Matrix Multiplication Benchmarking on Hadoop with Pig

Davide Azzalini, Matteo M. Fusi

Contents

1 Introduction

1.1 Purpose

This document is intended to analize the scaling capability of the Hadoop framework using Pig. Matrix multiplication will be employed as benchmark. The document will also describe how to set the environment up.

1.2 Document Organization

Section 1 describes the purpose of the document and furnishes other general informations. Section 2 describes the installation of the environment. Section 3 describes the test execution and how data have been collected. Section 4 exposes the results, our observations and what should be done in the future. Three appendices integrate the document.

1.3 Reference Links

Repository of the project https://github.com/fusiled/hadoop-pig-matrix-multiplication-benchmark

Ansible https://www.ansible.com/

ansible-cloudera-hadoop Repository by *sergevs* https://github.com/sergevs/ansible-cloudera-hadoop

Hadoop Pig https://pig.apache.org/

LyX (document production) https://www.lyx.org

UMLet (Figure 1 generation) http://www.umlet.com

MatPlotLib (graph generation) http://matplotlib.org/

OpenStack https://www.openstack.org/

PoliCloud http://policloud.polimi.it/

Raw Yarn logs and test matrices archive (7z format)
https://drive.google.com/file/d/0B8vc7cpy6aOcRGVDTHN2ZlVTbnM/view?usp=sharing

2 Setting the Environment Up

2.1 Architecture

On top of OpenStack we ran a Master along with a variable number of workers. Master was set as namenode with yarn and Pig running on it. Master contributed also to job execution. Figure 1 shows how the resulting architecture at the end of the installation.

2.1.1 Available Resources

The Hadoop cluster was deployed on the PoliCloud platform, the cloud structure hosted and managed by the Politecnico di Milano. PoliCloud is a IaaS built on top of OpenStack. The resources available to us were the following:

- Max number of instances: 8
- Number of virtual CPUs (VCPUs): 16
- Available RAM: 21 GB

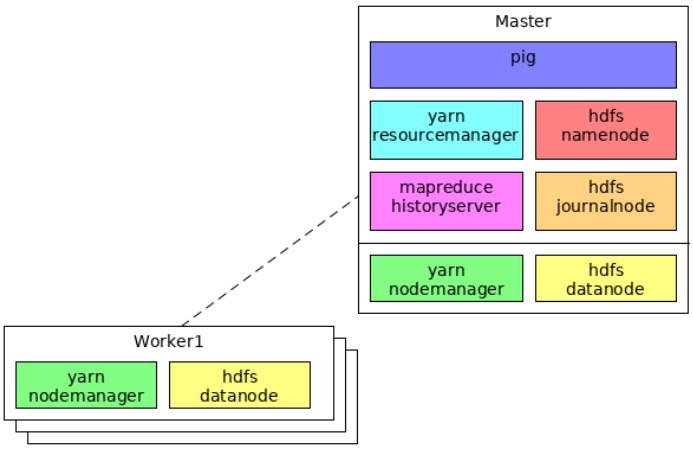

Figure 1: Deployment Diagram of the cluster

- Available persistent storage: 500 GB
- Max number of volumes: 10

2.2 Installation

This Section describes the different steps involved in the installation of the cluster on the PoliCloud environment. The entry point of PoliCloud for users is a dashboard reachable by the web from which you can easily launch the instances you need. After you've launched your instances you can connect to them via SSH and begin the installation of the cluster.

2.2.1 Requirements

A minimal knowledge of the Unix shell environment is required to understand what the scripts do. A knowledge of SSH and how ssh works is also needed. The scripts are anyway thoroughly commented. A knowledge of the RedHat/Fedora systems is recommended but not required in order to understand what the *yum* (the package manager of CentOS) command does.

2.2.2 Run the Instances

To launch an instance you need to select a flavour, an OS image and a SSH public key for the access. Flavours are a sort of predefined "*virtual hardware configurations*". You can select only predefined flavours, but you can use any valid OS image. You can also save instance images that can be reused in the future. Everyone of our instances ran CentOS 6 as operating system. The image of CentOS 6 is made available by the PoliCloud Environment. We allocated our resources in the following way:

Master A single instance with flavour *compute.large*, which means 8 VCPUs, 4GB of RAM and 80GB of disk.

Worker Multiple instances (from 2 to 7), each one with *general.small* flavour, which means 1 VCPUs, 2GB of RAM and 20 GB of disk.

It's important to highlight that, due to a bug, during the initializiation, the instances were not always able to load the key for the SSH access. We solved this issue by trial: we kept launching instances until one loaded the key correctly, then we replicated that instance several times. The image offered

by PoliCloud had also the firewall *iptables* running by default. This feature must be disabled in order for the nodes to be able to comunicate properly, use Algorithm 1 to do it.

Algorithm 1 How to disable the *iptables* firewall

```
#!/bin/bash
sudo su -
/etc/init.d/iptables save
/etc/init.d/iptables stop
chkconfig iptables off
exit
```

2.2.3 Installation Script

We used the a tool created by *sergevs*, a GitHub user, that is based on Ansible (Check the link at Section *1.3* for further details). You can easily clone the linked repository with git. The requirements for every machine are:

1. CentOS 6
2. Oracle Java installed
3. Access to the Cloudera 5 yum Repository. You can find the yum reference at the following path: https://archive.cloudera.com/cdh5/redhat/5/x86_64/cdh/cloudera-cdh5.repo
   Just copy the dowloaded file into the */etc/yum.repos.d* directory and type *sudo yum update* with root privileges.
4. SSH key passwordless authentication must be configured for root account for all target hosts. The default configuration of PoliCloud was already fine, we just needed to add the access key to the known hosts of the root user.
5. SeLinux must be disabled
6. Ansible and unzip on the machine that runs the script. We ran the script on Master node. It is required to set *remote_user = root* in */etc/ansible/ansible.cfg* file. Moreover, logging can be enabled setting the correct flag in the ansible.cfg file.

