SAUVEUR Mathieu
MS BGD

RAPPORT DE TP

SYSTEMES REPARTIS POUR LE BIG DATA, HADOOP MAPREDUCE FROM SCRATCH

Table des matières

SOMMAIRE 2

I. Programmation séquentielle 3

1) Première iteration du programme sequentiel et test sur des fichiers de petite taille 3

2) Deuxième itération avec fichiers de taille moyenne..... 4

3) Troisième itération avec des fichiers de grandes tailles 5

4) Synthèse des parties précédentes 5

II. Travail en réseau 8

1) Travailler avec plusieurs ordinateurs en réseau..... 8

2) Travailler avec des fichiers locaux / avec un server NFS 9

III. Programmation distribuée : Implémentation de MAPREDUCE d'Hadoop from scratch 10

1) Le MapReduce du framework Hadoop..... 10

2) Structure du dossier rendu comportant le code..... 10

3) Détail des différents modules qui composent le code. 12

4) Procédure d'exécution..... 19

5) Premiers résultats 22

6) Démonstration empirique de la loi d'Amdahl 23

7) Limites atteintes, problèmes rencontrées et perspectives d'amélioration de l'architecture du code 26

I. PROGRAMMATION SEQUENTIELLE

Un des objectifs finaux de ce TP étant de justifier de la pertinence de l'utilisation de systèmes distribués notamment par rapport à des exécutions de programmes de manière séquentielle, nous allons dans cette partie effectuer des jobs s'apparentant au mapreduce du framework hadoop mais menés sur une seule machine afin de d'évaluer les résultats et faire nos premières observations.

1) Première iteration du programme sequentiel et test sur des fichiers de petite taille

Tout d'abord, la structure de donnée la plus pertinente pour stocker les résultats est la Hashmap en Java et les dictionnaires en Python. Il y a 3 raisons à cela :

- Elles permettent toutes deux (Hashmap et Dict), un accès en temps constant aux données que ce soit pour insérer, supprimer et trouver des éléments. Cela est dû à leur implémentation par des tables de hashage qui induisent donc une complexité de O(1) pour réaliser les opérations citées.
- Leurs clés sont uniques, primordial afin de compter des mots, vu que chaque mot est représenté une seule fois.
- Leur mise à jour est très facile car la vérification de la présence ou la non-présence d'un est réalisée très aisément.

Après avoir réaliser le programme pour un **comptage de mot séquentiel pur**, on le test sur un fichier input.txt avec les mots suivants :

```
Deer Beer River
Car Car River
Deer Car Beer
```

J'obtiens le résultat ci-dessous qui correspond bien à ce qu'on attend du programme :

```
Deer: 2
Beer: 2
River: 2
Car: 3
Nombre de mots uniques : 4
```

Ensuite on implémente **un tri par nombre d'occurrences des mots** puis on test avec le même fichier input.txt. J'obtiens le résultat ci-dessous :

```
Car: 3,
Deer: 2,
Beer: 2,
River: 2,
Nombre de mots uniques : 4
```

Ensuite on implémente **un tri par ordre alphabétique pour les mots qui ont le même nombre d'occurrence**. Avec le même fichier on a le résultat suivant :

```
Car: 3,
Beer: 2,
Deer: 2,
River: 2,
Nombre de mots uniques : 4
```

2) Deuxième itération avec fichiers de taille moyenne

Dans cette partie nous avons tester notre programme avec d'autres fichiers plus volumineux en entrée provenant de *https://github.com/legifrance/Les-codes-en-vigueur*

