INF 727

Implémentation de Hadoop MapReduce "from scratch"

Jean-Bastien Sarda
MS BGD

Introduction :

Le projet réalisé permet de faire un wordcount distribué avec des options telles que la possibilité de fractionner le fichier d'entrée en plusieurs splits par worker, et la gestion des crash machines. Le dossier placé dans /tmp du master est structuré de la manière suivante :

- Script MASTER.py : commandes assurées par le master,
- fichier HOSTS.txt : liste des hostnames des workers désirés. De ce fichier découle le fichier machines.txt créé lors de l'opération, dans lequel sont listés les hostnames des machines distantes qui dont la connexion a été vérifiée.
- Script SLAVE.py : contient les commandes lancées sur chaque worker pour chaque phase (map, shuffle, reduce),
- Scripts CLEAN.py et DEPLOY.py : permettent de supprimer et de recréer tout dossier de travail sur les workers. Ces scripts sont automatiquement lancés lors de l'initialisation du programme.
- Dossier input contenant le fichier input.txt,
- Dossier output contenant le fichier output.txt, lequel répertorie par ordre décroissant les mots comptés et leurs occurrences. Ce dossier est créé pendant la réalisation du programme.

Commande pour lancer le programme : placer les fichiers dans /tmp/sarda-20 du master, puis lancer la commande suivante :

- *python3 MASTER.py HOSTS.txt [nb de splits par worker]*

Le rapport ci-dessous reprend de manière chronologique la réalisation du projet et liste les obstacles rencontrés et les mesures prises pour les contourner. A partir de la page 5, le rapport présente une étude des meilleurs paramètres sur le programme achevé, la fiabilité du programme étant mesurée par le nombre de « the » comptabilisés (le résultat est automatiquement retourné à la fin du programme).

Le code est disponible l'adresse *https://github.com/jbSarda/INF727*

I. Étape 10

A partir de l'étape 10, je parallélise les tâches par multithreading (fonction *threading.threads*). L'envoi des split distincts aux aves par SSH se fait alors conjointement.

D'autre part, les instructions à lancer en commande linux via le script python se font par *subprocess.check_output*, fonction qui permet d'attraper la sortie erreur au besoin (sinon renvoie *None*).

Chaque script python débute, outre l'appel des librairies, par les définitions

- du chemin (ici */tmp/sarda-20/*),
- du *hostname* de la machine qui lance le script,
- d'une fonction d'instruction de commande linux (appelée ici *cmdsh*)
- d'une fonction de multithreading (ici *mt*)

```python
#SLAVE
import time
import sys
import subprocess
import threading
from os import listdir
from os import path
import pandas as pd
from pandas.util import hash_pandas_object


pth = '/tmp/sarda-20/'
hostname = subprocess.check_output("hostname", shell=True, universal_newlines=True).strip()

# fonction de commande de shell
def cmdsh(cmd):
    try:
        output = subprocess.check_output(cmd, shell=True, universal_newlines=True, stderr= subprocess.STDOUT)
    except subprocess.CalledProcessError as exc:
        print(hostname + " Status : FAIL", exc.returncode, exc.output)
        return False
    else:
        return True


def mt(list_m, fct, *args ):
    threads = []
    for i in list_m:
        x = threading.Thread(target = fct, args = (i, *args))
        threads.append(x)
        x.start()

    for x in threads:
        x.join()
```

II. Étape 11

Le calcul du *hash code* de chaque mot n'est pas aussi trivial que qu'imaginé, en raison de la non-unicité du calcul de hash avec la fonction stardard *hash* de python (propre à l'ouverture de session). J'ai alors utilisé le module *black2b* de la librairie *hashlib* pour obtenir un code en hexadécimal (passage en décimal aisé via la fonction *int(,16)*).

