TRAVAUX PRATIQUES

Implémentation de Hadoop MapReduce "from scratch" en Python

Introduction

Dans ce rapport, nous allons dérouler une méthode simple pour développer « from scratch » un algorithme du style « Map – Shuffle- Reduce » en Python.
Nous expliquerons chaque étape de la construction de cet algorithme, les problèmes rencontrés, les solutions répondant à ces problèmes, ainsi que les remarques et réflexions qui se sont posées lors de cette implémentation.
Pour chaque étape cruciale du code, nous les chronométrons afin de comparer le calcul en système distribué au calcul en série.

I - Programme WordCount en séquentiel non parallélisé

Le programme wordcount.py implémenté est donné en annexe.
Nous avons utilisé un dictionnaire pour écrire cet algorithme car c'est la structure la plus pratique pour le comptage des occurrences et le triage de chaque mot d'un texte fourni en input ; le triage par nombre d'occurrence et par ordre alphabétique (si le nombre d'occurrence est identique pour plusieurs mots différents) est d'autant plus simple grâce à cette structure. Chaque étape sera chronométrée grâce à la librairie Python « time » et grâce à des pointeurs de temps du type « time.time() » qui seront placés à chaque début et fin des étapes.

Nous avons représenté dans le tableau ci-dessous, le temps de comptage et de triage pour chaque fichier texte testé. Les 50 premières occurrences sont listées dans l'annexe TP-Etape1.txt.

Fichier	Taille du fichier	Temps de calcul
input.txt	43 octets	Comptage de mots : 0.0999ms Triage de mots : 0.0041ms
forestier_mayotte.txt	1ko	Comptage de mots : 0.1709ms Triage de mots : 0.0432ms
deontologie_police_nationale.txt	7ko	Comptage de mots : 0.5202ms Triage de mots : 0.2897ms
domaine_public_fluvial.txt	71ko	Comptage de mots : 2.8510ms Triage de mots : 1.2062ms

sante_publique.txt	18,1 Mo	Comptage de mots : 681.3421ms Triage de mots : 57.4970ms
CC-MAIN-20170322212949-00140-ip-10-233-31-227.ec2.internal.warc.wet	398,8 Mo	Comptage de mots : 18172.6263ms Triage de mots : 13095.0668ms

II - Travailler en réseau sur plusieurs machines

Cette partie du rapport vise à passer rapidement sur les étapes 2 à 6 du projet, qui consistent à se connecter sur plusieurs machines de l'école en ssh, travailler sur des fichiers locaux, créer des répertoires sur ces machines et connecter plusieurs machines entre elles.

Pour passer rapidement sur cette partie nous allons simplement lister les commandes utilisées qui nous serviront dans les prochaines étapes.

Les commandes shell que nous utiliserons pour récupérer les adresse IP et les noms des machines sont :

- hostname (pour le nom de la machine)
- hostname –long (pour le nom long de la machine)
- nslookup ipconfig ou hostname -I (pour l'adresse IP)

Pour tester la communication entre plusieurs machines, nous avons utilisé les commandes ci-dessous:

- *ping* tp-1a201-40 (commande lancée depuis la machine tp-1a201-40). La commande *ping* fonctionne aussi avec le nom long et l'adresse IP de la machine. Les statistiques du *ping* sont :

  --- tp-1a201-40.enst.fr ping statistics ---
  4 packets transmitted, 4 received, 0% packet loss, time 79ms
  rtt min/avg/max/mdev = 0.032/0.035/0.043/0.008 ms

- la commande *ping* peut s'écrire aussi depuis ma machine. Les statistiques du *ping* sont :

  --- tp-1a201-40.enst.fr ping statistics ---
  5 packets transmitted, 5 packets received, 0.0% packet loss
  round-trip min/avg/max/stddev = 3.419/7.727/9.273/2.172 ms