On utilise d'abord déontologie_police_nationale.txt (456 mots uniques). Les 5 premiers mots parmi les 50 premiers de la liste triée résultat sont :

```
matsvr@DESKTOP-7883LKU:~/RenduTP_Mathieu_SAUVEUR$ python3 sequentiel.py
Les 50 premiers mots : [('de', 86), ('la', 40), ('police', 29), ('et', 27), ('à', 25), ('des', 24), ('les', 23), ('Artic
le', 19), ('le', 19), ('', 19), ('nationale', 15), ('en', 13), ('est', 13), ('DE', 12), ('TITRE', 12), ('ou', 11), ('qu
i', 11), (':', 9), ('DES', 9), ('aux', 9), ('fonctionnaire', 9), ('par', 9), ('Le', 8), ('leur', 8), ('ses', 8), ('au',
7), ('déontologie', 7), ('ne', 7), ('Code', 6), ('DEVOIRS', 6), ('ET', 6), ('FONCTIONNAIRES', 6), ('LA', 6), ('POLICE',
6), ('dans', 6), ("l'autorité", 6), ('a', 5), ('cas', 5), ('du', 5), ('faire', 5), ("l'ordre", 5), ('ordres', 5), ('pour
', 5), ('sa', 5), ('se', 5), ('sont', 5), ('Il', 4), ('La', 4), ('doit', 4), ('elle', 4)]
Nombre de mots uniques : 456
Temps d'execution pour le comptage des occurences : 0.0006649494171142578
Temps d'execution pour le tri : 0.0005269050598144531
```

On répète la même chose pour le fichier domaine_public_fluvial.txt (1493 mots uniques) et on obtient :

```
matsvr@DESKTOP-7883LKU:~/RenduTP_Mathieu_SAUVEUR$ python3 sequentiel.py
Les 50 premiers mots : [('de', 621), ('le', 373), ('du', 347), ('la', 330), ('et', 266), ('à', 209), ('des', 208), ('les
', 200), ('est', 169), ('dans', 136), ('par', 122), ('sur', 119), ('ou', 118), ('Article', 103), ('', 103), ('au', 97),
 ('en', 95), ('bateau', 74), ('tribunal', 72), ('pour', 69), ("l'article", 68), ('aux', 66), ('un', 61), ('lieu', 60), (
';', 56), ('qui', 54), ('Le', 52), (':', 50), ('bureau', 50), ('que', 46), ("d'un", 44), ('navigation', 43), ('où', 43),
 ('son', 43), ('bateaux', 41), ('se', 41), ("d'immatriculation", 40), ('domicile', 38), ('a', 37), ('juge', 36), ('La',
34), ('délai', 34), ('propriétaire', 34), ('Il', 33), ('il', 33), ('Les', 32), ('créanciers', 32), ('titre', 32), ('être
', 31), ('code', 30)]
Nombre de mots uniques : 1493
Temps d'execution pour le comptage des occurences : 0.004334926605224609
Temps d'execution pour le tri : 0.0841670036315918
```

On répète la même chose pour le fichier sante_publique.txt (52 180 mots uniques) et on obtient :

```
matsvr@DESKTOP-7883LKU:~/RenduTP_Mathieu_SAUVEUR$ python3 sequentiel.py
Les 50 premiers mots : [('de', 189699), ('la', 74433), ('des', 66705), ('à', 65462), ('et', 60940), ('les', 48772), ('du
', 48400), ('le', 44264), ('ou', 39464), ('par', 30224), ('en', 28922), ('au', 25026), ("l'article", 22772), ('dans', 22
570), ('Article', 21878), ('', 21810), ('est', 21258), (';', 19388), ('L.', 18410), ('un', 17594), (':', 16166), ('aux'
, 15874), ('pour', 14426), ('santé', 14423), ('sont', 14324), ('Les', 13532), ('une', 12720), ('sur', 12482), ('Le', 123
80), ('R.', 11224), ('que', 11002), ('qui', 9726), ("d'un", 9464), ('directeur', 8248), ('être', 8046), ('La', 7854), ('
peut', 7824), ('conditions', 7656), ('ne', 7574), ("d'une", 7276), ('son', 7206), ('cas', 6922), ('général', 6628), ('le
ur', 5670), ('1°', 5648), ('pas', 5596), ('2°', 5470), ('conseil', 5370), ('sécurité', 5154), ('dispositions', 5152)]
Nombre de mots uniques : 52180
Temps d'execution pour le comptage des occurences : 0.8140206336975098
Temps d'execution pour le tri : 0.5439491271972656
```

À la fin, on a rajouté un chronomètre pour savoir en combien de temps chaque étape (comptage des occurrences et tri) ont pris. Sur les screenshots ci-avant, on peut observer ces temps sur les 2 dernières lignes.