Algorithm 2 must be run on every machine to execute points 2,3,4 and 5. The reboot is needed to let the SeLinux deactivation apply. Ansible must be installed manually. It is in the *epel-release* repository. This repository was added by the above script, so, once you have run the script, on Master type the following command:

```
sudo yum install ansible -y
```

and set the *remote_user* flag as explained before.

2.2.4 Memory Partitions and Volumes

The available CentOS image created just a single partition of 8GB, which means that the disk associated to an instance was not fully utilized. This amount of memory was obviously not sufficient for our tests, so we needed to add some storage memory to the instances. We employed the volume utility of PoliCloud that allows you to create virtual disks. The property of these disks is that the memory is persistent: what is saved on a volume "*persists*" until the disk is deleted (a useful feature, although not needed in our case)[1]. We attached a volume of 50GB to each instance. We also created a swap

[1]Default disks of the instances don't have the persistent property. That's why they're called *ephemeral*.

Algorithm 2 Environment setup script

```
#!/bin/bash
#enter root environment
sudo su -
#add misc epel-release
yum install epel-release unzip wget -y
#download and install cloudera repo reference
wget -nc https://archive.cloudera.com/cdh5/redhat/6/x86_64/cdh/cloudera-cdh5.repo
mv -f cloudera-cdh5.repo /etc/yum.repos.d/cloudera-cdh5.repo yum update -y
#get Oracle Java and install it
wget -nc --no-cookies --no-check-certificate --header "Cookie: gpw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie" "http://download.oracle.com/otn-pub/java/jdk/8u121-b13/e9e7ea248e2c4826b92b3f075a80e441/jdk-8u121-linux-x64.rpm"
yum localinstall jdk-8u121-linux-x64.rpm -y
#enable root login with centos user key
#this is a simple turnaround: copy the authorized keys of the user centos and make them ok also for root user
/bin/cp -f /home/centos/.ssh/authorized_keys ~/.ssh/authorized_keys
#restart ssh deamon
service sshd restart
#disable selinux
echo "SELINUX=disabled" > /etc/selinux/config
echo "SELINUXTYPE=targeted" >> /etc/selinux/config
#reboot the system
reboot
```

partition for each instance exploiting part of the unoccupied disk by using *cfdisk* (a partition editor for Linux). Algorithm 3,4,5 and 6 perform these tasks. Algorithm 3 and 4 must be executed on Master and also on workers, then run Algorithm 5 on Master only and Algorithm 6 on every worker. Note that algorithms 5 and 6 are intended as a continuation of Algorithm 4. Now the instances should

Algorithm 3 Memory setup on an instance (Part 1)

```
#!/bin/bash
#from PoliCloud dashboard create a volume and associate
# it to the instance
#enter root environment
sudo su -
#call cfdisk on the virtual disk of the instance
#create a partition that will become the swap (/dev/sda2)
#and one partition with the remaining space (/dev/sda3)
cfdisk /dev/sda
#call cfdisk on the volume
#for Master just create one big partition (/dev/sdb1)
#for workers create 2 paritions:
# - 1 for yarn 28GB big (/dev/sdb1)
# - 1 for hdfs with the remaining space (/dev/sdb2)
cfdisk /dev/sdb
#reboot to detect partitions
reboot
```

Algorithm 4 Memory setup on an instance (Part 2, after the reboot, common part)

```
#!bin/bash
#!!WARNING!!
#!! After the reboot the name of disks may be swapped:
#!! /dev/sda may become /dev/sdb and viceversa
#!! Always check this fact. If the names are swapped
#!! change the below commands where needed
#!! Now we assume that /dev/sda and /dev/sdb are the
#!! same as the Algorithm 3, Part1
#enter root environment
sudo su -
#format swap and mount it
mkswap /dev/sda2
swapon /dev/sda2
```

be ready to install the Hadoop environment.

WARNING ON MOUNTED PARTITIONS The partitions are mounted, but we didn't altered the */etc/fstab*, which is the file that tells the OS which partitions mount at boot time. This means that after every reboot the partitions must be remounted.