- pour se connecter en ssh, il suffit de saisir depuis notre machine la commande ssh amichalewicz@tp-1a201-40 (= ssh nomUtilisateur@nomMachine)
- pour ne pas avoir à saisir le mot de passe à chaque connexion il suffit de générer des clés publiques / privées grâce à la commande *ssh*-keygen et ensuite de copier la clé dans le ssh des machines de l'école grâce à la commande :
  cat ~/.ssh/id_rsa.pub | ssh login@monserver.com "cat - >> ~/.ssh/authorized_keys"

Dans l'étape 3, nous avons seulement créé des dossiers et écrit des fichiers dans les répertoires temporaires des machines grâce aux commandes :

- mkdir /tmp/amichalewicz
- nano ~/fperso.txt
- cat ~/fperso.txt

Un fichier placé sur le répertoire personnel de la machine se retrouvera sur toutes les machines lorsque j'utiliserai mon compte mais un fichier sur le répertoire temporaire ne sera que sur le répertoire temporaire de la machine sur laquelle on l'a créé.
Si l'on souhaite depuis une machine A, copier de la machine B (amichalewicz@tp-1a201-40) vers la machine C (amichalewicz@tp-1a260-00) avec les disques locaux, il faut utiliser la commande :
scp amichalewicz@tp-1a201-40:/tmp/amichalewicz/local2.txt amichalewicz@tp-1a260-00:/tmp/amichalewicz

III- Exécution de programmes python depuis une machine sur une autre machine et gestion des erreurs (étape 4, 5, 6)

Pour lancer et exécuter des programmes python en ssh sur une machine, nous utilisons la commande :

- *python /tmp/amichalewicz/SLAVE.py* (une fois le répertoire /tmp/amichalewicz créé sur la machine de l'école)

De plus *ssh amichalewicz@tp-1a201-39 python /tmp/amichalewicz/SLAVE.py* est la commande pour exécuter le programme python en ssh depuis une autre machine.

De plus le programme SLAVE.py peut être exécuté depuis un autre programme python MASTER.py si celui-ci appelle les commandes ci-dessus sur la machine voulue grâce à la libraire subprocess et la commande « run ». Par exemple, afin de lancer le programme SLAVE.py via le programme Master.py, il suffit d'exécuter le code ci-dessous :

```
import subprocess as sp
commandShell = 'ssh amichalewicz@ tp-1a201-40 python3 /tmp/amichalewicz/SLAVE.py
monProcess = sp.run(commandShell , shell=True, capture_output=True, timeout=10).
```

L'attribut timeout=10 dans la commande sp.run de python, nous permet de gérer le timeout. Si l'exécution de Slave.py met plus de 10 secondes à s'exécuter, une erreur de type TimeoutExpired est générée. Il va falloir gérer ce type d'erreur.

Afin de gérer les erreurs lors de l'exécution, nous utilisons donc la commande *try … except* de python. Cette commande permet de gérer tout type d'erreur. On peut grâce à celle-ci gérer les erreurs de type timeout.

La commande ci-dessous nous permet d'exécuter Slave.py et de passer à la commande suivante si celui-ci obtient un timeout. Nous remarquons que grâce à la condition *if,* une erreur autre que timout serait affichée pour pouvoir la résoudre. Si l'on veut gérer toutes les erreurs et ne pas les afficher, il suffit de ne rien mettre en attributs du *except* à la place de *sp..TimeoutExpired*, cela gérera toutes les erreurs.

```
try:
    commandShell = 'ssh @ tp-1a201-40 python3 /tmp/amichalewicz/SLAVE.py'
    monProcess = sp.run(commandShell , shell=True, capture_output=True, timeout=10)
except sp.TimeoutExpired:
  print("echec d'execution de SLAVE.py dû au timeout", end='\n')
  pass
else:
  if monProcess.returncode:
    print("Problème d'exécution de SLAVE.py)
    print(monProcess.stderr)
  else:
    print("Fin d'execution de SLAVE.py')
  pass
```

