Map Reduce Implementation Project Report

Vinícius BERNARDES BONEMER

April 30, 2021

1 Introduction

The aim of this project is to implement the MapReduce programming model in Java in the context of the WordCount problem. The problem consists of calculating, for a given input file, how many times each word is repeated. The implementation will then be compared to a sequential one in light of their performances. The distributed implementation will then be run multiple times with different numbers of machines involved and the results will be used to calculate an approximation for the percentage of strictly sequential code by comparing the results with what's expected from Amdahl's law. The final goal is to see the predictions of Amdahl's law in practice.

2 Amdahl's Law

The Amdahl's Law gives an important formula in the area of distributed systems. It allows to calculate the expected speedup observed in a task as the number of parallel processors working on it increases. Every computation is described by a series of fundamental operations executed either in parallel or sequentially, but because it is impossible to do meaningful calculations without data, it is expected that information needs to flow from one operation to the next until the final result is achieved.

This necessity of data flow creates a dependency between operations – if one operation (i_0) relies on the data calculated by another one (i_1), then this imposes a total order between the two: $i_0 < i_1$, where $<$ denotes the precedence relation. The whole execution being comprised of both sequential and parallel computations, it is then described by a partial order of operations that will finally produce an output.

If a computation has no dependencies between its operations, then its fundamental operations could be calculated all in parallel, each in a different processor. If, on the other hand, a computation has dependencies between each operation to the next, then nothing can be parallelized. Amdahl's law take this into account, so that the calculated speedup depends on the fraction of strictly parallel code.

The Speedup $S(t_0, t_1)$ between two tasks is a factor that corresponds to "how much faster" t_1 is if compared to t_0. That is, if $L(t)$ represents the latency of the task t, then the speedup is as follows.

$$S(t_0, t_1) = \frac{L(t_0)}{L(t_1)} \tag{1}$$

In the context of the Amdahl's Law, this Speedup is calculated for a same task with different number of processes. In this case, the interest is focused on the comparison between parallel and sequential executions, so the Speedup of task t when executed in n processors, as opposed to a single processor, is represented as

$$S(t) = \frac{1}{(1 - P(t)) + \frac{P(t)}{n}} \tag{2}$$

where $P(t)$ represents the amount of parallelizable operations in task t.

3 MapReduce

From the Amdahl's law, it follows that in order to have satisfactory speedups when parallelizing execution of some code, it is necessary to have a small part of strictly sequential code. MapReduce is a programming model that ensures the resulting program will have this property, and thus will have greater speedups when ran in parallel.

This paradigm achieves a high level of parallelization by splitting a single input in many smaller inputs that can each be processed by a different machine. This requires that the actual computation is broken into smaller steps: *split*; *map*; *shuffle*; *reduce*; and, optionally, *retrieval*. Those steps are illustrated in the context of the WordCount problem by Figure 1.

As seen in Figure 1, the first step consists in splitting the input, here line by line. Each split is then sent to one different computer. The second step is mapping, where each word is placed in one line followed by 1.

The following phase is called shuffle, and its purpose is to prepare for the reduce phase, where all equal words will be joined into one line containing the word itself and how many times it's repeated in the document. In order for this to be performed, a single processor should be responsible for processing