La partie « split » du programme Master est conçue de telle sorte que le découpage de l'input se fait en fonction du nombre de Slaves disponibles (i), et du nombre voulu de split à traiter par Slave (j). Le nombre de lignes par split (m) est alors calculé par division euclidienne suivante du nombre de lignes (n) :

$$m = n // (i \times j) + 1$$

III. Étape 12

Les deux phases les plus lentes sont le *Map* et le *Shuffle*, alors que la plus courte est la phase de *Reduce*. Ces trois phases devraient s'allonger en fonction des split traités, alors que la première phase de vérification de l'environnement devrait durer un temps constant.

Le fichier contenant trois lignes de trois mots chacun est traité en 6s environ, avec un seul split par machine. Le résultat n'est à ce stade pas satisfaisant car un seul ordinateur traite le WordCount de ce fichier en 10ms environ. Il est probable que le fichier n'est pas assez lourd pour favoriser un cluster, lequel est pénalisé automatiquement par les temps de connexions entre machines et les envois de fichiers (shuffles) par le protocole *SCP*.

```
[tp-1d22-01% python3 MASTER.py HOSTS.txt                    /tmp/sarda-20 ]
VERIF CONNECTION ET ENVOI FIC MACHINES -- 1.11s
MAP FINISHED -- 1.93s
SHUFFLE FINISHED -- 1.94s
REDUCE FINISHED -- 0.34s
```

IV. Étape 13 – optimisation des scripts

1ère option : envoi groupé des *hashfiles*

Ayant subi des problèmes de *SSH-SCP* à cause d'un nombre trop important de connections pour le traitement du fichier sante_publique.txt, j'ai tenté le regroupement et l'envoi en une seule fois de tous les *hashfiles* vers un destinataire. Ce processus limite le nombre d'envois par package mais alourdi le code. Une autre option consiste à créer des dossiers par destinataire ou à créer des **.tar* qu'on envoie par la suite.

Les résultats globaux pour quatre Slaves sont encore une fois peu satisfaisants : le fichier *sante_publique.txt* est traité en 39s avec 4 Slaves contre 2.8s sur un unique ordinateur. L'optimisation du temps de calcul n'est pas linéaire, on descend à environ 30s pour 12 Slaves (étude réalisée supra).

2nde option : compilation des mots dans un seul fichier avant envoi entre les machines

Ces résultats me poussent à améliorer la lecture des splits et la création des fichiers de shuffle comme suit :

- le(s) fichier(s) de split est/sont lu(s) par le Slave et les mots sont concaténés dans une chaine de caractères avant que cette dernière ne soit intégralement écrite dans un fichier de map (*UMX.txt*),

- le shuffle est réalisé par le calcul du modulo du hash par le nombre de machines, et les mots à transmettre à chaque machine sont concaténés dans une chaine de caractère avant d'être intégralement écrite dans un fichier txt (lequel doit être envoyé à la machine).

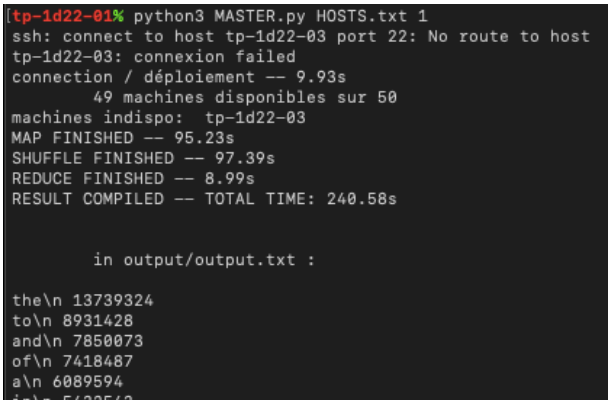
Cette amélioration permet de diminuer le temps de calcul : on passe à 5.38s pour le fichier sante_publique.txt avec 12 Slaves.

Optimisation du calcul de codes de hachage par calcul vectoriel

Le passage des mots extraits des fichiers de map (*UMX.txt*) en tableau *DataFrame* de la librairie *pandas*, permet de calculer de manière vectorisée les codes de hash des mots. La librairie *blake2b* est abandonnée pour être remplacée par une méthode de hash de *DataFrame* (qui semble délivrer un code constant).

