INF727 - Systèmes Répartis

Louis ROSE

6 Décembre 2020

1 Introduction

Lors de ce projet, on implémente MapReduce à l'aide du langage Python. Pour appliquer MapReduce à des fichiers volumineux, on utilisera une architecture de calcul répartie. En faisant varier la taille des fichiers et le nombre de machines utilisées, on pourra tirer des conclusions sur les liens entre temps de calcul, systèmes répartis et fonctionnement séquentiel versus parallèle (cf loi d'Amdahl).

2 Extraits des Étapes 1 à 9

J'ai souhaité rédiger cette première partie afin de "consolider" certaines choses apprises lors de ce projet. Cette section à pour but de condenser une partie des enseignements retirés des étapes 1 à 9. Pour plus de simplicité, les points appris qui sont réutilisés dans les étapes suivantes du projet ne sont pas mentionnés ici.

Quelques raccourcis Vim :

- :w pour sauvegarder, :q pour quitter, :n pour accéder à la n-ième ligne, v pour sélectionner, yy pour copier, p pour coller, gg pour accéder au début du document, G pour accéder à la fin du document, ?x pour chercher un mot dans le document.

Quelques commandes bash :

- hostname : renvoie le nom court de la machine utilisée. -d : nom de domaine. -f : nom long. -i : renvoie l'adresse ip de la machine.
- nslookup : permet d'obtenir le nom d'une machine à partir de son adresse IP, ou vice-versa.
- ip addr : renvoie l'ensemble des adresses ip de la machine : privée, publique, IPv4, IPv6...
- curl ifconfig.me renvoie l'adresse ip publique de la machine, tirée du site ifconfig.me via le client cURL.

- echo " 17 - 8" — bc -l : permet d'effectuer des calculs en ligne de commande, via Basic Calculator.

Quelques modules de python :

- os : os.system pour lancer une commande dans le terminal depuis un programme python. os.listdir pour récupérer une liste des fichiers contenus dans un dossier de la machine.
- subprocess : subprocess.check output(), qui permet de récupérer sous format de string le retour d'une commande de terminal.
- sys : sys.argv, qui permet de récupérer au sein d'un programme .py les arguments passés en ligne de commande.
- time : avec time.time() (qui renvoie le nombre de secondes écoulées depuis le 01/01/1970), on est en mesure de chronométrer le temps d'éxécution d'un programme Python.
- multiprocessing (as mp) : mp.cpu count() qui permet de connaitre le nombre de processeurs de la machine, mp.Pool, qui permet à l'aide de .map de lancer plusieurs processus en même temps sur une machine.

Protocole SSH et Clé Publique/Clé Privée :

Le protocole SSH permet à des machines distantes de communiquer. On parle usuellement de "client" pour la machine locale, et de "serveur" pour la machine distante. De manière à garantir la confidentialité des informations communiquées, on utilise un système de clés. (Un exemple d'attaque possible est : la technique du "middle man").

Supposons que l'on décide d'utiliser un système de clés pour communiquer avec un serveur. Chaque machine possède un clé publique (qui est accessible à tous, c'est son "empreinte digitale") et une clé privée. Lorsqu'on envoie un message, on chiffre ce dernier à l'aide de la clé publique du destinataire. Ainsi, il est le seul à pouvoir déchiffrer le message, étant le seul possesseur de la clé privée associée. Ainsi, la communication est sécurisée.

- ssh-keygen : permet de générer une clé privée et une clé publique pour notre machine. Les deux clés sont sauvés dans un dossier .ssh. (exemple et exemple.pub)
- ssh-copy-id .ssh/exemple.pub user@IPserveur : permet d'envoyer sa clé publique au serveur.

Dans le dossier .ssh, on trouve le dossier authorized keys, qui contient les clés publiques des machines considérées comme "sûres". Ainsi, on n'a pas besoin de rentrer son mot de passe pour se connecter via ssh à ces machines, seule une passphrase (optionnelle) est demandée.