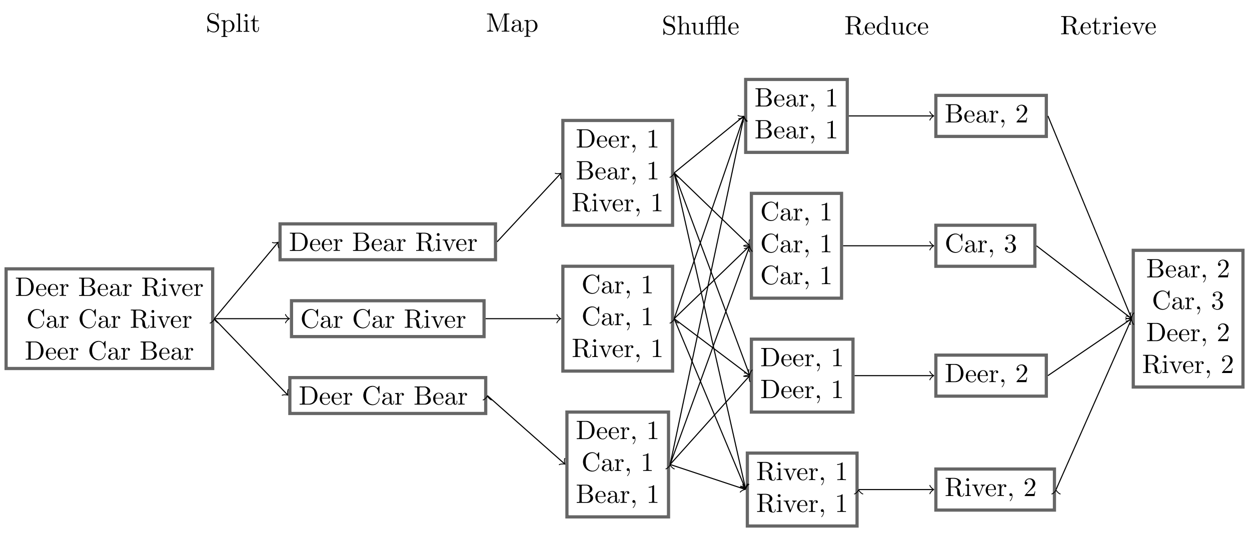

Figure 1: MapReduce execution

every single line that a given word appear on, so that the reduce phase actually computes the right number.

The shuffle phase is responsible for sending each line to the correct processor. In order for this to happen, a hash of each word is calculated and, assuming machines are numbered in a fixed order, the machine to which the line should be sent is calculated as follows.

$$machine = \mod_{\#machines} \text{hash}(word) \tag{3}$$

Finally, after the reduce phase, the results are ready in each of the nodes in the network. A last retrieve phase can be performed to bring those results back to the original machine and merge them.

4 Implementation

The implementation is broken into 5 modules, two of which implement the algorithm (`master` and `slave`), two helper modules (`clean` and `deploy`) and one final module which implements a sequential version of word counting to compare (called `sequential`).

The `clean` module is used for deleting any files created during previous executions in the remote machine, while the `deploy` module creates the expected directory structure and sends the code of the `slave` module to the remote computers in a jar file. The `slave` module has the code that is run in remote machines, while `master` is the code used in the local machine which starts the system. The only module that runs on the remote computers is `slave`, while the other ones run locally.

4.1 Deploy and Clean

The `deploy` package is responsible for building the expected directory structure and sending the slave jar file to the remote machines. All the tests are going to be run in Télécom's computers, which have a remote file system that ensures files from one user is accessible from any computer in the network. This is a problem if a distributed implementation needs to be tested, because it doesn't allow to differentiate between local and remote files.

In order to work around this problem, all files will be stored in the `tmp` directory of each computer, because this directory is stored in the actual machine. The deploy phase will thus create a subdirectory of `tmp` named `vbonemer` to store the used files, and ensure they are not mixed with other students' execution results. The expected starting structure is as illustrated by Figure 2.

The `clean` package is responsible for making sure every file from a previous execution is removed before a new one starts. This is necessary to avoid mixing up files and getting a wrong result. Because the `deploy` package creates the needed file structure before an execution, the `clean` package can be very simple: it just deletes the created directory.

```
/tmp
└─ vbonemer
   └─ slave.jar
```

Figure 2: Starting directory structure in a remote machine

4.2 Master

The `master` package is responsible for splitting the original file, sending each split to one machine and coordinating instances of the `slave` program to compute the result. The master program is initialized with a list of machines to use. The connection to each of the machines is then tested and the list is filtered to contain only available machines. The input file is then split and each split is sent to the appropriate machine before it can start the first phase of MapReduce. A list of all the used machines is also sent to each one, as it is necessary in the shuffle phase.

To split the original file, it's size is divided by the number of available machines to determine an approximate split size (s_a) in bytes. Then, s_a bytes are copied to the split file and the original file is read until a white-space character is found. Those extra characters are then copied to the split file before moving on to the next one.

A minimum average split size can also be set when the size of the input file is unknown. That way, if a file is too small, the number of splits could be less than the number of available machines, which may be beneficial – when files are too small executing more sequential code in one machine can result in better performance due to the reduced amount of data exchanged over the network.