Ce procédé permet d'obtenir un temps de calcul du WordCount pour le fichier WET de 300 Mo d'environ 70s avec [12 ; 1] comme paramètre [nb_Slaves ; nb_splits/Slave].

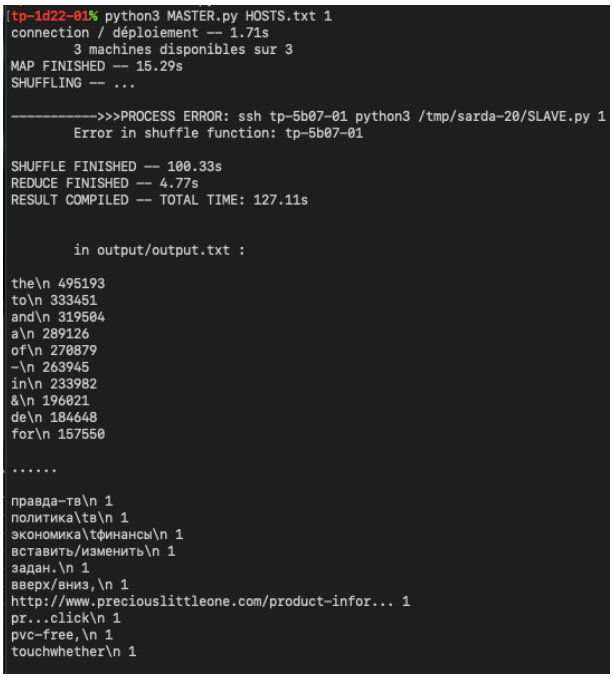
Grace à ce nouveau procédé de calcul, le cluster vient à bout du fichier texte de 380 Mo, ce qu'il ne parvenait pas à réaliser auparavant ; en effet, l'opération de hash (dépasse le million de calcul et de création de fichier de shuffle) était effectuée de manière séquentielle et les machines « s'essoufflaient » en dépassant le million de calcul (possible saturation de la RAM).



Enfin en testant le programme sur de plus gros fichiers, on se rend compte que le délai total de traitement n'est pas proportionnel à la taille du fichier.

En effet, un fichier de 4,5 Go (11,8 fois plus volumineux que le fichier initial) est traité en 240s (image ci-contre), soit 7 fois plus lent seulement. Ainsi, le calcul distribué répond bien à de l'optimisation du temps de calcul sur les fichiers volumineux.

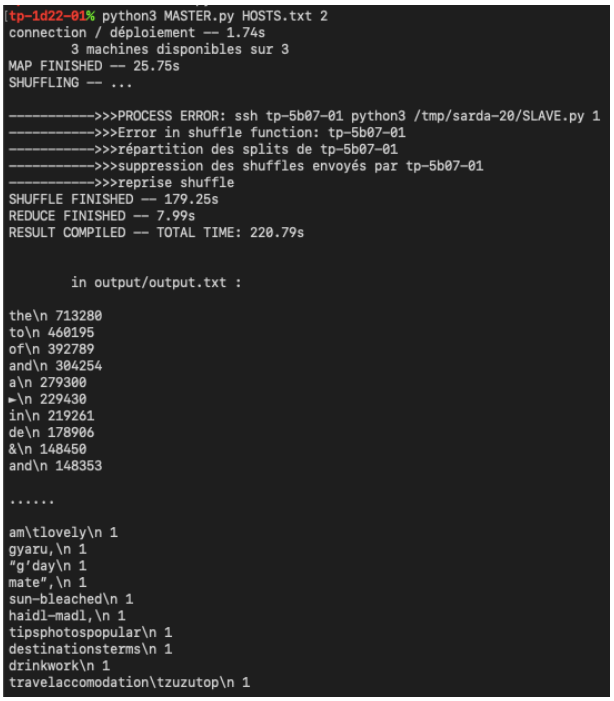
V. Résolution des pannes



```
[tp-1d22-01% python3 MASTER.py HOSTS.txt 1
connection / déploiement -- 1.71s
        3 machines disponibles sur 3
MAP FINISHED -- 15.29s
SHUFFLING -- ...

------------>>>PROCESS ERROR: ssh tp-5b07-01 python3 /tmp/sarda-20/SLAVE.py 1
        Error in shuffle function: tp-5b07-01

SHUFFLE FINISHED -- 100.33s
REDUCE FINISHED -- 4.77s
RESULT COMPILED -- TOTAL TIME: 127.11s


        in output/output.txt :

the\n 495193
to\n 333451
and\n 319504
a\n 289126
of\n 270879
-\n 263945
in\n 233982
&\n 196021
de\n 184648
for\n 157550

......

правда-тв\n 1
политика\tв\n 1
экономика\tфинансы\n 1
вставить/изменить\n 1
задан.\n 1
вверх/вниз,\n 1
http://www.preciouslittleone.com/product-infor... 1
pr...click\n 1
pvc-free,\n 1
touchwhether\n 1
```

