SLR207 - Technologies de calcul parallèle à grande échelle

Hadoop MapReduce from scratch

Rafael Sandrini Guaracho

July 3, 2023

Contents

1 Introduction

Hadoop MapReduce is a powerful software framework designed to process massive amounts of data in a distributed and fault-tolerant manner. It enables the processing of multi-terabyte data sets by leveraging large clusters of common hardware. The framework divides the input data into independent chunks that are processed in parallel by map tasks. The outputs of these map tasks are sorted and fed into reduce tasks. The input and output of a MapReduce job are typically stored in a file system. The framework handles task scheduling, monitoring, and re-execution of failed tasks.

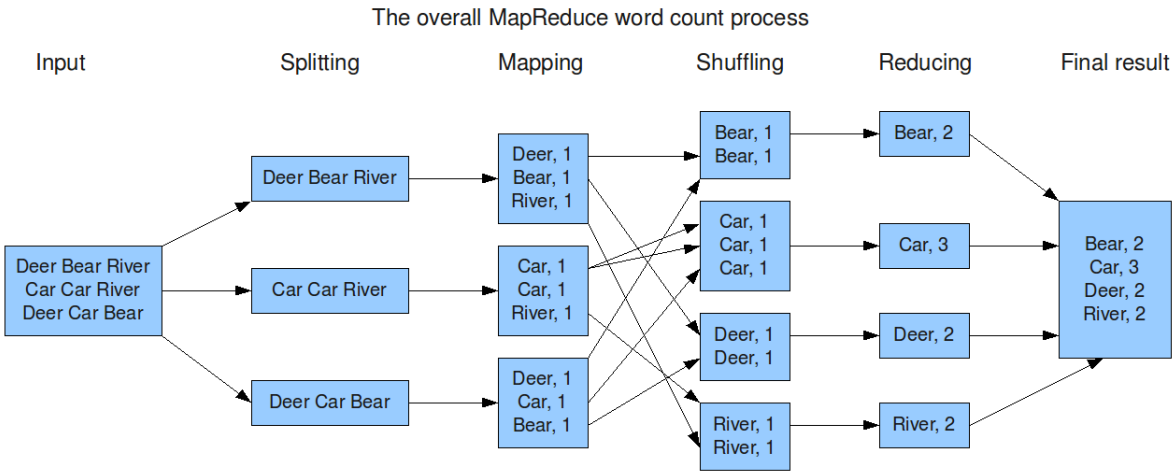

Figure 1: Hadoop MapReduce diagram

2 Setup

To run the algorithm, the text file must be in the Master computer (it is coded as sourceFilePath in the SimpleClient.java file). It is possible to use the *deploy.sh* file to copy, compile SimpleServerProgram.java and ServerThreadListener.java and run SimpleServerProgram on N chosen computers.

To run SimpleClient.java: is expected to enter with the arguments in this order: portNumber host0 host1 host2.

Example: java SimpleClient 9976 tp-1d22-05.enst.fr tp-1d22-06.enst.fr tp-1d22-07.enst.fr tp-1d22-08.enst.fr

To run SimpleServerProgram.java: is expected to compile the file ServerThreadListener.java and enter with the port argument.

Example: java SimpleServerProgram 9976

3 Implementation

For the implementation, we followed the design of a Master that will be responsible of synchronizing its Slaves into all the sub-tasks we projected for the algorithm.

3.1 Master

The master (SimpleClient.java) receives as main arguments the port number and the IP addresses of all its slaves. He builds a connection HashMap (connexionDict) giving an id for each slave and it's IP address, then he sends this HashMap for all its slaves (connexionMessage).

After that, the master reads the file and splits in equal sized chunks to send as bytes to its slaves, enabling a fast split and transmission of the data.

Then the master starts his paper of synchronizing, waiting for the slaves' messages in their respective sockets.

The message protocol built consists of 8 different messages that the Master can receive from the slaves:
OK: Consists of the slave's final message, after printing its final local MapReduced

$DICT_{ACK}$: Slave's message after successfully receive the master's file, retrieve it into strings, split and build a HashMap. Then the master can send the prepare shuffle message.

$PRESHUFFLE_{ACK}$: Slave's message after sending all its words to their respective responsible (also a slave). Then the master can send the wait shuffle message.

$WAITSHUFFLE_{ACK}$: Slave's message after successfully read all its Threads, i.e received all the words that were assigned to it. Then the master can send the shuffle message.

$SHUFFLEK_{ACK}$: Slave's message after successfully build the shuffled HashMap with its words.

Range: Slave's message with its calculated range (its maximum and minimum word count). Then the master can calculate the global range and send to their slaves.

$SECONDPRESHUFFLE_{ACK}$: Similar to $PRESHUFFLE_{ACK}$, slave's message after sending all its words to their respective responsible, but this time the responsible was obtained with the ranges. Then the master can send the second shuffle message.

$SECONDSHUFFLEK_{ACK}$: Similar to $SHUFFLEK_{ACK}$, slave's message after successfully build the second shuffled HashMap with its words. Then the master can send the finish message, to turn off all the slaves.

3.2 Slaves

The slaves (SimpleServerProgram.java) receives as main argument the port number.

Each slave start receiving the all slaves' ids and IP Addresses from the master, so it is possible to build its own connexionMap and also determine its id and the total number of slaves.

After that, it will receive the file from the Master and build its HashMap mapServerWords defining one list of strings for each slave (based on a hashCode function and the mod function by the total number of servers).

Then the slave sends all its words to their respective responsible. The words that the slaves share between them are all allocated into the mapThread HashMap (inside ServerThreadListener.java) for each Thread. The words the slave should locally keep are allocated into mapPreShuffle HashMap.