3) Troisième itération avec des fichiers de grandes tailles

Maintenant, on va travailler sur d'encore plus gros fichiers. Pour cela, nous sommes allés récupérer des pages internet au format texte stockées sur *http://commoncrawl.org*.

Nous avons donc utilisé des fichiers d'environ 350Mo au format WET (sans images) que nous avons au préalable décompressé avec la commande gzip.

Après avoir exécuté le programme sur un fichier de 344 Mo (4 196 682 mots uniques), nous avons les résultats suivants :

```
matsvr@DESKTOP-7883LKU:~/RenduTP_Mathieu_SAUVEUR$ python3 sequentiel.py
Les 50 premiers mots : [('the', 372139), ('to', 297443), ('and', 294423), ('-', 260544), ('de', 256556), ('of', 255417),
 ('a', 251449), ('in', 221704), ('for', 130604), ('&', 122495), ('is', 109833), ('|', 104551), ('on', 90507), ('The', 82
360), ('la', 82022), ('/', 79599), ('with', 77399), ('(1)', 75880), ('en', 72596), ('you', 71861), ('-', 70527), ('by',
69665), ('your', 67779), ('и', 66190), ('that', 62265), ('1', 59290), ('are', 56229), ('I', 55800), ('or', 54677), ('в',
 52936), ('►', 52808), ('0', 51857), ('2023', 51519), ('this', 46100), ('at', 45963), ('be', 45390), ('as', 44718), ('2'
, 44635), (':', 44576), ('2022', 42651), ('=', 41339), ('und', 40692), ('from', 40239), ('y', 39924), ('conversion', 393
80), ('Content-Type:', 39107), ('text/plain', 39105), ('Content-Length:', 39101), ('WARC-Date:', 39101), ('WARC-Record-I
D:', 39101)]
Nombre de mots uniques : 4196682
Temps d'execution pour le comptage des occurences : 19.37468647956848
Temps d'execution pour le tri : 13.45397686958313
```

Le temps d'exécution total du programme est d'environ 35 secondes.

4) Synthèse des parties précédentes

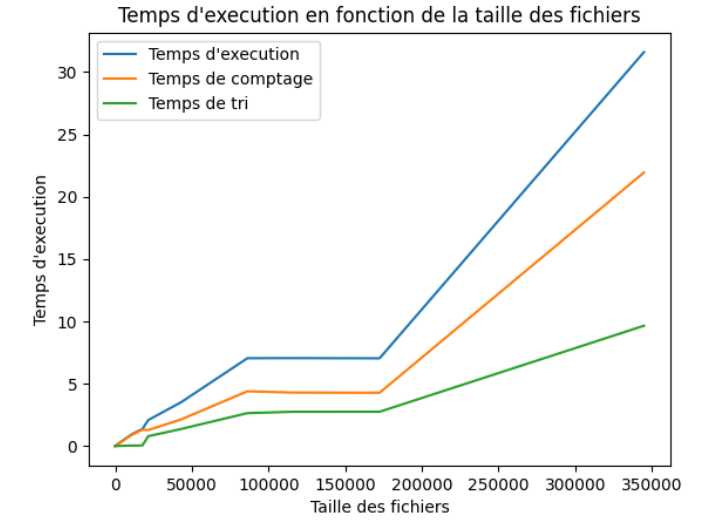
Pour synthétiser les observations que nous avons pu faire précédemment et en tirer une tendance, on veut réaliser un graphique montrant le temps de calcul en fonction de la taille du fichier lu par le programme.