Je propose de coder le programme Master de telle sorte que deux jeux différents de splits sont envoyés à chaque machine : l'un à traiter et l'autre en cas de dépannage (réplication). Surveillant les retour erreur des Slaves, le Master peut alors demander, lors d'un crash d'un Slave, de réaliser en urgence la tâche conférée initialement à ce dernier.

En tuant le processus de shuffle sur un des workers, on obtient bien sur la machine Master un retour d'erreur lors de l'opération Mapreduce avec le nom du worker crashé et la fonction durant laquelle il a crashé. Le temps de travail est notablement impacté et le résultat final est logiquement tronqué (sur l'exemple ci-contre, 495 193 « the » comptabilisés contre 730 280 normalement).



```
[tp-1d22-01% python3 MASTER.py HOSTS.txt 2
connection / déploiement -- 1.74s
        3 machines disponibles sur 3
MAP FINISHED -- 25.75s
SHUFFLING -- ...

------------>>>PROCESS ERROR: ssh tp-5b07-01 python3 /tmp/sarda-20/SLAVE.py 1
------------>>>Error in shuffle function: tp-5b07-01
------------>>>répartition des splits de tp-5b07-01
------------>>>suppression des shuffles envoyés par tp-5b07-01
------------>>>reprise shuffle
SHUFFLE FINISHED -- 179.25s
REDUCE FINISHED -- 7.99s
RESULT COMPILED -- TOTAL TIME: 220.79s


        in output/output.txt :

the\n 713280
to\n 460195
of\n 392789
and\n 304254
a\n 279300
►\n 229430
in\n 219261
de\n 178906
&\n 148450
and\n 148353

......

am\tlovely\n 1
gyaru,\n 1
"g'day\n 1
mate",\n 1
sun-bleached\n 1
haidl-madl,\n 1
tipsphotospopular\n 1
destinationsterms\n 1
drinkwork\n 1
travelaccomodation\tzuzutop\n 1
```

La reprise du travail des splits du ou des workers crashés fonctionne, comme le montre la capture d'écran ci-contre. Les splits initialement affectés aux machines crashées sont redispatchées chez certaines machines en état de marche. D'autre part, il faut réaffecter les hashfiles à une seule machine en état de marche.

