nodes, this also demonstrates Amdal's law, although the plateau could not be well represented since going above 20 nodes causes multiple erreurs and ssh timeout due to large number of connections required.

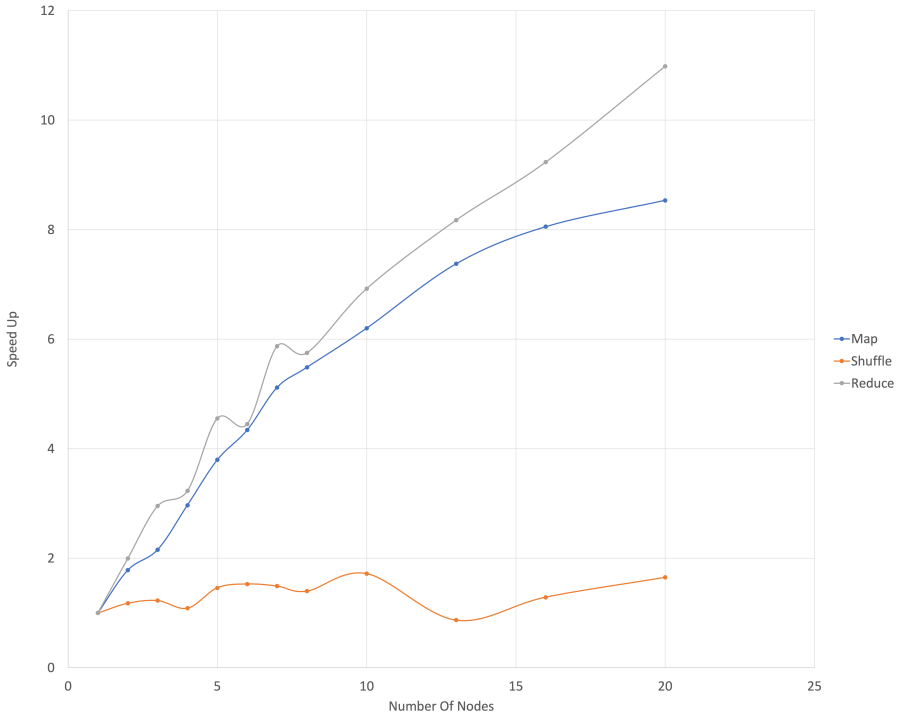

FIGURE 4 – Amdahl's law

The results of the word count for the AWS crawl are displayed tin table 1 however these values will not be discussed further in this report.

Word	Count
the	567896
a	474187
to	432431
and	426549
of	369365
WARC	305280
in	293656
de	268154
s	264926
2	228001
for	195277
n	193818

TABLE 1 – AWS web crawl top words

4.2. Potential improvements

Most easy wins where implemented in this map-reduce, however two key areas could have been implemented.

4.2.1. Error handling

In its current version all errors are cough by the master and are clearly displayed, stopping any mistakes from happening to the final count, this has the advantage of guaranteeing the validity of the final word count. However some form of failure detection could be implemented where the master detects what node went down and for what reason and acts accordingly.

4.2.2. Moving the master

Quit a bit of time is lost transferring files from the master (local personal computer) to the TP computers, this time could be reduced by moving the master to the TP computer. This was not implemented in the project since its objective was to understand and improve map-reduce, moving the Master would have made development more complicated (due to the lack of IDE and graphical interface of TP computers). The efficiency of the implementation can be judged without the transfer times, this being said the total map reduce could have been reduce by about 75%.

Références

[1] Jeffrey Dean et Sanjay Ghemawat. “MapReduce : simplified data processing on large clusters”. In : *Communications of the ACM* 51.1 (2008), p. 107-113.