IV – Nettoyage et Déploiement du programme Slave.py sur un ensemble de machines (étape 7 et 8)

a) Déploiement du programme Slave.py sur un ensemble de machines

Pour cette étape, le programme DEPLOY.py sera écrit grâce à 2 fonctions ; la deuxième sera appliquée sur une liste de machine allumées. Nous exécuterons dans la l'ordre suivant :

- la fonction « **rechercheMachinesOn** » qui recherchera et testera la connexion ssh sur chaque machine de l'école grâce à la commande shell
  'ssh amichalewicz@' + Id machine + ' hostname'

  Pour vérifier la connexion sur chaque machine, nous utiliserons la fonction python *Try …except* et la commande shell ci-dessus avec un timeout de 10 secondes (c'est-à-dire que nous laissons 10 secondes à notre machine pour se connecter à chaque machine de l'école avant de renvoyer un TimeOutexception). En fonction de ce Timeout, nous afficherons « Id machine éteinte » si une erreur Timeout est renvoyé et « Id machine allumé » sinon. La fonction *Try… except* prend aussi en compte si un autre type d'erreur est renvoyé lors de la connexion sur chaque machine.

- Une création d'une liste de machines allumées est faite grâce à la fonction rechercheMachineOn, en vérifiant si celle-ci nous renvoie le nom de la machine ou non.

- la fonction « **copieSurMachineDeSlave** » qui copiera le programme SLAVE.py sur toutes les machines allumées de l'école repérée par la fonction « **rechercheMachinesOn** ».

  Cette fonction créera le dossier /tmp/amichalewicz/ grâce à la commande shell :
  'mkdir -p /tmp/amichalewicz/ ' si ce répertoire n'existe pas sur la machine.
  Le fait de lancer d'abord cette commande shell en ssh sur la machine puis de lancer une nouvelle commande shell qui copie le fichier SLAVE.py (commande shell *'scp ' +*

fichier +' amichalewicz@' + machine + ':/tmp/amichalewicz'), nous assure que le mkdir se termine avant de copier le fichier dans le répertoire adéquate.

N.B : un timeout de 10 secondes est aussi utilisé dans cette fonction pour créer le répertoire et copier le fichier, cela permet de gérer l'erreur si l'une de ces 2 méthodes ne fonctionne pas.

Il y a 2 méthodes pour lancer ces étapes : de manière séquentielle avec un boucle sur toutes les machines allumées ou alors de manière parallèle grâce à la fonction « **map** » de la librairie python *multiprocessing*. **Nous utiliserons la fonction map pour lancer ces 2 étapes de manière parallèle.**

b) Nettoyage d'un répertoire sur un ensemble de machines

Pour cette étape, le programme CLEAN.py sera écrit grâce à 2 fonctions :

- la fonction « **rechercheMachinesOn** » qui est exactement la même fonction que pour le programme DEPLOY.py

- la fonction « **nettoyageMachine** » qui va nettoyer en ssh le répertoire /tmp/amichalewicz/ grâce à la commande shell *'ssh amichalewicz@' + idmachine + ' rm -rf /tmp/amichalewicz/'*.
  Bien sûr nous utiliserons un try … except pour gérer les erreurs, si erreurs il y a lors du nettoyage du répertoire.

De la même manière que pour le DEPLOY, nous utiliserons la fonction « **map** » de la librairie python pour lancer le nettoyage de manière parallèle.

IV – MapReduce - SPLIT et MAP (étape 10)

a) Copie des splits sur les machines

Dans le MASTER, nous écrivons une fonction **copySplits** qui de la même manière que pour les étapes précédentes, créera un répertoire /tmp/amichalewicz/splits/ en ssh avec un mkdir -p /tmp/amichalewicz/splits' , puis copiera avec la commande shell *scp*. Nous lançons d'abord la commande shell *mkdir* sur la machine puis la commande *scp* qui copie les splits. Cela nous assure que le *mkdir* se termine avant de copier les splits dans le répertoire adéquate.