VI. Étude du choix optimal des paramètres [nb_Slave ; nb_Splits]

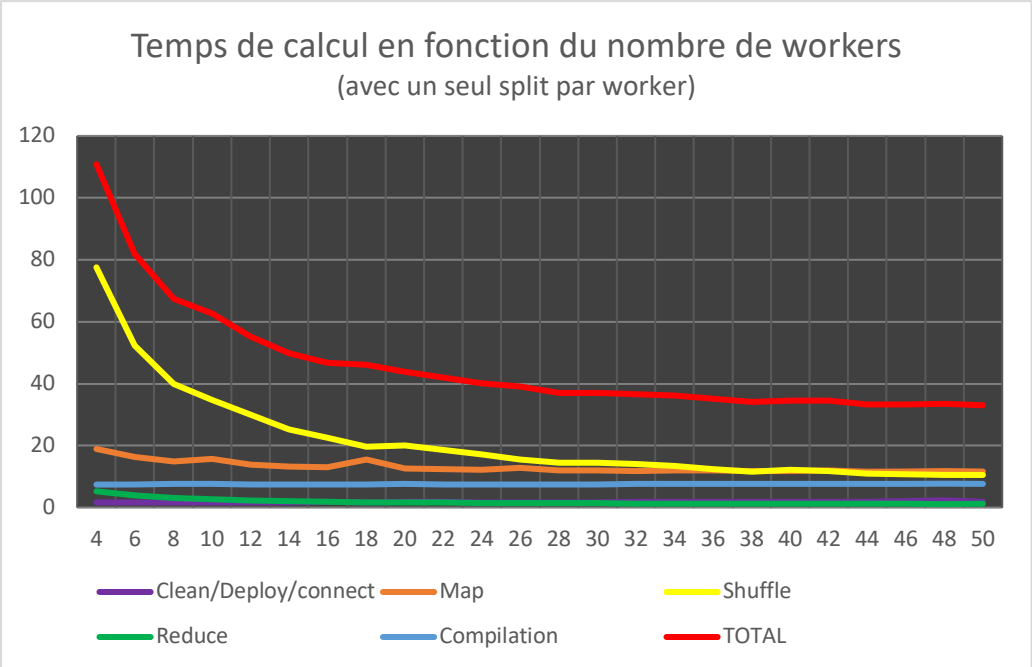
```
tp-1d22-01% python3 MASTER.py HOSTS.txt 1
connection / déploiement -- 1.71s
        30 machines disponibles sur 30
MAP FINISHED -- 11.94s
SHUFFLE FINISHED -- 14.49s
REDUCE FINISHED -- 1.40s
RESULT COMPILED -- TOTAL TIME: 37.09s


        in output/output.txt :

the\n 713280
to\n 460195
and\n 452607
a\n 419368
of\n 392789
-\n 379508
in\n 323772
de\n 289964
►\n 229430
&\n 228254

......

diet."so\n 1
<urn:uuid:65480a9c-a5d5-4558-bb1a-4f75a24bac4b>\n 1
sha1:brvb4u3yrdyaax2agozv4ap5a64ve4e4\n 1
$.abovethetop\n 1
$("div.top_paysites_menu").hover(function()\n 1
#11155cartoon\n 1
gbp$\n 1
typesalt\n 1
<urn:uuid:f3846c27-c591-458a-a6f9-d8c0df6e2b39>\n 1
katana?\n 1
```

La recherche de l'optimisation passe aussi par la modulation du nombre de splits et du nombre de Slaves. En prenant en input un fichier texte de 380 Mo, on teste l'optimisation du nombre de Slaves en conservant un seul split.

- Les temps présentés ci-dessous comprennent les phases initales de CLEAN et de DEPLOY,