Algorithm 5 Memory setup on Master (Part 2, after the reboot)

```
#!/bin/bash
#CHECK Algorithm 4 WARNING
#format volume and mount it
#this partition will be used for hdfs
mkfs.ext4 /dev/sdb1
#create path
mkdir /var/lib/hadoop-hdfs
#mount the partition ath the new path
mount /dev/sdb1 /var/lib/hadoop-hdfs
#do the same for /dev/sda3 and /var/lib/hadoop-yarn
mkfs.ext4 /dev/sda3
mkdir /var/lib/hadoop-yarn
mount /dev/sda3 /var/lib/hadoop-yarn
#exit root environment
exit
```

Algorithm 6 Memory setup on a worker (Part 2, after the reboot)

```
#!/bin/bash
#CHECK Algorithm 4 WARNING
#format volume partitions
mkfs.ext4 /dev/sdb1
mffs.ext4 /dev/sdb2
#create paths
mkdir /var/lib/hadoop-hdfs
mkdir /var/lib/hadoop-yarn
#mount the partitions
mount /dev/sdb1 /var/lib/hadoop-hdfs
mount /dev/sdb2 /var/lib/hadoop-yarn
#exit root environment
exit
```

2.2.5 Ansible

Ansible is an open-source automation engine that automates cloud provisioning, configuration management, and application deployment. Once installed on a control node, Ansible, which is an agentless architecture, connects to a managed node through the default OpenSSH connection type. You can define a set of rules that can be executed in a specified order and when specific conditions are met. The set of tasks and rules are specified in a playbook. We defined tasks and rules by editing a file called *hosts*. In this file you can specify the hadoop roles of each machine (namenode, datanode) and which software you want to install on it (zookeeper, spark...). We installed a simple cluster with Master as namenode and datanode (it had enough computationl power to perform both tasks) and the workers as datanodes. The yarn resource manager was installed on Master. It is required to fix the file *group_vars/all* that contains some useful variables. The java path it is not correct so you must manually edit it. If you ran the provided script you just have to change the first two uncommented lines into the following ones:

```
java_package: java-1.8.0
java_home: /usr/java/jdk1.8.0_121/
```

As described in the README of *sergevs* repository, run the script typing this command from the root directory of the repository:

```
ansible-playbook -i hosts site.yaml
```

If the installation succedes, then you can install Pig.

2.2.6 Pig

Apache Pig is a high-level platform for creating programs that run on Apache Hadoop. The language for this platform is called Pig Latin. Pig can execute its Hadoop jobs in MapReduce, Apache Tez, or Apache Spark. Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for RDBMSs. Pig was not included in the playbook so we installed it manually after the playbook was executed, using the Cloudera 5 Repository. Just type the following command on Master:

```
sudo yum install pig
```

2.3 End of the Installation

At the end of the installation you should have an architecture that corresponds to the deployment diagram shown in Figure 1. To launch any task on the cluster just SSH the Master and then switch to *hdfs* user environment. To enter in such environment type the following command:

```
sudo su - hdfs
```

3 Test Execution and Data Collection

As we have already stated before, the purpose of this work is to acquire a deeper awareness on how our cloud architecture scale. To achieve our goal we designed a test that consisted in repeatedly run a benchmark algorithm while progressively adding more resources to our cluster. We started with the just Master and two workers and we kept adding one worker at a time up to eight. The benchmark algorithm we chose is a classical matrix-multiplication algorithm. The algorithm multiplies the same square matrix by itself. In addition to scaling on the dimension of the cluster we decided that would have been interesting also to scale on the dimension of the matrix, we chose a 500 rows by 500 column matrix before and a 1000 rows by 1000 column then. Note that we always used the same two matrices for every test, for consistency reasons. So, to sum up, we ran the following tests:

1. 500 by 500 matrix on a 3-node cluster
2. 500 by 500 matrix on a 4-node cluster
3. 500 by 500 matrix on a 5-node cluster
4. 500 by 500 matrix on a 6-node cluster
5. 500 by 500 matrix on a 7-node cluster
6. 500 by 500 matrix on a 8-node cluster
7. 1000 by 1000 matrix on a 3-node cluster
8. 1000 by 1000 matrix on a 4-node cluster
9. 1000 by 1000 matrix on a 5-node cluster
10. 1000 by 1000 matrix on a 6-node cluster

11. 1000 by 1000 matrix on a 7-node cluster
12. 1000 by 1000 matrix on a 8-node cluster

In order for our results to be consistent we run each of the just mentioned test 10 times. Unfortunately, due to the significant amount of time required, we hadn't been able to run each test 30 times, as the common practice would suggest.

3.1 Generation of the Matrices

We used a simple C program to generate a dense square matrix of size *DIMxDIM*. This program outputs a matrix in "row, column, value" format: *<row \t column \t value>*. The program uses OpenMP to speed up the generation of the matrix. To compile the source just type the following command:

```
cc -fopenmp -DDIM=<matrix_side_length> genMatParallel.c -o genMatParallel
```

The *-fopenmp* flag enables the parallelism. If you don't want use OpenMp you can simply compile the program without the just specified flag. Remember to change *<matrix_side_length>* with a integer value. After the compilation run the executable to generate the *outMatrix.txt* file, which contains the matrix. In our case we set DIM to 500 at first, and then to 1000.

Algorithm 7 Matrix generation source

```
#ifndef DIM
#define DIM 10
#endif

#ifndef FILE_PATH
#define FILE_PATH "outMatrix.txt"
#endif

#include <stdio.h>
#include <omp.h>
#include <time.h>
#include <stdlib.h>
int main()
{
        FILE * fp = fopen(FILE_PATH,"w");
        srand( (unsigned int) time(NULL));
        int i,j;
        #pragma omp parallel for collapse(2) private(i,j)
        for(i=0; i<DIM; i++)
        {
          for(j=0; j<DIM; j++)
                {
                        int random = rand();
                        fprintf(fp, "%d\t%d\t%d\n", i,j, random);
                }
         }
       fclose(fp);
}
```

3.2 The Test Script

The Pig Latin script shown in Algorithm 8 performs the multiplication. Since our goal was to run a benchmark and not to actually multiply matrices we didn't bother to choose a particularly efficient implementaion and thus we went for the classical naive school-book algorithm with time complexity $O(n^3)$.