La difficulté se trouve dans l'association d'un fichier splits par machine. Pour cela nous avons créé un vecteur de tuples (split n°i , machine n°i). Pour trouver le nombre de fichier Splits à copier nous comptons le nombre de fichier splits dans le répertoire de notre machine, et pour chacun de ses fichiers splits nous lui associons une machine unique.

La difficulté a été lorsque, nous « splitons » un fichier de taille assez importante (c'est-à-dire de taille supérieure au Mo). En effet si le nombre de split est plus grand que le nombre de machines disponibles, il faut du coup envoyer plusieurs splits sur une machine, ou alors créer un nombre de split égale au nombre de machines disponibles. Ce problème n'existe que sur

les fichiers de taille importante et non avec le fichier input.txt de départ. Nous parlerons de la méthode choisie dans l'étape 13 en essayant d'améliorer le programme.

Nous utiliserons la fonction « **map** » de la librairie python pour lancer la **copie des splits sur chaque machine de manière parallèle**.

b) MAP par le programme SLAVE.py

L'appel du programme SLAVE.py avec l'argument 0, lancera la phase de map du SLAVE.
Le Map se fera grâce à la fonction « write_map » dans le programme SLAVE.py. Cette fonction prend en attribut un fichier split, son chemin et le numéro i du split (ex : pour S0.txt, i=0). Cette fonction va créer le dossier /maps/ dans le répertoire correspondant à l'attribut « chemin ». Ensuite elle créera le fichier « UMi.txt » dans ce dossier, et écrira sur chaque ligne de ce fichier un tuple résultant de la fonction « count_word » qui associe chaque mot du fichier split à l'occurrence 1.
Nous gérons les erreurs d'écriture grâce à un try - except, et un message est affiché lorsque que l'écriture du fichier « UMi.txt » se déroule sans erreurs.

c) Lancement du SLAVE.py option 0 par le MASTER.py

A partir de chaque tuple (split n°i , machine n°i) créés précédemment, nous créons un nouveau tuple en lui ajoutant l'attribut 0 correspondant à l'option 0 du SLAVE. Une fonction « runSlaveOn » prenant en attribut le fichier, la machine et l'option du SLAVE lance le programme SLAVE.py sur cette machine grâce à la commande Shell suivante :
'ssh amichalewicz@' + machine + ' python3 /tmp/amichalewicz/SLAVE.py 0 /tmp/amichalewicz/splits/' + fichier

Attention, notre programme SLAVE.py ayant été écrit en python 3, il ne faut pas oublier de lancer la commande shell « *python3 /tmp/amichalewicz/SLAVE.py* » et non « *python /tmp/amichalewicz/SLAVE.py* », car les machines de l'école possèdent plusieurs versions de python et donc la commande shell *python « /tmp/amichalewicz/SLAVE.py »*, lancerai le SLAVE en python 2 et une erreur serait générée.
Dans l'éventualité où l'exécution du SLAVE sur la machine génère une erreur, nous utilisons encore un try – except.
Afin de lancer de manière parallèle, la fonction « runSlaveOn » sur chaque machine nous utiliserons la fonction « **starmap** » de la librairie python multiprocessing.pool ; cette fonction fait exactement la même chose que la fonction « map » utilisée précédemment, sauf que le deuxième argument de cette fonction peut être un tuple. Cela permet de lancer la fonction « runSlaveOn »avec ses 3 attributs.
Une fois le lancement en parallèle des « runSlaveOn » terminé, nous affichons "*MAP FINISHED" (print('MAP FINISHED', end='\n\n')*) », cela permet de savoir que tous les Slaves sont terminés sur chaque machine.

V – MapReduce - SHUFFLE (étape 11)

a) Envoi du fichier machine.txt sur les machines

A l'aide de la liste de machines allumées créées précédemment lors de l'étape 7 et 8, nous écrivons un fichier « machine.txt » avec un hostname de machine sur chaque ligne de ce fichier.