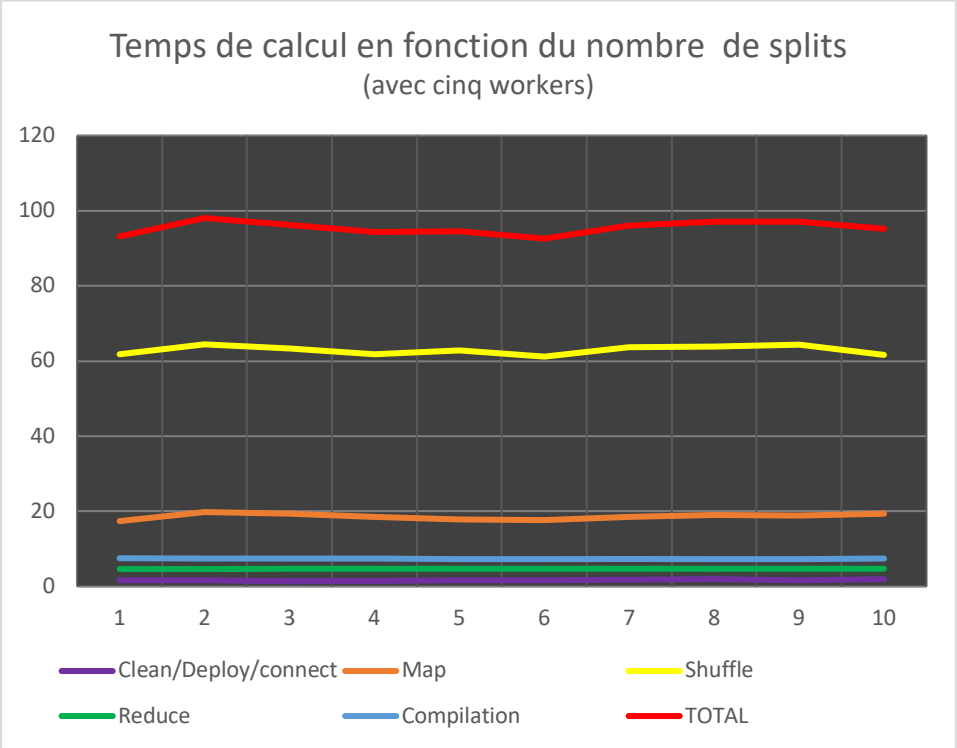
- la sortie est contrôlée automatiquement pour s'assurer que le résultat donné par MapReduce soit juste (713 280 « the » comptabilisés),

- l'opération est lancée deux fois pour ne pas avoir de latence due à une première connections sh.



On peut constater qu'à partir de 26 workers on descend sous la barre des 40 secondes. Le minimum n'est pas atteint avec 50 machines, mais le gain de temps devient minime à partir de 36 machines.
Le shuffle est la phase à qui bénéficie le plus la multiplication des workers.

On fait désormais varier le nombre de splits pour un même nombre de workers. On va mener l'étude avec 5 workers pour constater les améliorations notables du temps de calcul.




```
tp-1d22-01% python3 MASTER.py HOSTS.txt 1
connection / déploiement -- 1.96s
        50 machines disponibles sur 50
MAP FINISHED -- 8.87s
SHUFFLE FINISHED -- 7.61s
REDUCE FINISHED -- 1.32s
RESULT COMPILED -- TOTAL TIME: 26.74s


        in output/output.txt :

the\n 713280
and\n 452607
to\n 433764
of\n 392789
a\n 390887
-\n 361423
in\n 304526
de\n 277046
&\n 228254
►\n 212642

......

pauvrescop21:\n 1
développementcoordination\n 1
ferdi.cette\n 1
bar/prep\n 1
umgebungharzharzvorlandhessenhessisches\n 1
jacuzzi(whrilpool)\n 1
gesundheitsmanagement2.3\n 1
durchhaltevermögen\n 1
ferra.ru,\n 1
productsnewsvideosabout\n 1
```

A partir de 11 splits pour 5 workers, le programme crashe pour cause de nombre de fichiers ouverts trop élevé. Pour 50 workers, il n'est pas possible d'excéder 9 splits. On gagne environ 1 secondes avec 50 xorkers et 9 splits par rapport à 50 machines et 1 split.

Après avoir constaté que les résultats n'étaient pas satisfaisants, j'ai parallélisé le programme SLAVE au maximum pour gagner des délais sur la phase de Shuffle et de Reduce. Toutefois, l'utilisation du multithreading me fait perdre du temps avec 5 workers (entre 5 et 8 secondes pour 4 splits par workers avec 5 workers, et environ 2 secondes avec 4 splits et par workers et 50 workers).

On va alors tenter de gagner du temps non pas sur la phase de shuflle par worker, mais sur la phase de split de l'input. La commande linux *split* permet de gagner environ 5 secondes sur le découpage de l'input. On descend alors à 27 secondes pour 1 splits et 50 workers (capture d'écran ci-contre). Le fait d'augmenter le nombre de splits par worker fait perdre du temps (environ 1 seconde en passant à 2 splits).