J'ai utilisé 13 tailles de fichier différentes :

1. input.txt (**1ko**)
2. forestier_mayotte.txt (**2ko**)
3. deontologie_police_nationale.txt (**8ko**)
4. domaine_public_fluvial.txt (**70ko**)
5. procedure_civile.txt (**1.5Mo**)
6. travail.txt (**1Mo**)
7. sante_publique.txt (**18Mo**)
8. Fichier warc.wet divisé en 16 (**22Mo**)
9. Fichier warc.wet divisé en 8 (**43Mo**)

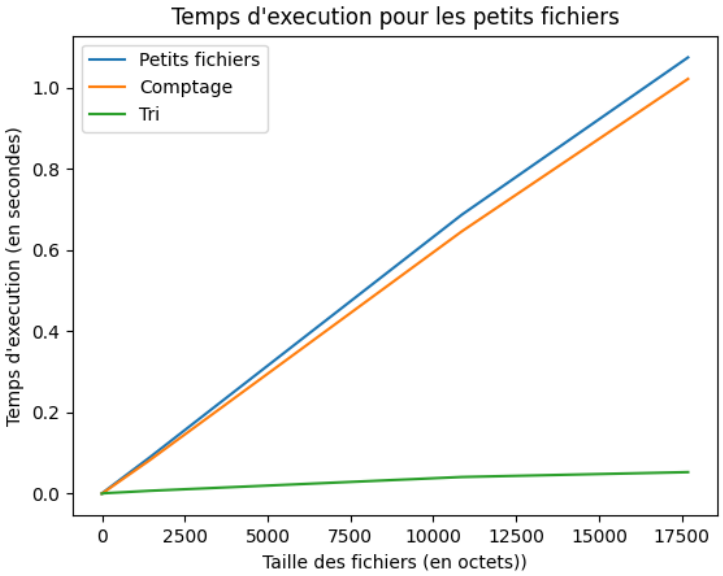
10. Fichier warc.wet divisé en 4 (**88Mo**)
11. Fichier warc.wet divisé en 3 (**118Mo**)
12. Fichier warc.wet divisé en 2 (**177Mo**)
13. Fichier warc.wet entier (**344Mo**)

Les temps d'exécutions peuvent être retrouvés dans le module graphique.py dans le dossier de rendu. Le graphe obtenu est le suivant :



Sur le graphique on peut observer 4 phases :

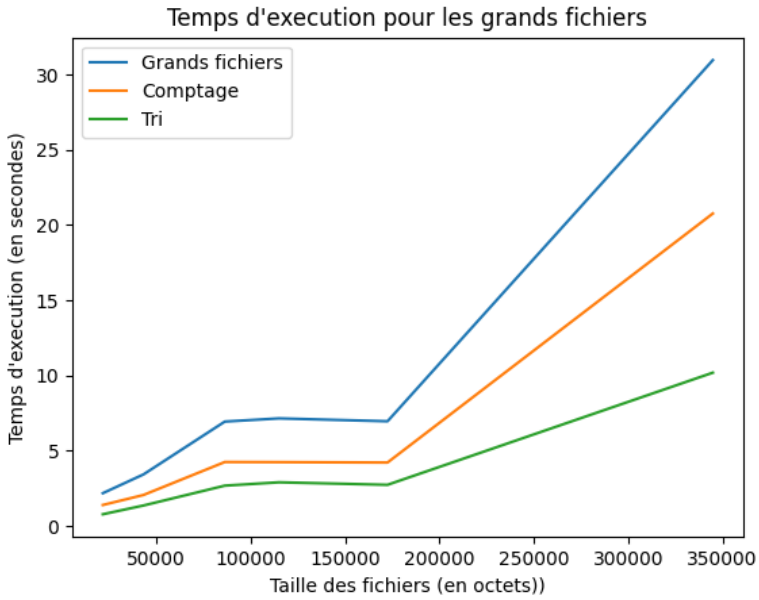
- Une première phase au niveau des très petits fichiers dans laquelle le temps de d'exécution est quasiment également au temps de comptage des occurrences de mots (temps de tri négligeable en comparaison.



- Une seconde phase durant laquelle la phase de tri reprend un peu de poids dans le temps d'exécution globale avec une allure générale croissante.

- Un plateau soudain entre 88ko et 200ko environ, durant lequel le temps d'exécution est stable (ce résultat est peut-être dû au manque de discrétisation de la courbe, pas assez de points, ou encore à la partie aléatoire des résultats obtenues due à des évènements inopinés survenant localement sur la machine personnelle comme un job très lourd demandant beaucoup de ressources tournant en arrière-plan

- Une quatrième phase où les temps augmentent de manière encore plus significative que dans la seconde phase.