Ensuite une fonction copyFichierNbMachine(machineTP) va copier le fichier « machine.txt » sur la machine « machineTP » au répertoire /tmp/amichalewicz/. Bien sûr nous lançons cette fonction en parallèle grâce à la fonction « map » de la librairie python multiprocessing.pool .

b) Préparation et Exécution du SHUFFLE dans le SLAVE.py

L'appel du programme SLAVE.py avec l'argument 1, lancera la phase de SHUFFLE en entière.
La phase de préparation et d'exécution du shuffle se fait grâce à la fonction «**prepareAndSendShuffle** » du SLAVE.py. Cette fonction comprend 2 attributs :

- Un fichier UM (=Unsorted Map)
- Un répertoire dans lequel on créera le dossier /shuffles/

A noter que les étapes suivantes se feront avec un try – except pour gérer les erreurs qui pourrait ressortir de celles-ci.
Sur la machine sur laquelle nous exécuterons la phase de shuffle, nous créerons le répertoire /shuffles/.
Les deux principales étapes du «**prepareAndSendShuffle** » sont :

- La première sera d'ouvrir chaque fichier « UMi.txt », et pour chaque ligne de ce fichier, nous prendrons le premier élément de chaque ligne qui correspond au mot et nous appliquerons ensuite la fonction de hashage *zlib.adler32(bytearray())* de la librairie zlib. Nous n'utiliserons pas la fonction *hash* de python car celle-ci renvoie aussi des nombres négatifs pour certains mots. Grâce à cette clé de hashage, nous regardons si le fichier .txt lui correspondant existe, si oui, nous ajoutons le mot et son occurrence dans ce fichier, sinon nous le créons au nom de « cléDeHashage-nomMachine.txt » (=<hash><hostname>) et rajoutons aussi le mot et son occurrence dans ce nouveau fichier. Nous ferons la même chose pour tous les mots du fichier « UMi.txt ».
  A noter que l'existence du fichier « cléDeHashage-nomMachine.txt » sera tester avec la fonction *os.path.isfile* de la librairie os de python.

- La seconde étape sera d'envoyer tous les fichiers « cléDeHashage-nomMachine.txt » ayant la même clé de hashage sur une même machine. Pour savoir sur quelle machine envoyer les mêmes clés de hashage, nous appliquons un modulo sur la clé de hashage avec le nombre de machine c'est-à-dire numeroMachine = hash % nbMachines.
  De la même manière que pour la création des autres répertoires, un répertoire shufflesreceived sera créé la machine numeroMachine s'il n'existe pas.

c) Lancement du SHUFFLE via le SLAVE.py par le MASTER.py

Cette fois ci, partir de chaque tuple (split n°i , machine n°i) créés en étape 7 et 8, nous créons un nouveau tuple en lui ajoutant l'attribut 1 correspondant à l'option 1 du SLAVE et en remplaçant le split n°i par l'UMi.txt ; le tuple sera de la forme (UMi.txt, machinen°i, 1). De la même manière que précédemment, la fonction « **runSlaveOn** » lancera le programme SLAVE.py avec l'option 1 sur la machine grâce à la commande Shell suivante :
'ssh amichalewicz@' + machine + ' python3 /tmp/amichalewicz/SLAVE.py 1 /tmp/amichalewicz/maps/' + fichierUM

Dans l'éventualité où l'exécution du SLAVE sur la machine génère une erreur, nous utilisons encore un *try – except*.

Afin de lancer de manière parallèle la fonction « runSlaveOn » sur chaque machine nous utiliserons aussi la fonction « **starmap** » de la librairie python multiprocessing.pool. Cela lancera la fonction « runSlaveOn » avec l'option du shuffle avec ses 3 attributs.
Une fois le lancement en parallèle des « runSlaveOn » terminé, nous affichons « *SHUFFLE FINISHED » (code :print(SHUFFLE FINISHED, end='\n\n'))*, cela permet de savoir que tous les Slaves sont terminés sur chaque machine.