- ssh -add : permet d'autoriser temporairement des clés publiques. Durée de conservation par défaut : le temps de la session. -t : durée de conservation. -l : donne la liste des clés en mémoire.

Programmes CLEAN.py et DEPLOY.py :

Ces deux programmes propres au projet ont respectivement pour principe de supprimer les dossiers temporaires (/tmp/lrose/) des machines distantes utilisées, et d'envoyer le programme SLAVE.py dans les dossiers temporaires de ces machines. Ces programmes nous permettent de souligner deux points importants :

- Connexions en parallèle plutôt qu'en séquentiel : Pour ces deux programmes, l'ensemble des instructions sont envoyées à partir de l'ordinateur local. On les envoie en séquentiel, c'est à dire que l'on met à profit l'ensemble des processeurs de la machine (ici, 12), afin d'envoyer ces instructions en parallèle. Pour cela, on utilise le module multiprocessing :

- S'assurer de l'exécution d'une commande avant d'en lancer une seconde : Dans le cas du programme DEPLOY.py, on veut être certains que le dossier temporaire personnel est créé, avant d'y copier le programme SLAVE.py. Pour cela, on utilise aussi le module multiprocessing : on créé un premier pool d'instructions exclusivement pour la création des dossiers, et un second pour la copie du programme SLAVE.

3 Structure du MapReduce Implémenté

Cette implémentation de MapReduce est exécutée depuis le terminal à l'aide de la commande suivante :

python3 MASTER.py file.txt n

avec "file.txt" le fichier sur lequel on souhaite appliquer un WordCount, et "n" le nombre de machines distantes qu'on veut utiliser pour cela.

Le programme MASTER.py va tout d'abord lancer le programme mach.py, qui renvoie une liste de n machines distantes. Le MASTER.py va ensuite tester si ces machines sont opérationnelles en lançant la commande "hostname" à distance.

Les programmes CLEAN.py et DEPLOY.PY sont ensuite lancés : ainsi, on supprime l'ensemble des fichiers temporaires personnels sur chacune des machines distantes opérationnelles, et on recrée ce dossier temporaire avant d'y déployer le programme SLAVE.py, qui sera utilisé sur et par les machines distantes. Le multiprocessing est utilisé pour lancer ces 2 * n commandes, avec n le nombre de machines distantes utilisées.

Ensuite, MASTER.py exécute l'étape de SPLIT. Le fichier.txt est séparé en n parts de taille égale (en nombre de lignes) : chaque fraction du fichier est envoyée à 1 machine distante (Sx.txt). La machine principale gère l'ensemble de ces envois, on utilise le multiprocessing pour cette étape (notre machine compte 12 processeurs).

Suite à cela, c'est l'étape de MAP qui est faite (SLAVE.py (0) sur l'ensemble des machines distantes opérationnelles). Pour chaque fichier Sx.txt, un fichier UMx.txt est créé, qui présente une ligne par mot présent dans Sx.txt, suivi du chiffre 1. C'est depuis la machine principale que les exécutions à distance sont lancées.

On exécute ensuite SLAVE.py (1) sur les machines distantes : c'est l'étape de SHUFFLE. Pour chaque mot/Unsorted Map, un Hash est calculé, qui détermine la machine distante auquel il sera envoyé. Cette Etape compte de nombreux échanges données : plus d'1 fichier envoyé par mot du fichier initial ! (comme un même mot peut se trouver dans des splits différents, la découpe des splits étant "aveugle" au contenu du fichier).

Enfin, l'étape de REDUCE est suivie sur l'ensemble des machines distantes, avec le programme SLAVE.py (2). Là, les occurrences des mots sont additionnées sur chaque machine distante. Enfin, les données sont regroupées, et transmises à la machine principale, qui les regroupe une seconde fois, dans un fichier results.txt.

4 Étapes 10 à 13 en détail

Cette étape à pour objectif de répondre à plusieurs questions spécifiques, qui sont posées au sein de l'énoncé du projet entre les étapes 10 et 13.