Les observations que l'on peut faire sont donc que non, le temps de calcul n'est pas linéaire en fonction de la taille de fichier. Par ailleurs, on peut très bien supposer que cela le devient si on continue d'observer pour des fichiers encore plus gros.

On peut donc estimer que le temps de calcul est linéaire pour de plus gros fichier et ce qui peut nous permettre d'extrapoler ce temps de calcul pour de gros fichiers (comme ceux étudiés dans la patie 3)

Remarque : Dans la suite du TP j'ai préférer faire tourner le programme séquentiel.py sur les grros fichiers afin de pouvoir les comparer directement avec les performances des clusters mis en place.

II. TRAVAIL EN RESEAU

Dans cette partie nous allons aborder quelques notions de réseau qui nous seront utiles dans l'élaboration de systèmes distribués plus tard.

1) Travailler avec plusieurs ordinateurs en réseau

Pour pouvoir travailler en réseau, il est nécessaire de connaître les identifiants des machines que nous utilisons afin de pouvoir les appeler correctement.

Ma machine a un nom court qui est tp-1a201-13 que nous pouvons avoir avec la commande « hostname » et un nom long qui est tp-1a201-13@ssh.enst.fr (avec le nom de domaine) que nous pouvons avoir grâce à la commande nslookup.

La machine est également identifiable par son adresse ip qu'elle utilise. Pour connaître toutes les adresses ip de la machine, on peut utiliser la commande « ip a » ou « ip address ». Il existe également des sites internet pour connaître notre adresse ip comme « ipinfo.io » ou tous sites permettant de réaliser des speedtest de connexion internet.

Grâce au nom d'une machine, nous pouvons remonter à son adresse ip, en effectuant par exemple la commande « nslookup nom-de-l-ordinateur ». De même on peut connaître les noms associés à une adresse ip grâce à nslookup adresse-ip.

On peut réaliser des commandes ping pong entre machines afin de voir en combien de temps la connexion est réalisée entre elles.

On peut également réaliser des calculs aussi bien sur notre propre machine, mais également sur des machines à distance.

Pour calculer 2+3 sur sa propre machine, on peut utiliser « echo $((2+3)) » sur windows ou « expr 2 + 3 » sur Linux. À noter que nous pouvons utiliser un outil de calcul intégré que nous activons avec bc + « Entrée »

Cette fois-ci, pour un calcul à distance, il suffit de se connecter à la machine distante avec « ssh utilisateur@adresse_distant » puis de taper « echo $((2+3)) » vue avant. On peut par ailleurs utiliser l'outil de calcul sur la machine à distance avec ssh utilisateur@adresse_distant 'echo "2 + 3" | bc'

Aussi, pour lancer un calcul à distance en utilisant SSH sans taper le mot de passe et en une seule ligne de commande, il faut :

- Dans un premier temps, générer une paire de clés SSH (publique et privée) que j'ai déjà personnellement mais que nous pouvons avoir avec « ssh-keygen ».
- Ensuite, copier la clé publique sur l'ordinateur distant avec « ssh-copy-id utilisateur@adresse_distant » qui va ajouter ma clé publique au fichier « ~/.ssh/authorized_keys » à distance.
- Maintenant, se connecter à la machine à distance ne nécessite pas de mot de passe (notamment avec la commande vu avant : « ssh utilisateur@adresse_distant »).

Pour pouvoir faire un calcul sans taper de mot de passe il nous faut juste retaper « ssh utilisateur@adresse_distant "echo $((2+3))" ».

2) Travailler avec des fichiers locaux / avec un server NFS

Sur la machine à distance le chemin absolu de mon répertoire personnel est « /cal/exterieurs/msauveur-23 ». On le récupère en se mettant dans son répertoire personnel avec « cd » si on n'y est pas déjà puis avec « pwd » (print working directory).