VI – MapReduce - REDUCE (étape 12)

a) Phase de REDUCE exécutée par le SLAVE.py

L'appel du programme SLAVE.py avec l'argument 2, lancera la phase de REDUCE en entière.
La phase de REDUCE se fera en plusieurs étapes. La première sera d'abord de créer le répertoire /reduces/ si celui-ci n'existe pas déjà sur la machine sur laquelle nous sommes.

Ensuite l'étape suivante sera d'ouvrir chaque fichier « cléDeHashage-nomMachine.txt » se trouvant dans le répertoire /tmp/amichalewicz/shufflesreceived/ de la machine et de regrouper tous les mêmes mots dans un seul fichier « cléDeHashage.txt ».
Par la fonction os.path.isfile de la librairie os de python, nous vérifions si oui ou non le fichier « cléDeHashage.txt » existe, le cas échéant nous le créons.

Un fois chaque fichier « cléDeHashage.txt » créé, nous lui appliquons la fonction words_Reduced qui va simplement compter le nombre de ligne du fichier (qui est égale au nombre d'occurrence du mot) et va réécrire le fichier « cléDeHashage.txt » avec seulement comme texte à l'intérieur : « mot nboccurrence »

A noter que les étapes ci-dessous se feront avec un try – except pour gérer les erreurs qui pourraient ressortir de celles-ci. En fonction du résultat, un message sera affiché pour dire si oui ou non si le fichier reduce a bien été créé.

b) Lancement du REDUCE via le SLAVE.py par le MASTER.py

Cette fois ci, nous créons un nouveau tuple avec l'option 2 du SLAVE ; le tuple sera de la forme (0, machine[i], 2). De la même manière que précédemment, la fonction « runSlaveOn » lancera le programme SLAVE.py avec l'option 2 sur la machine grâce à la commande Shell suivante :
'ssh amichalewicz@' + machine[i] + ' python3 /tmp/amichalewicz/SLAVE.py 2 /tmp/amichalewicz/'

Dans l'éventualité où l'exécution du SLAVE sur la machine génère une erreur, nous utilisons encore un try – except.
Afin de lancer de manière parallèle, la fonction « runSlaveOn » sur chaque machine nous utiliserons aussi la fonction « **starmap** » de la librairie python multiprocessing.pool. Cela lancera la fonction « runSlaveOn » avec l'option du reduce avec ses attributs.

Une fois le lancement en parallèle des « runSlaveOn » terminé, nous affichons « REDUCE FINISHED » (code : *print(REDUCE FINISHED', end='\n\n')*), cela permet de savoir que tous les Slaves sont terminés sur chaque machine.

c) Chronométrage de chaque phase du map reduce

Au début de chaque phase du MAP REDUCE des pointeurs de temps du type time.time() seront placé à chaque début et fin de phase pour les chronométrer (time étant une librairie python). Il suffira de faire la différence entre un pointeur de début et un pointeur de fin de phase pour savoir le temps qu'elle aura duré.

Analysons le temps réalisé de chaque phase pour le fichier input.txt avec seulement les mots (Beer, Deer…). Le résultat est le suivant :

Temps de connexion et recherche des machines allumées : 34606.36019706726 ms
Temps de copie des splits sur les machines allumées : 1094.895839691162 ms
Temps de copie des fichiers machine.txt : 818.1920051574707 ms
Temps du MAP : 797.5130081176758 ms
Temps du SHUFFLE : 2080.7721614837646 ms
Temps du REDUCE : 776.6368389129639 ms
Temps du rapatriement des fichiers reduces et de l'affichage résultat :773.49400520324 ms

- La phase la plus longue est la phase de SHUFFLE : en effet le temps d'envoi des fichiers « cléDeHashage-nomMachine.txt » vers les autres machines est long ainsi que le temps de connexion de ces machines entre elles car elles se connectent toutes en ssh.
- La phase de MAP est en revanche la plus rapide, comme celle de REDUCE, même si cette dernière est légèrement plus longue que la phase de MAP. Ces phases s'exécutent en effet sur chaque machine en parallèle mais ne demande de communication entre chaque machine. Par conséquent, cela ne leur fait pas perdre de vitesse d'exécution.