If, for example, the input file (f_{in}) from Figure 1 is used, ten machines are available (n) and the minimum average input size is set to $s_{a,min} = 12$, the algorithm will calculate the approximate split size as follows, assuming each character is coded in one byte.

$$s_a = \left\lfloor \frac{|f_{in}|}{n} \right\rfloor = \left\lfloor \frac{43}{10} \right\rfloor = 4$$

Because the minimum average input size $s_{a,min} > s_a$, the value of $s_{a,min}$ will be adopted as the new average size. The algorithm will then copy the first $s_{a,min} = 12$ bytes of input to the first split file, which corresponds to the string "deer bear ri", which stops in the middle of "river". Next, the three following characters would be copied until the next white-space is reached, resulting in the first split "deer bear river". The second split is created from the next 12 bytes incremented until the white-space, so $S_2 =$ "car car river". Finally, $S_3 =$ "deer car bear". When the end of the file is reached, the splits are ready.

With this algorithm, splits can (and most often will) have different sizes. The size of a split i will be $|S_i| = s_a + \delta_i$, where δ_i is the number of characters between the s_ath character of the input and the next white-space. For most of the splits, this value will be negligible if the size of the split is much larger than the size of the largest word in the text, but these values accumulate, as the start of the next split is always offset by $\Delta_i = \sum_{j=1}^{i-1} \delta_j$ relative to the expected start based on s_a. This means that the last split may have a much smaller size than expected. If this is a problem, the algorithm can be adapted to make each split with size $|S'_i| = s_a - \delta'_{i-1} + \delta'_i$, by copying the first $s_a - \delta'_{i-1}$ characters plus what's left before the white-space, where δ'_i is the equivalent of δ_i in this new configuration.

4.3 Slave

The `slave` package actually execute the algorithm described in section 3. The code doesn't differ much from that algorithm, except that it didn't specify how words were shared between processes. In the actual implementation, each step creates a new directory and saves files to this new directory and these files are accessed from the following step.

In the first phase, each instance of the slave program takes the split file that the master saved in its `splits` directory and starts the map. When the map phase is finished the slave will have written a file with its output into the `maps` directory. After this phase, the shuffle is executed. To perform the shuffle, the slave creates a new `shuffles` directory. Then, for each word, it joins all lines with this word in a single file named `<word_hash>-<machine_name>.txt`, where `<machine_name>` is the name of the machine that calculated that shuffle. Those files are then sent each to the appropriate destination and saved in a directory called `shuffles_received`.

The final phase that the slave executes is the reduce. It gets all the files it received in the previous step and sums the occurrences of each word. For each word, the result is written in a file named `<word_hash>.txt` in the `reduces` directory. In the example input discussed in Figure 1 and section 4.2, the resulting directory structure for the machine that received split S_0 is as illustrated in Figure 3.

```
/tmp
└─ vbonemer
   ├─ slave.jar
   ├─ machines.txt
   ├─ splits
   │  └─ S0.txt
   ├─ maps
   │  └─ UM0.txt
   ├─ shuffles
   │  ├─ 2066512-tp-1a252-17.txt
   │  ├─ 2126094-tp-1a252-17.txt
   │  └─ 78973420-tp-1a252-17.txt
   ├─ shufflesreceived
   │  ├─ 2126094-tp-1a252-17.txt
   │  └─ 2126094-tp-1a252-19.txt
   └─ reduces
      └─ 2126094.txt
```

Figure 3: Resulting directory structure in remote machine `tp-1a252-17`

4.4 Remote command execution and file exchanges

In several steps of the algorithm, and in each of the packages, it is necessary either to execute commands in a remote machine or to send files to a remote machine. These were implemented using `ssh` and `scp`. In order to execute such commands in Java, the class `ProcessBuilder` was used.

`ProcessBuilder` allows for a process to be created and it executes the code given as parameters to the constructor. A related class also provides methods for waiting termination of a process with an optional timeout. This method is, evidently, a blocking method, as it needs to stop until the process has finished, which can be a problem when dealing with several machines in parallel. It can, though, be advantageous when actions have to take place in order, as is the case for creating a directory in a remote computer and then placing a file in it.

The code necessary to implement these functionalities would become very repetitive, as it includes not only waiting but, sometimes, checking the output of a program and often executing some code when one or all of these processes have finished.

In order to avoid repetition and reduce complexity, a class `ProcessRunner` was created. It receives one or many instances of `ProcessBuilder`, each associated with a process timeout, and a closure to be executed when the processes are finished. The processes associated with each process builder are then executed sequentially. If, for example, a directory needs to be created in a remote machine and then a file needs to be transferred to it, a first process builder can be instantiated to create the directory, another one for transferring the file, and they can be passed in the right order to the process runner, which guarantees they execute in order.

Most of the times though a series of tasks have to be executed not in one, but in many machines in parallel. To make it easy to do that, `ProcessRunner` implements the `Runnable` interface, so a `Thread` can be created with the instance and it executes the builders when the thread is started.