Pour savoir sur quel disque du serveur un fichier est physiquement stocké, on peut utiliser la commande « df ». La commande df ~/perso.txt montre le système de fichiers sur lequel se trouve notre répertoire personnel et donc le fichier perso.txt. Cela permet d'avoir des informations comme le point de montage du système de fichiers, l'espace utilisé, l'espace libre, et autres.

Pour savoir où sont stockés physiquement les éléments, nous pouvons utiliser la commande « df /tmp/<nom d'utilisateur>/ ». Le système de fichiers mentionné dans la colonne "Filesystem" ou similaire est celui où le dossier est physiquement stocké.

Pour transférer un fichier de l'ordinateur A vers l'ordinateur B en utilisant scp, tout en s'assurant que les répertoires nécessaires existent, il suffit d'abord d'exécuter la commande « scp /tmp/local.txt utilisateur@adresse_B:/tmp/< nom d'utilisateur>/local.txt » dans le terminal. Si le répertoire /tmp/<nom d'utilisateur> n'existe pas sur l'ordinateur B, il y a l'erreur « no such file or directory ». Il faut donc créer ce dossier en se connectant à distance à la machine B et exécuter « mkdir -p /tmp/< nom d'utilisateur> » puis réitérer l'étape précédente.

« scp utilisateur@adresse_B:/tmp/<nom d'utilisateur>/local.txt /chemin/local/sur/A/local.txt »

III. PROGRAMMATION DISTRIBUEE : IMPLEMENTATION DE MAPREDUCE D'HADOOP FROM SCRATCH

Nous avons d'abord créé un programme de comptage d'occurrences de mots dans un fichiers et leur tri le tout en séquentiel. Puis nous avons revu quelques notions de communication entre machine en réseau utiles pour la mise en place d'un système distribué. Dans cette partie, nous allons pouvoir désormais éditer « from scratch » le MAPREDUCE d'Hadoop pour ensuite comparer les performances d'un programme séquentiel par rapport à une distribution des tâches mais aussi tenter de démontrer empiriquement la loi d'Amdahl qui est la loi qui définit le gain de rapidité potentiel d'un algorithme en fonction du nombre de cœurs utilisé pour faire tourner cet algorithme.

1) Le MapReduce du framework Hadoop

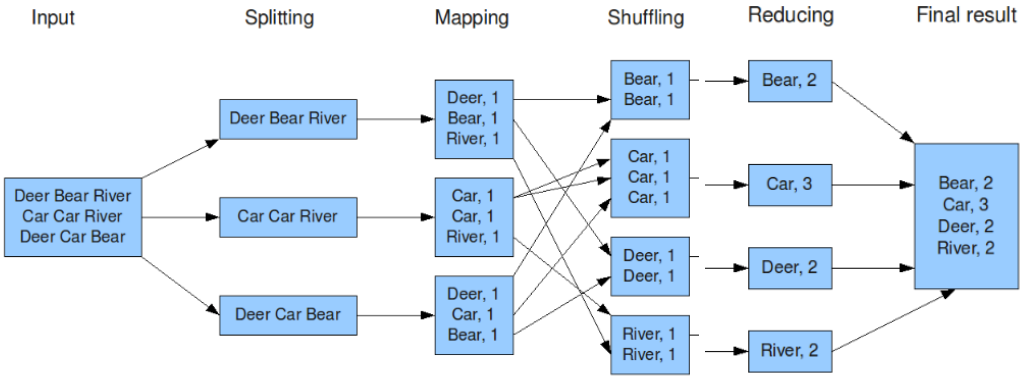
Le MapReduce du framework Hadoop est constitué de différentes étapes réalisés par les Workers qui constituent le cluster. Chaque worker effectue les différentes phases du MapReduce en parallèle, ce qui permet d'optimiser grandement dans certain cas l'exécution d'algorithmes sur de gros volumes de données ou avec une certaine complexité.

Il se compose d'une phase de split qui sépare un fichier initial en plusieurs fichiers répartis équitablement entre les workers.

Une phase de map dans laquelle chaque Worker reçoit les splits générés avant.