Pour un fichier de cette taille-là, un MAP-REDUCE sur un WordCount n'est pas très efficace car en comparant avec nos résultats de départ (ci-dessous) on voit qu'un wordcount en séquentiel est plus rapide :

Comptage de mots : 0.0999ms
Triage de mots : 0.0041ms

Il faudrait comparer avec des fichiers de tailles plus importantes (cela sera fait dans la partie suivante correspondant à l'étape 13).

VII – MapReduce – création des Splits et proposition d'optimisation (étape 13)

a) Création des Splits et Affichage du résultat

1- Création des Splits

Les splits seront créés à partir du fichier input.txt grâce à la fonction « split ». Cette fonction prend en entré un chemin et va créer un répertoire /splits à ce chemin s'il n'existe pas et ensuite va ouvrir le fichier input.txt à ce même chemin et écrire un fichier split par ligne de ce fichier, c'est-à-dire que le contenu d'un fichier split correspondra à une ligne du fichier input.txt. Nous utiliserons dans cette fonction un try – except pour gérer les erreurs qui pourrait ressortir lors de la création des splits.

2- Affichage du résultat

Une fois les fichiers « Reduce » écris, nous les rapatrions tous sur notre machine via la fonction « **copyReduces** » qui copie chaque fichier « Reduce » de toutes les machines sur notre machine avec la commande shell suivante :
'scp -r amichalewicz@' + machine + ':/tmp/amichalewicz/reduces/ /maMachine/tmp/amichalewicz'

Nous lançons cette fonction en parallèle sur toutes les machines avec la fonction « **map** » de la librairie python ; cela permet de rapatrier en même temps tous les fichiers « Reduce » de toutes les machines.
Pour afficher le résultat, nous le faisons grâce à la fonction « **afficheReduce** » qui affiche simplement le résultat du Map-Reduce en lisant chaque fichier REDUCE et en affichant son contenu.

b) Propositions d'optimisation

Regardons les résultats du MAP-REDUCE sur des fichiers plus volumineux et essayons de proposer des optimisations à celui-ci :

Fichier	**Taille**	**MAP**	**SHUFFLE**	**REDUCE**
input.txt	43oct	798ms	2081ms	776ms
forestier_mayotte.txt	1ko	1176ms	14027ms Timeout sur quelques machines	2175ms
deontologie_police_nationale.txt	7ko	2385ms	44654ms Beaucoup de Timeout	287ms
domaine_public_fluvial.txt	71ko	9428ms	176031ms Énormément de timeout lors du SHUFFLE	9783ms

sante_publique.txt	18,1 Mo	Time Out Exception après plus de 3h d'exécution
CC-MAIN-20170322212949-00140-ip-10-233-31-227.ec2.internal.warc.wet	398,8 Mo	

Nous pouvons remarquer que pour des fichiers de taille importante, la phase de SHUFFLE pose des problèmes de Timeout, c'est-à-dire que les communications entre les machines prennent trop de temps. Cela nous renvoie donc au final une MAP-REDUCE incomplet.

Optimisation n°1 :

