Project report: Cloud Computing and Big Data

T. Ragonneau* and G. Hugonnard†

Department of Computer Science and Applied Mathematics
2^{nd}-Master – Performance in Software, Media and Scientific Computing
Toulouse INP – E.N.S.E.E.I.H.T.

January 26, 2019

Contents

1 Introduction

The aim of this project is to implement an application scenario which illustrates the use of the following techniques:

- docker – it can be used to deploy a cluster of virtual machines on a laptop, but can also be used in a distributed setting with several laptops,

*High Performance Computing & Big Data
†Imaging & Multimedia

- spark – a spark infrastructure, including HDFS, a master and several slaves are deployed in the docker infrastructure.

The source code is available at the following link: https://github.com/TomRagonneau/PSMSC-DockerSparkHDFS. It refers to the courses of Cloud Computing and Big Dat of the MSc track Performance in Software, Media and Scientific Computing given at Toulouse INP-E.N.S.E.E.I.H.T. Eng. School and Paul Sabatier Faculty of Science and Engineering.

2 How to use the scripts?

2.1 Visualization of the executed scripts

We will firstly illustrate what the scripts do.

2.1.1 Locally

The figure (1) shows what the scripts are doing locally.

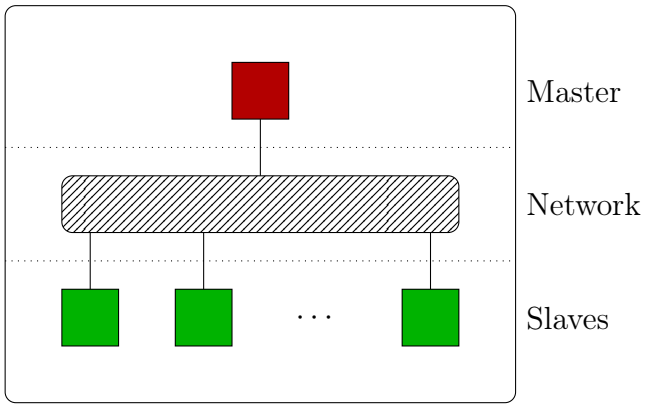

Figure 1: Local Script Diagram

A master and several slaves are communicating via a bridge network. Each slave represents a HDFS disk and a Spark computation node. The master manages the Spark application (it is the Hadoop Nanenode), and communicates with the slaves through a YARN application (resource manager), which takes care of the IP addresses and the ports of communication (there are no human intervention in the handling of the virtual machines' addresses). The source code is available at this addresse: https://github.com/TomRagonneau/PSMSC-DockerSparkHDFS/tree/master/local.

2.1.2 Remotely

The figure (2) show what the scripts are doing remotely.
On each guest host, the infrastructure is quite similar to the local version. However, when the master is launched, it opens a swarm tunnel, and puts an overlay network inside of it. Visually, the swarm is a tunnel and the overlay network is the road inside of the tunnel. When the swarm is initialized, the primary overlay network and all the containers are up and running on the manager, the worker builds the containers and runs them. When the first container of each worker try to connect to the overlay network that already exist inside the swarm, it grabs it locally. Then, all local slaves can connect to this network.

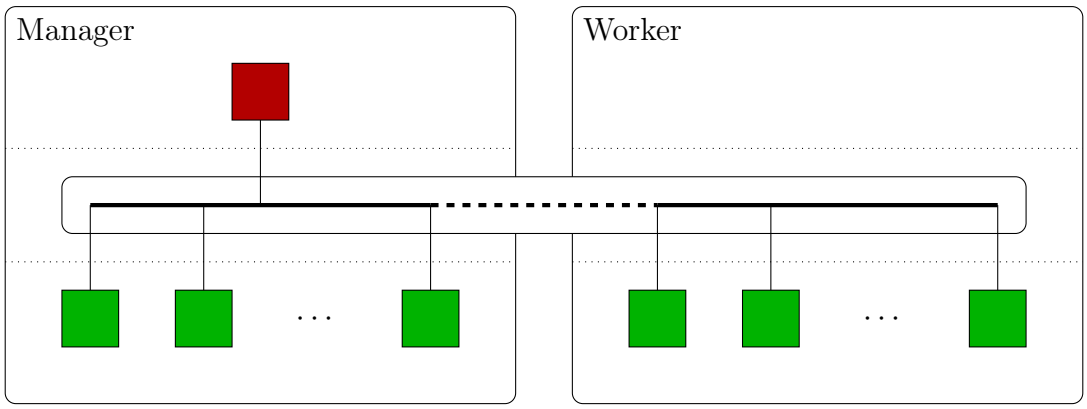

Figure 2: Remote Script Diagram

Once more, the manager communicates with all slaves (on every nodes) through a YARN application, and there are still no human intervention in the handling of the VMs.The source code of the remote version is available at this addresse: https://github.com/TomRagonneau/PSMSC-DockerSparkHDFS/tree/master/remote.