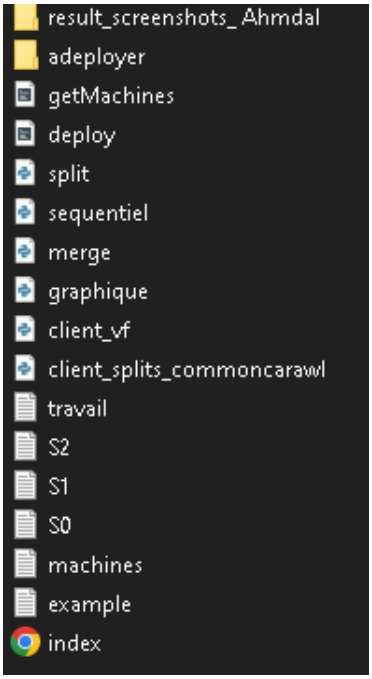
Une phase de shuffle qui va redistribuer les mots, en attribuant un mot unique à une unique machine. Lors de cette phase de shuffle, les workers s'envoient à chacun les mots dont ils sont en charge.

Une phase de reduce dans laquelle les Workers vont simplement compter les mots qu'ils ont obtenus du shuffle.



2) Structure du dossier rendu comportant le code

Le dossier remis nommé RenduTP_Mathieu_SAUVEUR est organisé de la façon suivante :

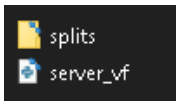


D'abord le fichier *sequentiel.py* ainsi que *graphique.py* sont les fichiers utilisés pour les premières parties du TP lors de la création du programme séquentiel. Dans le premier fichier se trouve tout le code nécessaire pour faire tourner ce programme avec des fichiers de tailles différentes. Dans le second, se trouve le code pour créer les graphiques exposés dans la première partie : il permet de regrouper tous les fichiers dans des listes et faire tourner l'algorithme séquentiel sur celles-ci tout en récupérant les temps de calcul.

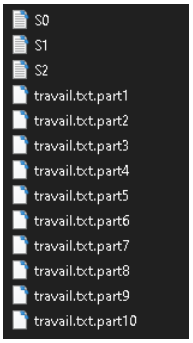
Ensuite nous avons le fichier *getMachines.sh* qui contient le script qui nous permettra d'aller regarder l'état de toutes les machines de TP (*index.html*) de l'école pour ensuite, après un shuffle, en récupérer un nombre définit à l'exécution par l'utilisateur. Leur adresse 'nom_machine.enst,fr' est inscrite à l'issue de l'exécution dans le fichier *machines.txt* ligne par ligne.

On a également un fichier *deploy.sh* qui va se charger de déployer sur toutes les machines de TP enregistrées dans *machines.txt* les fichiers dont ces dernières ont besoin pour pouvoir réaliser le mapreduce. Le script contenu dans le deploy.sh possède trois commandes distinctes qui vont :

- Tuer tout processus en cours d'exécution sur les machines puis va créer le dossier *adeployer* dans ses fichiers temporaires pour le préparer à recevoir une copie des fichiers locaux. (S'il y est déjà avant l'exécution de la commande, il est d'abord supprimé).
- Copier les fichiers présents dans le dossier *adeployer* localement.
- Lancer le module *serveur_vf.py* pour faire démarrer l'écoute des machines sur le port défini désormais en attente d'une connexion entamée plus tard par l'exécution du module *client_vf.py*.



Dans le dossier *adeployer* qui sera déployé sur toutes les machines, nous avons le module *serveur_vf.py* mais aussi un dossier *splits* qui contient les fichiers sur lesquels vont travailler les machines pour le mapreduce. C'est dans ce dossier *splits* que le fichier split.py dépose les n coupures d'un fichier initial.



Dans l'exemple de contenu du dossier splits, nous avons les fichiers ci-contre. Les parties de 1 à 10 (pour 10 machines de TP) du fichier travail.txt ont été générées automatiquement après l'exécution de split.py. Pour séparer un fichier il faut le copier au préalable dans le dossier splits, intégrer le nom de ce fichier dans le script de split au niveau du chemin d'accès, puis lancer le module. Les coupures sont générées et le fichier initial est supprimé pour éviter de déployer sur les machines virtuelles des fichiers inutiles.