- Lors de l'étape 10, nous avons évoqué la difficulté lorsque, nous testons le split sur un fichier de taille assez importante. En effet pour les fichiers « domaine_public_fluvial.txt », « sante_publique.txt » et « CC-MAIN-20170322212949-00140-ip-10-233-31-227.ec2.internal.warc.wet », le nombre de split est plus grand que le nombre de machines disponibles. Nous envoyons alors plusieurs splits sur une machine, pour ce faire nous créons une liste de machine ayant pour longueur le nombre de splits, il suffit de repartir de la première machine disponible lorsque nous atteignons la dernière disponible. Par exemple, nous allons savoir sur quelle machine envoyer chaque splits grâce à cette liste :
  ['tp-1a201-01', 'tp-1a201-02', 'tp-1a201-03', 'tp-1a201-04', 'tp-1a201-05', 'tp-1a201-06', … , 'tp-1a260-29', 'tp-1a260-30', 'tp-1a260-31', 'tp-1a260-32', 'tp-1a260-33', 'tp-1a201-01', 'tp-1a201-02', 'tp-1a201-03', 'tp-1a201-04', 'tp-1a201-05', …,'tp-1a260-32', 'tp-1a260-33', 'tp-1a201-01', 'tp-1a201-02', …].
  La machine 'tp-1a201-01' étant la première disponible et la machine 'tp-1a260-33' étant la dernière.

Optimisation n°2 :

- Lors de la phase de MAP, nous avons par exemple pour un des fichiers UMi.txt « Car 1 » écrit plusieurs fois, une solution pourrait être de calculer un « mini-REDUCE » en renvoyant dans le SHUFFLE non pas plusieurs fois « Car 1 » mais un seul « Car 2 ». A noter que pour un simple WordCount cela fonctionnerait car nous n'effectuons qu'une simple addition d'un nombre de mots mais pour d'autres fonctions plus complexes (ex : racine carré, carré, fonction trigonométrique…) cela ne fonctionnerait plus, car la linéarité n'est plus respectée.

Optimisation n°3 :

- Le streaming : MAP-REDUCE fonctionne par Batch, c'est-à-dire que nous attendons la fin de chaque étape (MAP, SHUFFLE, REDUCE). Nous pourrons penser à un système de streaming où l'on n'aurait pas besoin d'attendre que chaque étape soit terminée sur toutes les machines.

Optimisation n°4 :

- Il faudrait non pas que la communication entre les machines se fasse en ssh mais par une technologie plus rapide. Une communication en ssd entre les machines serait plus

rapide, cela réduirait le temps de SHUFFLE et l'envoi de chaque UMi.txt à chaque machine adéquate (la technologie SPARK utilise cette communication en ssd).

Optimisation n°5 :

- Paralléliser encore plus le programme SLAVE.py et MASTER.py et augmenter le nombre de machines disponibles. Une parallélisation encore plus importante des codes sources SLAVE.py et MASTER.py, pourrait faire gagner du temps, ainsi qu'augmenter le nombre de machine sur lesquelles faire tourner le MAP-REDUCE. Mais selon la Loi d'Amdhal, même avec une très forte parallélisation du code, on aurait une hausse de performance mais l'on atteindrait un plafond à un certain moment (le nombre de machines nécessaires ne serait pas beaucoup plus grand que ce que l'on possède déjà).

Conclusion

En comparant nos résultats en séquentiel avec ceux de notre système distribué, nous avons mis en évidence les faiblesses du système MAP-Reduce qui réside dans la phase de SHUFFLE. C'est dans de la communication entre machines que réside le problème, c'est cette étape qu'il faut chercher à optimiser (les solutions proposées ont été évoquées plus haut).

De plus, le fait d'avoir pris en compte la possibilité de pannes de certaines machines (grâce essentiellement à des try-except lors des étapes principales du MAP-REDUCE), cela a rendu notre code robuste. En effet nous pouvons le voir lorsque nous avons tester le MAP-REDUCE sur les fichiers :

- « domaine_public_fluvial.txt »
- « sante_publique.txt »
- « CC-MAIN-20170322212949-00140-ip-10-233-31-227.ec2.internal.warc.wet »

De nombreux time-out ont été généré mais, le MAP-REDUCE continuait de tourner malgré les erreurs et les timeouts générés. Pour les fichiers d'input « forestier_mayotte.txt » et « deontologie_police_nationale.txt », un résultat est retourné malgré la génération de timeout. Seule les UMi.txt des machines ayant générées des time-out ne sont pas dans le résultat final.