2.2 Run a WordCount scenario locally and on a cluster

Please read the README file on the repository to run a WordCount scenario: https://github.com/TomRagonneau/PSMSC-DockerSparkHDFS/blob/master/README.md

3 Performance analysis of parallelism

3.1 Analysis of the local version

3.1.1 Description of the experiment

For this first part, we only use one laptop which will host both master and slaves. We will analyze the time it took for a varying number of slaves, to count the words in a `.txt` file of approximately 2.7GB.
The number of logical cores in the laptop we use is 4.

3.1.2 Intuitive analysis

Firstly, before analyzing the results we have obtained, let's figure out the results we should, intuitively, obtain.
As the number of slaves increases, the word-count's time should decrease. This tendency should at some point slow down and reverse itself because of the time it takes for the virtual machines to communicate. This tendency is illustrated on Figure 3.

3.1.3 Results Analysis

Figure 4 shows the results we have obtained.
We can clearly distinguish the tendency we have intuitively predicted. As the number of slave containers increases the computation time of the word-count drastically decreases.

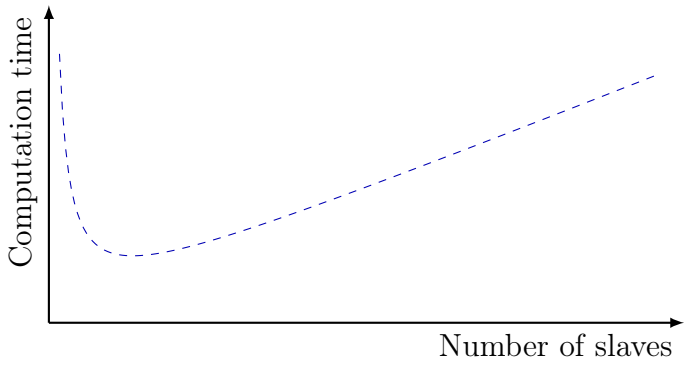

Figure 3: Intuitive tendency of local performance

We can also observe that the computation time reaches a plateau. We believe this plateau is specific to our laptop and if we had a more powerful computer, we would maybe be able to continue decreasing the computation time by adding more slaves.
Furthermore, we have 4 slave containers in the last value plus a master container, but we have only 4 cores. This is made possible by hyper-threading. The fact that it takes more time for 4 slave containers is directly correlated to hyper-threading.

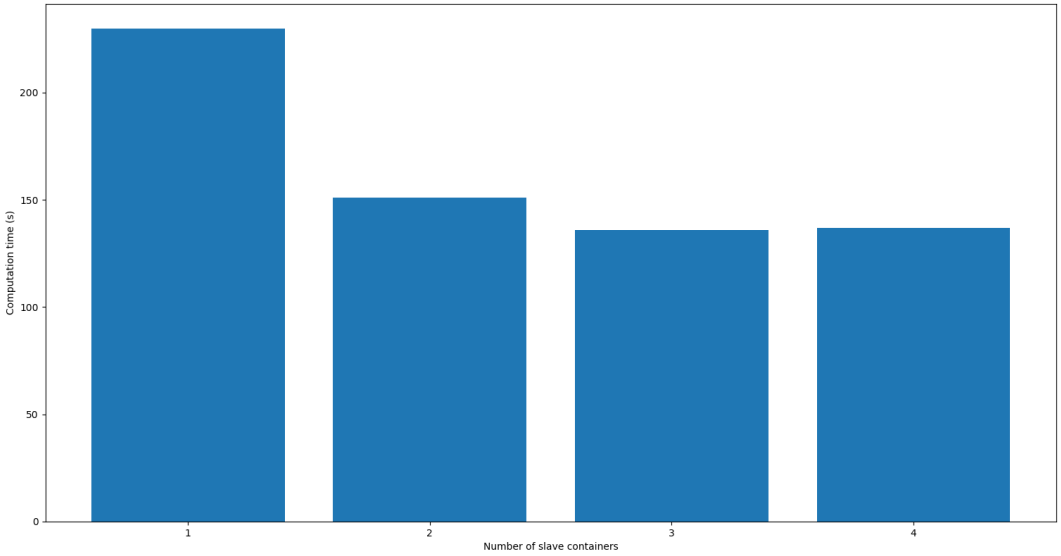

Figure 4: An analysis of the local version

3.2 Primary analysis of the remote version

3.2.1 Description of the experiment

We have at our disposition 2 laptops : 1 with 4 logical cores and 1 with 2 logical cores. The first laptop will be our manager and the second one will be a worker.
To analyse the performance of our system, we will vary the number of slaves on each machine and vary the size of the `.txt` file we count the words of, and compare the computation time to a baseline : a iterative word-count.
The 2 configurations of our test is :

- 2 guest hosts which both hosts 1 slave (Total : 2 slaves)

- 2 guest hosts : the first one with 3 slaves and the second one with 2 slaves (Total : 5 slaves)

We have tested our program with files ranging from 300MB to 5.4GB.