One problem with the process runner is that it was created to suit one of the packages and then adapted to the others in a case by case basis, which means that each package has its own version of a slightly modified process runner class. This is, of course, not ideal as it leads to code duplication and makes it harder to change the code. One possible improvement would be to create a single process runner that provides the functionality needed by each of the packages and extract it into a common package that all others would depend on.

4.5 Connection tests

Programs executed in the local machine (i.e. clean, deploy and master) have access to a file that lists which remote machines they should connect to. Those machines are not guaranteed to be available, though. For that reason, every local program has to test which machines it can reach before starting.

To resolve this problem a `ConnectionTester` class was created. It tests the machines by connecting to them via `ssh`, running the `hostname` program on them and checking if the output is the expected name. If all this is true and if it happens inside a given time, the file is considered to be available.

The tester receives the file that lists the machines and uses the `ProcessRunner` to test those in parallel, possibly printing their status to the standard output. When tests are done, it provides a list of all available machines, which can then be used normally. These tests were particularly useful, given that during development, several of the remote machines went offline for extended periods of time due to power outages in the region.

One problem that was encountered during development was that remote machines had a different behaviour when first connected to after some time. They would, in this situation, take an unusual long time to respond to requests, which would lead to a timeout in the connection tester and an empty list of available machines. If the code was ran again the problem wouldn't repeat itself, so a simple fix was implemented. After running the tests, if at least one machine is found to be offline, tests are ran once more and the second result is used instead of the first. Although simple, this was sufficient to deal with this anomaly.

5 Latency Analysis

The goal of MapReduce is to create highly paralellizable code, so, to test it, the proposed implementation is going to be ran with a same input file and varying number of machines. The chosen input file can be found at https://github.com/legifrance/Les-codes-en-vigueur/blob/master/sante_publique.txt. This is an 18 MB text file written in french.

Each configuration was executed five times and the average was used for reference. Tests were made with values ranging from 1 to 40 machines and all latencies were compared to the one obtained with one machine to calculate relative speedups (Equation 1). Figure 5 shows the obtained results and compares them to the expected speedups given by Amdahl's law (Equation 2) for different percentages of strictly parallel code.

The one-machine execution was done using the same program of all other executions. An alternative could be to create a sequential implementation to use as base, but because Amdahl's law considers only the parallelization in a single program and doesn't take into account other costs of parallelizing code (e.g. networking), using the same program was preferred. This introduces some losses in performance as, for example, the one-machine execution sends files to itself using `scp` during the *shuffle* phase, but this makes it closer to the other executions.

The *retrieve* phase of the algorithm consists in getting back all the files from the remote machines through scp and then merging them locally. This is a highly sequential part of the code and takes much longer than all other parts combined, so this phase was not considered in order to focus on the parts that can be parallelized (i.e. *map*, *shuffle* and *reduce*).

It can be seen in Figure 5 that the speedup increases as the number of machine is increased and it is very similar to what's expected by the Amdahl's law. The measured speedup follows really closely the line for the prediction of 90% parallelized code up to 25 machines, where it had a drop in performance.

When too many machines are involved there's more data being transmitted via the network, which could lead to a larger latency, but this was only expected to happen when the split size was smaller and, so, it wouldn't compensate to split any further. Because the original file has 18MB, with 40 machines the splits would still have around 700kB each, which still seems like a large enough file.

It could also be that some of the machines were taking longer to respond. The tests were ran on a fixed list of machines. They would be used in order, so when testing 10 machines, the first 10 of the list would be picked. This list was composed of available machines that had no active ssh sessions, but the tests took long enough for this to change during execution, which could have an impact on the results.

Each test actually takes longer than the considered latency because of the *retrieve* phase, as mentioned before. The first few tests took about 10 minutes to finish, while the ones involving 20 to 40 machines took 1 minute or less. Each configuration was executed 5 times and some of the tests had to be ran once again because the local connection had a significant drop in speed during the first series of tests.