Algorithm 8 The test script in Pig Latin

```
1 SET default_parallel n;
2 matrix1 = LOAD 'mat1' AS (row,col,value);
3 matrix2 = LOAD 'mat1' AS (row,col,value);
4 A = JOIN matrix1 BY col FULL OUTER, matrix2 BY row;
5 B = FOREACH A GENERATE matrix1::row AS m1r, matrix2::col AS m2c, (matrix1
      ::value)*(matrix2::value) AS value;
6 C = GROUP B BY (m1r, m2c);
7 multiplied_matrices = FOREACH C GENERATE group.$0 as row, group.$1 as col
      , SUM(B.value) AS val;
8 store multiplied_matrices into 'matmul_out';
```

Algorithm 9 is just a wrapper of Algorithm 8: It iterates 10 times (this is the number of repetitions for each test case) for each matrix (so once for 500 and once for 1000) and it prepares the environment by cleaning the hdfs, copying the matrix and launching Algorithm 8. Algorithm 9 also generates a file where it stores for each line the timestamp of the begin, the matrix dimension, the number of nodes and total execution time. This will beecome helpful for the log processing.

Algorithm 9 Wrapper of the basic test script

```
#!/bin/bash
#init variables
matrices_dir="matrices_dir"
HDFS_MATRIX_NAME="mat1"
PIG_EXEC="matmul.pig"
HDFS_OUT_DIR="matmul_out"
output_dir="test_output"
all_time_storage="$output_dir/pig_times.txt"
time_buf_file="tmp/time_buf"
#get the number of workers
n_datanodes=`hdfs dfsadmin -report | grep Live | tr -d -c 0-9`
#do the test for every matrix in $matrices_dir
for matrix in `ls $matrices_dir`
do
    #do the test 10 times
    for iter in `seq 1 10`
    do
        #PREPARE THE ENVIRONMENT
        timestamp=`date -Iseconds`
        output_file=$output_dir/$matrix-$timestamp
        #copy test matrix ind hdfs
        hadoop fs -copyFromLocal -f $matrices_dir/$matrix $HDFS_MATRIX_NAME
        #clean old results that could be present in hdfs
        hadoop fs -rm -r -f $HDFS_OUT_DIR
        #RUN THE TEST
        #use unix time command, not the one of bash to estimate pig
            exec_time
        #redirect pig output in $output_file
        /usr/bin/time -f %e -o $time_buf_file pig $PIG_EXEC |& tee
            $output_file
        pig_time=$(cat $time_buf_file)
        #save exec_time in pig_times.txt
        echo "$timestamp,$matrix,$n_datanodes,$pig_time" >>
            $all_time_storage
    done
done
```

3.3 Log Processing

Algorithm 10 has been used to mine the yarn logs of each node. Raw logs can be found at the link cited at Section 1.3. This script generates an output file at the path specified by *$log_times* where it stores app_id of a job, the name of the node and the execution time. The script mines the *app_id*s from the logs obtained by the pig executions contained inf the directory *$pig_log_dir*. For every *app_id* found it searches for occurrenceof it in every yarn log. A yarn log is uniquely associated to a node. From the timestamp of the first and the last occurence you can easily compute the execution time. Check the code and the comments to understand better how Algorithm 10 works.

NOTE Algorithm 10 were run with AWK=4.1.4. Different versions may not work.

Algorithm 10 Yarn miner script

```
#!/bin/bash
#Recognize dates with a regex
DATE_REGEX="2017-[0-1][0-9]-[0-3][0-9]\s
    [0-2][0-9]\:[0-5][0-9]\:[0-5][0-9]"
#path variables
yarn_log_dir="yarn_logs"
pig_log_dir="pig_logs"
tmp_file="tmp.txt"
log_times="log_times.csv"
for pig_log in `ls $pig_log_dir`
do
    #app_ids have the following regex: [0-9]{13}_[0-9]{4}
    #use awk to find occurrences in $pig_log_dir/$pig_log.
    #remove duplicates with pipe combination of sort and unique
    app_id_ar=`awk 'match($0,/[0-9]{13}_[0-9]{4}/){ print substr($0,RSTART
        ,RLENGTH)}' ./"$pig_log_dir/$pig_log" | sort | uniq`
    for app_id in $app_id_ar
    do
        for yarn_worker_log in `ls $yarn_log_dir`
        do
            #just get timestamp of the first and the last occurrence
            #of $app_id, convert them in seconds and substract them
            #to obtain the execution time
            grep $app_id $yarn_log_dir/$yarn_worker_log > $tmp_file
            first_occ=`head $tmp_file -n 1`
            first_time=`echo "$first_occ" | grep -o $DATE_REGEX`
            first_time_sec=`date --date="$first_time" +%s`
            last_time=`tail $tmp_file -n 1 | grep -o $DATE_REGEX`
            last_time_sec=`date --date="$last_time" +%s`
            exec_time=`expr $last_time_sec - $first_time_sec`
            echo "$app_id,$yarn_worker_log,$exec_time" >> $log_times
        done
    done
done
rm $tmp_file
```

3.4 Log Files

On the GitHub repository of the project, in addition to the original log files, we have pushed also some other files that contains a cleaner and more human readable version of the original log files. These files are the following:

pig_matrix-mul_coarse.csv contains extremely aggregated data. There is one row for each group of test (i.e. one row for each cluster dimension for each matrix dimension). In addition to the mean total time we have included also some statistics such as the median and the standard deviation.

pig_matrix-mul_fine.csv contains aggregated data for each test. For each of test 120 we ran we have reported some statistics for each one of the two jobs that compose each test. Check Appendix B for instructions on how to interpret the header.

pig_matrix-mul_extra-fine.csv contains a finer graned version of *pig_matrix-mul_fine.csv*. The execution times on each worker are also reported for each job of each test.