3.2.2 Iterative Word-count baseline

As a baseline, we wrote a simple iterative word-counter in python. Simply split the file with a space regex and count the words in a simple `for` loop.
This rudimentary method obviously has its limit. When the size of the analysed file gets to large, the memory of the computer becomes full and our program can simply not compute a file that is too large.

3.2.3 Results Analysis

Figure 5 (log graph) shows the different times it took for several configurations to count the words in a varying size of a `.txt` file.
Firstly, when we compare to the iterative baseline, we can see that for a small file, the baseline is faster but very quickly, parallelising the word-count becomes much more efficient.
When we compare the two configurations, we can clearly see that, for a small enough file, the first configuration is faster. We believe this is due to the communication times.
But for a file of more than about 2GB, the second configuration becomes faster, as the computation times of each slaves becomes larger, the communication times become less prominent.

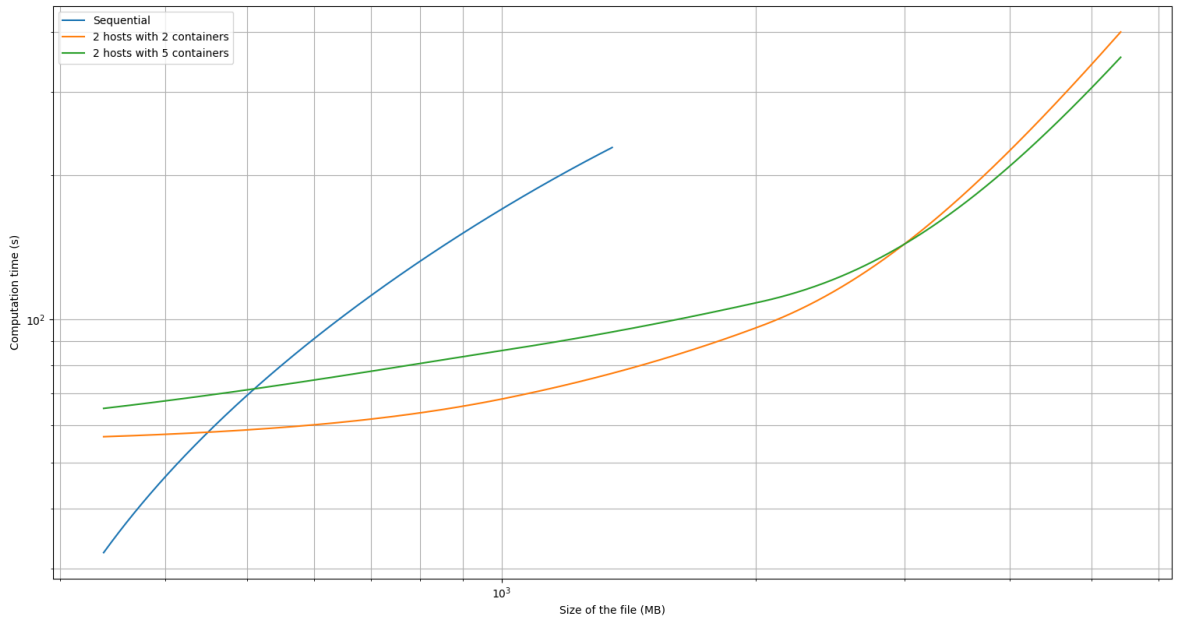

Figure 5: An analysis of the remote version

4 Conclusion

To conclude, throughout this project we have illustrated the improvements CPU and I/O parallelism by comparing the computation time of a simple word-count application using docker and spark.
Locally, by increasing the number of slaves, we have parallelized CPU on our laptop and a gain in performance is explicitly shown in Figure (4). By increasing the number of slave containers, the computation time drops.
Remotely, by increasing the number of hosts and of slave containers within them, we have combined a I/O and a CPU parralization and the computation time significantly drops and completely outperforms brute iterative word-counter. This tendency is even more explicit as the size of the file increases.
As a point of reference, the largest file we have tested is the first book of Harry Potter we have copied approximately 6400 times. And the time it took to count all those words is under 5 minutes.
Our laptops did not have enough memory to continue testing with even larger files but, intuitively, the gap in computation time between the first and second configuration should widen as the file becomes even larger.