- S'assurer de l'exécution d'une commande avant d'en lancer une seconde :

Que ce soit dans l'étape de SPLIT, celle de MAP, ou celle de SHUFFLE, on créé des dossiers que l'on remplit immédiatement. On veut donc s'assurer que ces dossier sont créés avant d'y copier des fichiers, au risque de perdre ces derniers. Pour cela, on utilise le multiprocessing: On créé un premier pool d'instructions

qui ne font QUE créér les dossiers demandés. Le(s) pool(s) suivant(s) y copient les fichiers pertinents. Comme un pool d'instructions ne peut démarrer que lorsque le précédent est terminé, on à la sécurité demandée.

code générique utilisé :

```
import multiprocessing as mp
pool = mp.Pool(n)
for batch in batchs : pool.map(fonction, batch)
```

- Lancement des SLAVES en parallèle :

Lors des étapes de MAP et de SHUFFLE, on veut lancer l'éxécution des SLAVES en parallèle sur les machines distantes. Pour cela, on utilise le multiprocessing sur la machine principale : sur chaque processeur, on va lancer l'exécution à distance du SLAVE.py via la commande ssh lrose@machinedistante python3 SLAVE.py ...

- Comment s'assurer que les SLAVES ont bien terminé ?

Cette question est posée lors des étapes de MAP et de REDUCE. Là encore, c'est le module multiprocessing qui nous apporte la réponse : le lancement des SLAVES.py en pool nous permet de nous assurer de leur exécution complète avant que les processeurs concernés puissent passer aux instructions suivantes.

5 Performances : Constats et Analyses

Une fois mon implémentation de MapReduce terminée, j'ai pu tester et chronométrer mon programme sur les différents fichiers vus en Etape 1 : forestier mayotte.txt (71 lignes) deontologie police nationale.txt (205 lignes) domaine public fluvial.txt (841 lignes) forestier.txt (20531 lignes)

Les deux variables jouant sur le temps d'exécution sont la taille du fichier, et le nombre de machines distantes utilisées (n). J'ai conduit mes tests avec ces valeurs de n : 1, 2, 4, 6, 8, 10, 12, 24. Voici les conclusions tirées de ces tests :

Constats :

- L'Etape prenant le plus de temps dans la majorité des cas est le SHUFFLE. Ensuite viennent l'ETAPE 0 (tests des machines distantes, CLEAN et DEPLOY) et le REDUCE. Enfin, les étapes de SPLIT et de MAP ont une durée limitée. Les autres Etapes (envoi de la liste des machines et concaténation des résultats) ont une durée négligeable.

- La durée de l'étape de SHUFFLE augmente avec la taille du fichier traité, et diminue avec le nombre de machines distantes (jusqu'à un certain point).

- La durée des étapes de CLEAN DEPLOY, SPLIT, et MAP augmente fortement avec le nombre de machines distantes utilisées, et est peu impactée par la taille du fichier traité.

- La durée de l'étape de REDUCE augmente avec le nombre de machines distantes, et la taille du fichier traité

Donc :

- Avec un nombre faible de machines, toutes les étapes prennent très peu de temps, sauf l'Etape de SHUFFLE, qui prend un temps très long.

- Lorsque on augmente le nombre de machines au début, la durée du SHUFFLE diminue, et celle des CLEAN, DEPLOY, SPLIT, MAP, et REDUCE augmente.

- En passant de 1 à 2 machines distantes, la durée du SHUFFLE peut diminuer de 20 pourcents ! Cette baisse ralentit au fur et à mesure.

- A partir d'un certain seuil, la durée du SHUFFLE commence à moins diminuer. Alors, la durée totale du MapReduce augmente de nouveau avec le nombre de machines : on a dépassé le"n" optimal.

- Le nombre de machines distantes optimal dépend de la taille du fichier : pour les quatre fichiers étudiés, il vaut respectivement : 1, 4, 6, et 12.