After that, the slave can merge all the mapThreads into a big map called mapThreadsPreShuffle and use this result to merge with its own mapPreShuffle, resulting in the final first shuffled result map mapShuffleWords. It is important to note that we are building this map with a list of integer in its value, so it is possible to determinate how many words are coming from each slave, if some application request it.

With the result from the first shuffling, it is easy to sum all the occurrences and build the MapReduced HashMap, having the count of occurrences for each list of strings, and then determine its maximum and minimum value (range) to send to the Master.

After receiving the respective calculated global ranges for all the slaves, each slave iterate through its MapReduced and send the words to their respective responsible. So we have a similar approach to the one already explained, with each Thread allocating its local result into a secondThreadMap HashMap and after that being merged with the mapSecondShuffleWords HashMap, finally giving us the final result. It is also possible to do a local sort with this map, depending on the application purposes.

The message protocol was already explained in the Master sub-topics, it consists on the same messages but without the $_{ACK}$ suffix.

4 Analysis

The implementation was tested with the text file "CC-MAIN-20230320211022-20230321001022-00511.warc.wet" from the common crawl repository and the responses times were plotted.

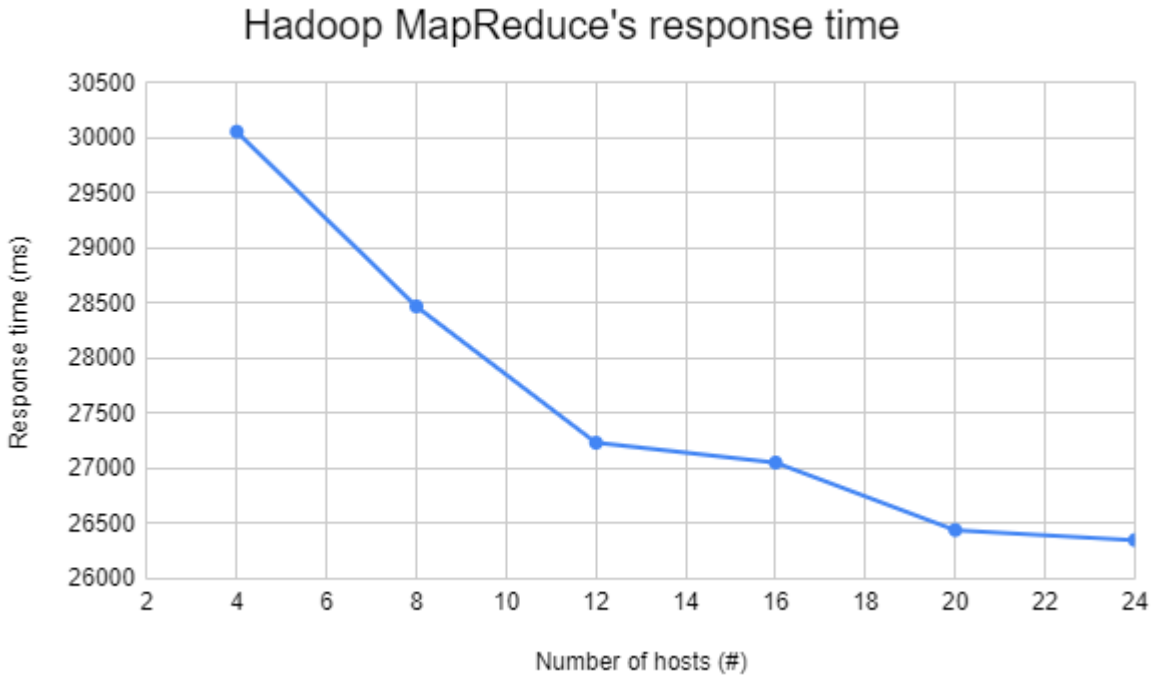

Figure 2: Hadoop MapReduce's response time

From figure (4), we can verify that the response time got better results as we increased the number of hosts. The worse time of 30058ms was achieved with 4 hosts and the best time of 26351ms was achieve with 24 hosts.

It is important to note that in the implementation we have two outputs of time: the first one is the result after the build of all final local MapReduceds (after the second shuffle) and that is the result we showed in the graph. The second time is the one after printing all their final local MapReduceds (what takes a lot of time for the machines with lots of words - average 115500ms).

5 Conclusion

As we saw in the last page, it is possible to conclude that the results were expected and aligned with Amdahl's law. We got a faster system each time we increased the number of hosts, but the difference in time gains were tending to decrease, inferring that after a determined number of hosts there will be no significant improvement in response time.

The implementation is working for the 300MB files and all other low sized files tested. It can give an error for a big number of hosts, when the master takes too long to send the file (problem mentioned in the next topic - blocking sending) and then it may lose the response from the slave if it finishes faster than the master sends data to all slaves to start listening.

6 Possible improvements

As I noticed during the tests, my algorithm is taking a significant time to read the text file and send the chunks to all the slaves, it is happening because it is reading each chunk, stopping and sending to the determined slave, blocking until the end of the write. It can be improved making a parallel reading of the file with a chunk sized offset, making it possible for all the slaves to receive their content in almost the same time (instead of making N blocking communication steps).

Another point I observed is that the global range calculation it is being done in a bad way because I am not tokenizing my words before building the HashMaps, so, there are some common words and punctuation such as *to*, *the*, *and*, *of*, *de*, :, − that are appearing a lot of times (more than 35000 for example) and compromising the range calculation. It results in ending with almost all the words in the first machine because it is the only one with a "good range", and that is the reason why the second response time (after the print of the final result) is so high, it takes a lot of time to print almost all the words.

Improving the blocking read to a parallel read and doing a tokenization in the words set would make a huge difference to my result.