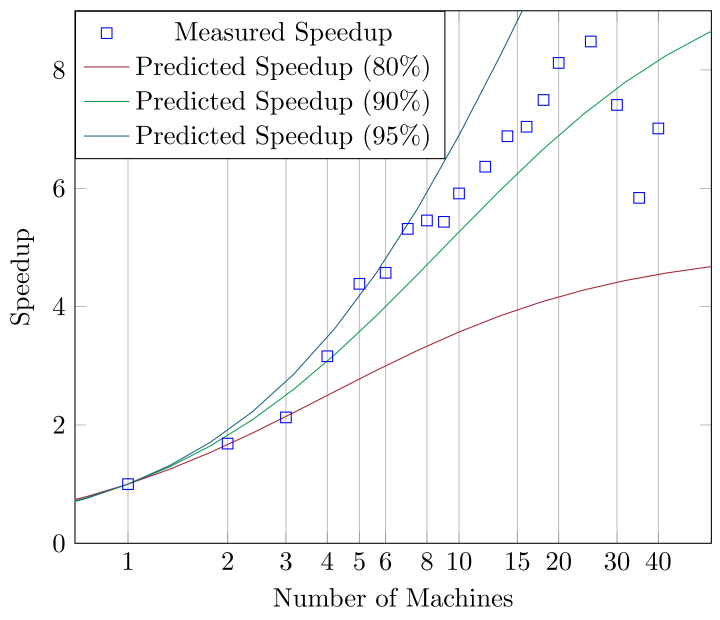

Figure 4: Experimental and theoretical speedups

This led to more than 12 hours of tests, during which other students could connect to some of the used machines impacting the results.

The fact that the one-machine reference was calculated from a parallel program also has a big impact on the resulting speedups. To show this, a sequential version of the solution to the word count problem was implemented and tested. Then, the latency obtained in the previous tests were used to calculate new speedups, but now taking this sequential implementation as a reference. It can be clearly seen in Figure 5 that this has a very big impact on performance – in fact, the sequential implementation is faster than parallel executions.

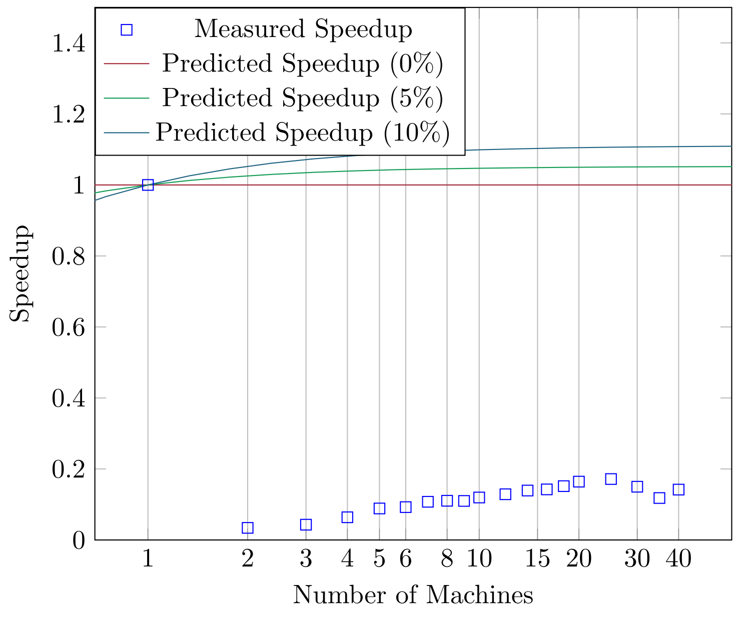

Figure 5: Experimental and theoretical speedups

As Figure 5 makes it clear, the sequential version of the code is much faster than the distributed one. This is due to the extra work performed to run parallel code, including the cost of sending files over the network. As the size of input files increases, it is expected that the linear implementation takes longer to complete and the cost of networking is then diluted down. So, with large enough files, the distributed implementations should end up being better even if this sequential code is taken as base case.

6 Conclusion

In this project an implementation of the MapReduce framework was implemented in Java; it was tested in a distributed environment using different configurations; the results were analysed in light of their latency; and the speedup of the distributed executions in relation to one with a single machine were compared with the expected speedups given by Amdahl's law.

It was possible to see that Amdahl's law gives a good prediction of the expected speedup of a real application when the same code is used in all tests. It was also shown that when sequential code is developed for the first case and used for comparison, the distributed version has worse performance for the tested file size.

It would be interesting to do further tests using machines that are completely dedicated to this task and that have a reliable communication between them. In this scenario, tests with larger files could provide more valuable information as to how the distributed algorithm compares to the sequential one.

In actual applications, there's only incentive to develop a distributed version of some code if it is better than a version optimized for sequential execution, so only the graph on Figure 5 would be of interest. The proposed extensions to the current tests would be essential to show how big the input should be before an investment in a distributed version of some algorithm is justified.