- Résultats obtenus (en secondes), par nombre de machines distantes

- for. mayotte : 32 — 43 — 75 —100— 138—141—148—309
- dé. pol. nat : 164 —146— 139—148—158—180—190—425
- dom. fluvial : 496 —403— 338—297—297—286—291—499
- forestier .txt : xx–xxx–xxx–xxx–xx—1718—1537 1460—1866

Analyses :

- Les opérations effectuées par ce MAPREDUCE peuvent être classées en deux catégories : les opérations de calcul (sur une machine donnée), et les transferts de données (entre deux machines). Avec l'augmentation du nombre de machines distantes, la quantité d'informations à traiter par machine diminue, mais le volume d'échange entre machines augmente.

- Ainsi, on comprend bien pourquoi les Etapes de CLEAN, DEPLOY, et RE-

DUCE prennent plus de temps lorsque n augmente : ces étapes impliquent des échanges de données (via ssh ou scp), et sont de toute manière peu lourdes en termes de calculs.

- Réciproquement, on comprend que la durée du SHUFFLE puisse diminuer lorsque n augmente : cette étape est fortement calculatoire. Mais cette étape comprend aussi des échanges de données (les envois des shuffles). Ainsi, il y a bien un stade où la réduction du temps de calcul est trop faible par rapport à l'augmentation du temps d'échange de données.

- On observe que pour les quatre fichiers, le temps d'exécution avec n = 24 est supérieur au temps d'exécution pour n = 12. Ce n'est pas trop surprenant pour les trois premiers fichiers (dont les n optimaux sont respectivement 1, 4 et 6). Mais pour le fichier forestier.txt, c'est digne d'être mentionné : son n optimal (parmi les n évalués!) étant 12 (le nombre de processeurs dont on dispose par machine), il y a fort à parier qu'avec plus de processeurs (par exemple, 14), son temps d'exécution global pourrait encore être réduit. Mais avec n = 24 ce n'est pas le cas ici. La raison : nos processus sont synchrones, ce qui signifie qu'à chaque étape concernée, les 12 premier processus doivent être terminés pour que les 12 suivants se lancent. Dès lors, quel intérêt de séparer les données traitées en deux, en passant de n = 12 à n = 24 ? Il n'y en a pas : on augmente le nombre d'échanges de données, sans réduire le temps total d'exécution des calculs.

6 Pistes d'amélioration

Afin d'améliorer la performance du MapReduce, j'ai envisagé plusieurs pistes, que je n'ai pas pu suivre par manque de temps. Les voici :

- Utiliser le multiprocessing sur les machines distantes, afin d'accélerer l'exécution du programme SLAVE.py (étapes de MAP, SHUFFLE, et REDUCE).

- Modifier l'étape de SHUFFLE pour n'envoyer que chaque machine n'envoie qu'un seul fichier à chaque autre machine, au lieu d'un fichier par mot. (Cela aurait aussi nécessité la modification de l'étape de REDUCE, afin de pouvoir gérer des fichiers contenant plusieurs mots).

- Utiliser le module multithreading en plus du module multiprocessing, afin d'accélerer l'exécution du programme SLAVE.py sur les machines distantes : pouvoir créer différents threads au sein de chaque processus aurait été utile.

- Implémenter de l'asynchronicité dans mon MapReduce, afin de pouvoir utiliser intelligemment plus de 12 machines distantes.

7 Conclusion

Ce projet d'implémentation de MapReduce m'a permis de progresser dans de nombreux domaines, à la fois pratiques et théoriques, à savoir : les commandes Linux, la programmation en Python, l'utilisation de la documentation Python/Linux/Vi, l'utilisation de machines distantes et de NFS, la compréhension du concept MapReduce, la notion de calcul réparti et de multiprocessing, et enfin, les notions de complexité d'un programme et de temps d'exécution. J'ai aussi pu avoir un premier aperçu des notions de synchronicité/asynchronicité et de multithreading.