The above described files are the result of the combination of pig_logs (contained in the linked compressed folder), *pig_times.txt* and *log_times.csv* (contained in the repository *log* folder).

3.5 Results

Figure 2 shows the total execution times with both matrices w.r.t. the different cluster dimensions. We decided to use a boxplot for the representation because in a single plot are included a lot of information about the distribution of the results along the test. Representations of execution times on each worker

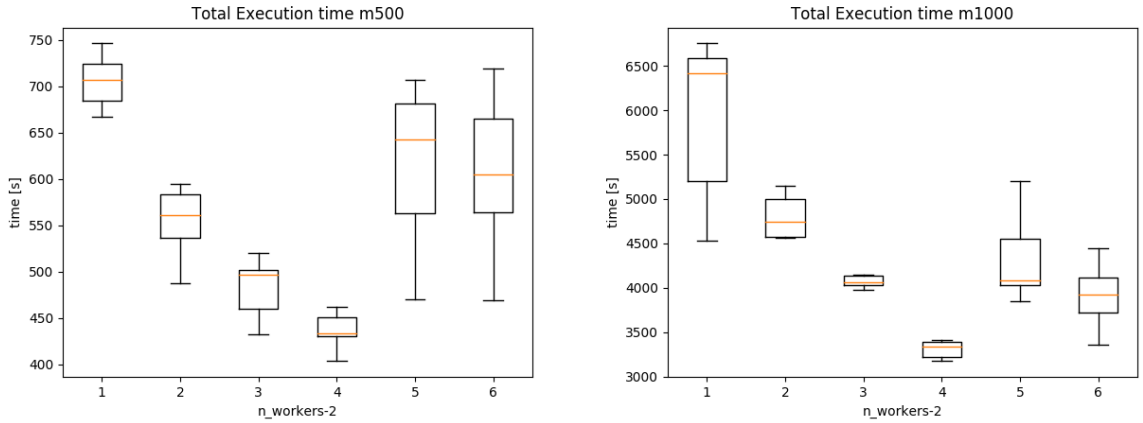

Figure 2: Boxplot of the results

can be found in Appendix A. Figure 3 shows the speed-up obtained from increasing the datanodes from 3 to 8. We considered as "base case" the execution time of the algorithm with 3 nodes. Figure 4 has been obtained by overlapping the graphs contained in Appendix A. It is easy to observe that, for each job, lines are "well overlapped" for the 1000x1000 matrix, which means that the time required by each job is equal on each node. The graph of the tests with the matrix 500x500 is more confusing: this means that on each machine the execution time per job is more variable.

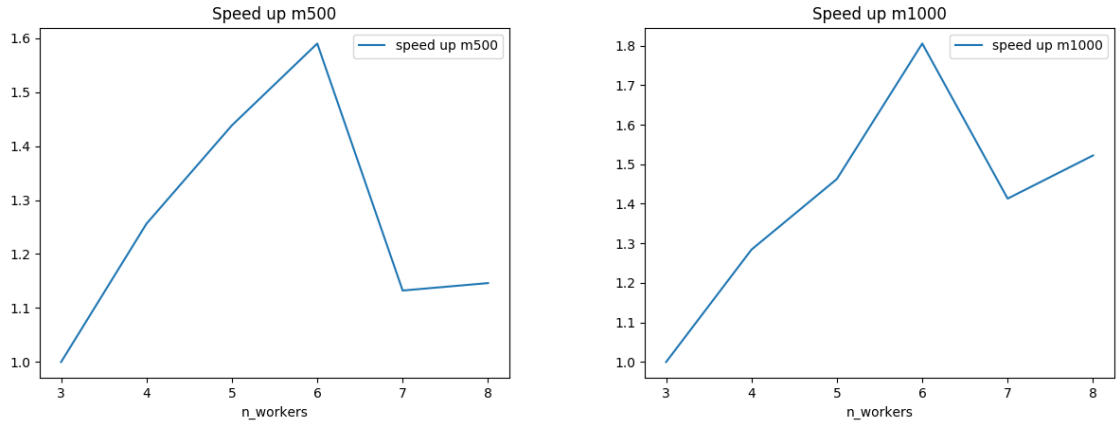

Figure 3: Speed-up 1000x1000 matrix

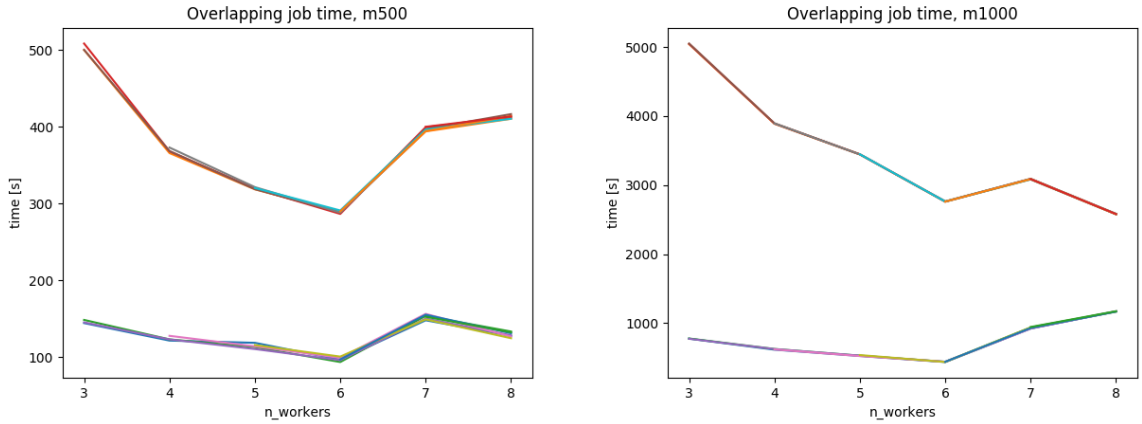

Figure 4: Graph obtained by overlapping of the graphs of the Appendix A

4 Conclusions

As one would expect, by increasing the number of working nodes the time required for the algorithm to complete decreases. In our opinion, the interruption of this decreasing trend after the addition of the seventh worker can be explained with the fact that we don't really know how OpenStack allocates the instances and where these instances are physically located. The fact that this worsening is more empasized with the 500 by 500 matrix can be addressed by the different ratio between the time required for the management of the workers and the time of actual computation. As future work we suggest to retake the tests, in fact, the degradation in the performance could be also due to a temporal excess load on the PoliCloud at that time. We would also suggest to continue adding more machines, beyond eight, and see how the scaling behaves. We found PoliCloud very easy to use, although some bugs on the infrastrucutre slowed down our work: in addition to the bug on the ssh key mentioned above, we had some problems also with the elimination of some instances that didn't seem to be willing to be eliminated. We had to ask to the system adminitsrator to delete them. The main issues we faced during the test phase were related to the correct sizing of the storage space of the instances w.r.t the size of the test matrices beacuse when the join is computed a lot of memory space is needed. A matrix too smal wouldn't have been significant for the test, but a too big one would have filled the partition

of the nodes making the them to become unhealthy or filling the swap, resulting in an arrest of the entire test case. *sergevs*' tool is quite powerful. We just used a small part of its potential.

A Execution Time per Job Graphs

Each figure shows the average execution time on a node w.r.t. the dimension of the cluster. The times for each of the two jobs that compose a test are represented separately in the plot. Plots realted to worker3 start from 4 workers up to 8 because Worker3 is used only when the dimension of the cluster is greater than 3. The same holds also for the other workers up to Worker8. The graphs for Worker7 are not significant because they basically consist of two points, so they've been replaced by Table 1.

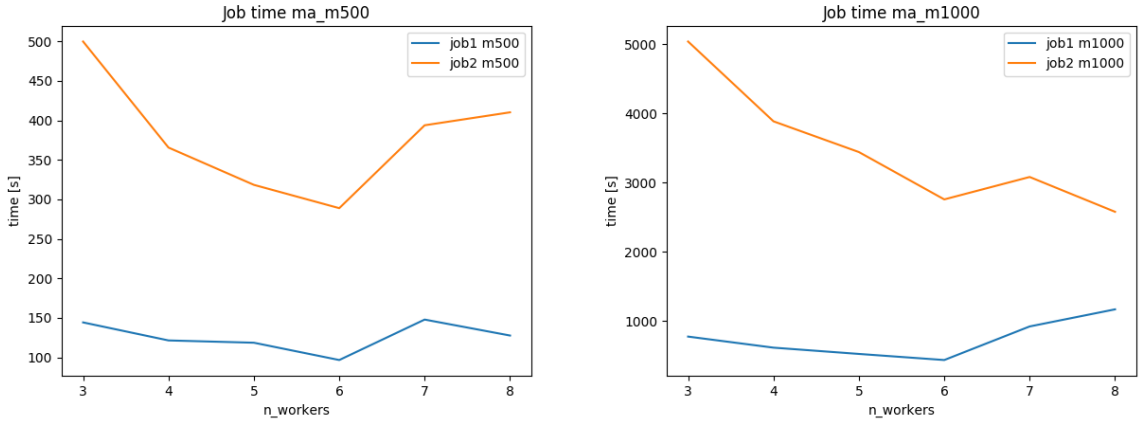

Figure 5: Execution time per job on Master

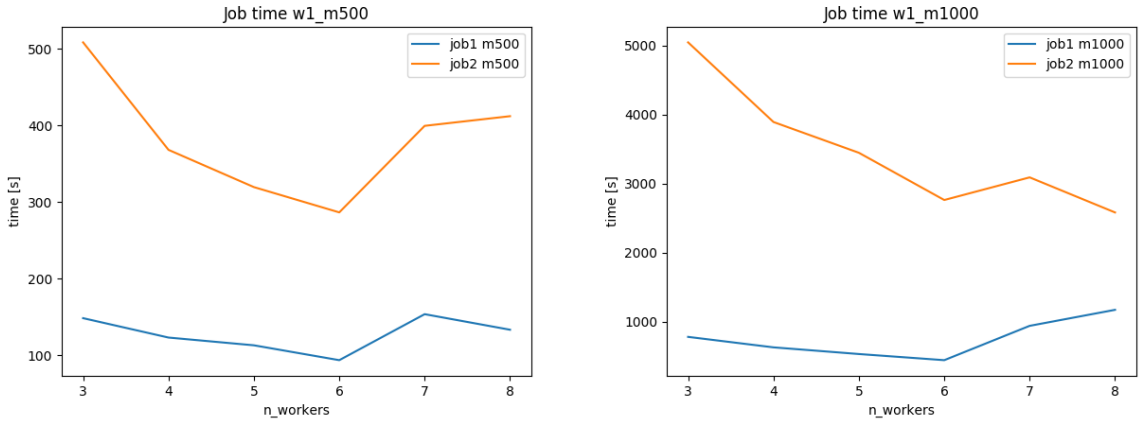

Figure 6: Execution time per job on Worker1

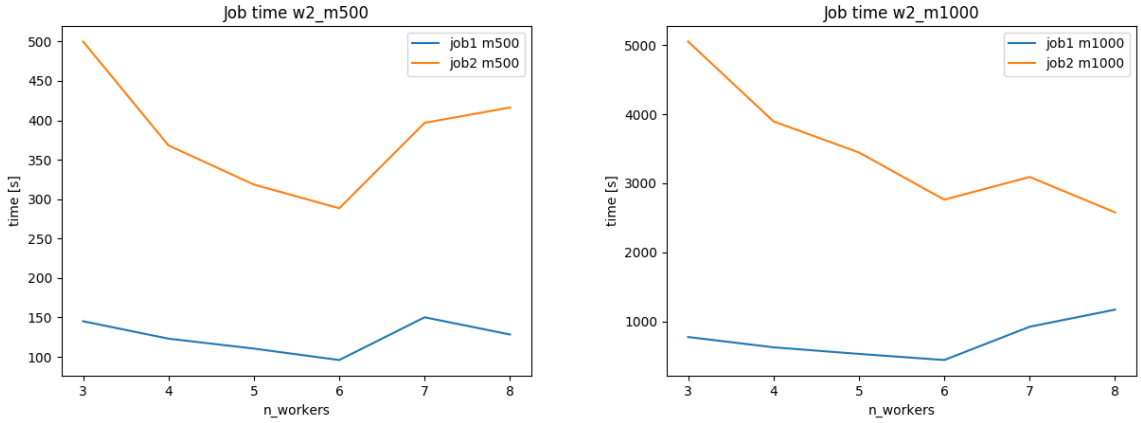

Figure 7: Execution time per job on Worker2

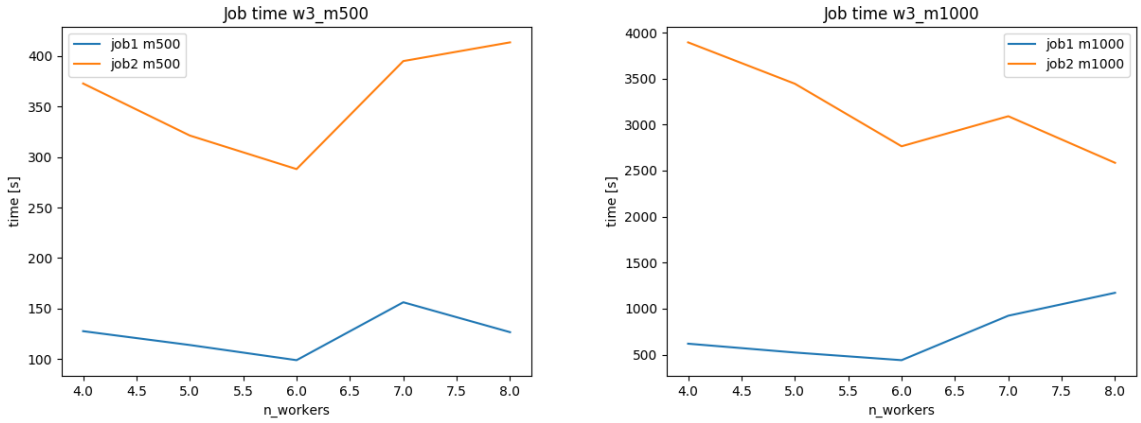

Figure 8: Execution time per job on Worker3

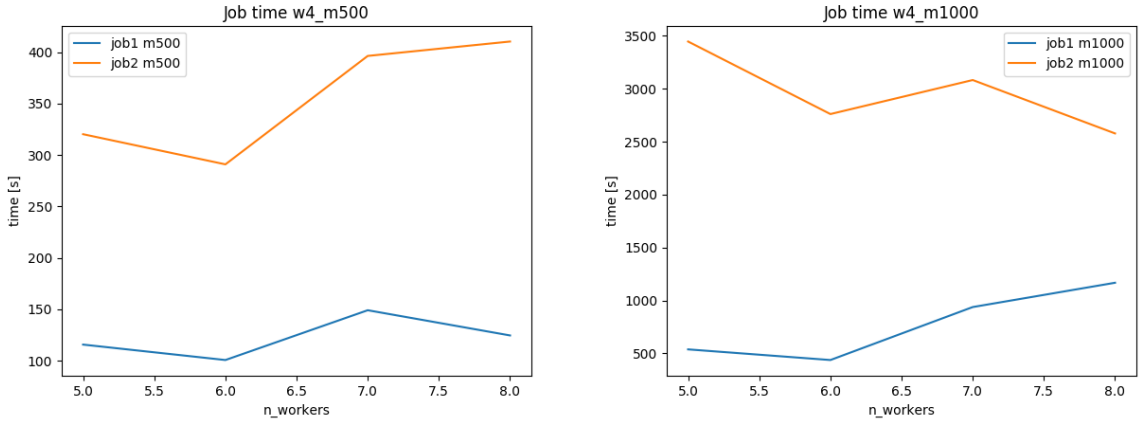

Figure 9: Execution time per job on Worker4

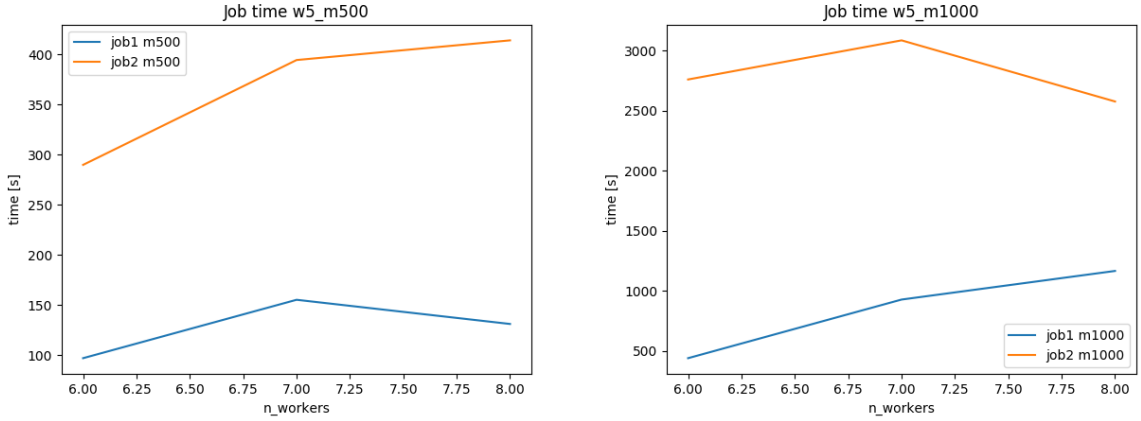

Figure 10: Execution time per job on Worker5

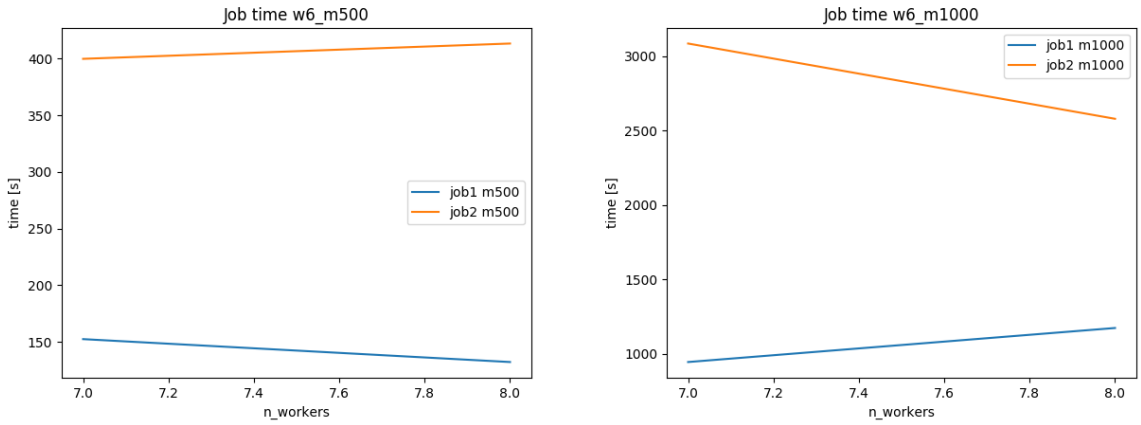

Figure 11: Execution time per job on Worker6

500	
job	time[s]
1	124.6
2	410.4

1000	
job	time[s]
1	1170.5
2	2580.8

Table 1: Execution time per job on Worker7

B Header Legend

The header of the file *pig_ matrix-mul_ fine.csv* cited in Section 3.4 is composed as follows:

- Nodes: the number of datanodes we were using;
- Dim: the dimension of the matrix (either 500 or 1000);
- Test: a number that goes from 1 to 10 that represents the number of the test for a given cluster dimension and matrix dimension;
- Job: since each test is composed of two jobs this value can be either 1 or 2;
- JobId: a unique job identification number;
- Maps: number of maps of this job (this value depends on the dimension of the input);
- Reduces: number of reduces of this job (this value depends on the number datanodes);
- MaxMapTime: time of the map that took more time;
- MinMapTIme: time of the map that took less time;
- AvgMapTime: average map time;
- MedianMapTime: median map time;
- MaxReduceTime: time of the reduce that took more time;
- MinReduceTime: time of the reduce that took less time;

- AvgReduceTime: average reduce time;
- MedianReducetime: median reduce time;
- Alias: ”variables” of the pig latin script used for this job;
- Feature: operations performed in this job;
- Date;
- Time: total time for the test (job1 + job2)

C Structure of the Repository

The *doc* folder contains this document and the images in it. The *log* folder contains the logs in *csv* format. The *src* folder contains all the scripts used in this project. Scripts are well described in this document and they're organized in subfolders.