Nous avons le module python *split.py* qui contient un script qui a pour but, après la récupération d'un nombre de machines n, de séparer un fichier en n parties pour préparer l'exécution du job mapréduce en distribué sur ces machines. Ce module se lance juste après le getMachines.sh.

Ensuite, il y a le module *merge.py* qui lui m'a permis de regrouper des fichiers pour en créer de nouveau de tailles encore plus grandes.

Enfin, il y a le module *client_vf.py*. C'est l'exécution de ce module sur une machine (qui est dite Master du cluster) qui va communiquer en ouvrant des sockets vers les machines virtuelles via un port défini pour leur envoyer et recevoir des paquets et leur donner des instructions pour finalement réaliser le job de mapreduce. Ce module client_vf.py utilise des splits créés localement à partir du module split.py (dans le dossier splits du dossier adeployer) puis envoyés aux Workers lors du lancement de deploy.sh.

On note qu'il y a un fichier nommé client_splits_commoncrawls.py qui est une autre version du client_vf mais qui demande aux Workers de récupérer directement des gros fichiers dans le répertoire local /cal/commoncrawl.

On dénote par ailleurs la présence de tous les fichiers .txt utilisés dans la première partie et qui nous seront utiles dans la suite.

3) Détail des différents modules qui composent le code

Dans cette partie, nous nous focaliserons principalement sur les modules client_vf.py et server_vf.py et la structure du script qui les compose car ce sont les deux modules principaux qui vont déterminer la gestion du déroulement des jobs MapReduce en système distribué.

CLIENT_VF.PY

Comme évoqué dans la partie précédente, le module client_vf.py et le module qui va être exécuté sur la machine Master et qui va organiser le séquencement des opérations MapReduce en envoyant

des messages aux différentes machines par des sockets afin de démarrer les différentes étapes et dans le bon ordre.

Le script dispose des parties ci-dessous :

- Les fonctions d'envoie et de réception de messages (communes à serveur_vf.py)

On les utilise lorsqu'on va vouloir envoyer ou recevoir des packets aux ou de la part des machines Workers.

```python
# Fonction pour envoyer et recevoir des messages
CodiumAI: Options | Test this function
def send_one_message(sock, data):
    length = len(data)
    sock.sendall(struct.pack('!I', length))
    sock.sendall(data)


CodiumAI: Options | Test this function
def recv_one_message(sock):
    lengthbuf = recvall(sock, 4)
    if not lengthbuf:
        return ''
    length, = struct.unpack('!I', lengthbuf)
    return recvall(sock, length)


CodiumAI: Options | Test this function
def recvall(sock, count):
    fragments = []
    while count:
        chunk = sock.recv(count)
        if not chunk:
            return None
        fragments.append(chunk)
        count -= len(chunk)
    arr = b''.join(fragments)
    return arr
```

- Les fonctions de démarrage des étapes de Map, de Shuffle, et de Reduce

Ces fonctions sont définies en amont du script pour faciliter la procédure d'envoie des ordres de début de map, shuffle et reduce du Master vers les Workers.

Les trois fonctions sont construites de la même manière. D'abord le master va ouvrir une socket pour établir la connexion avec un des Workers pour lui demander de débuter la phase correspondante et éventuellement lui envoyer les fichiers dont il a besoin pour réaliser sa tâche (à noter que les messages envoyer sont toujours encodés grâce à la fonction encode() pour n'envoyer que des bits lisibles par la machines qui les reçoit, puis décodés lors de leur réception avec la fonction decode())

Si on prend l'exemple de al fonction map_start, après la connexion à la socket correspondante, le Master va envoyer l'ordre « map start » ainsi que le fichier split contenant les mots à mapper. Le fichier split étant un des fichiers issus de l'exécution du module split.py mentionné avant séparant un fichier initial en n parties égales. À la fin du mapping, le Master reçoit du Worker une réponse informant que l'action est terminée.

Les fonction shuffle_start() ainsi que reduce_start() sont construites de la même manière.

```python
def map_start(host, port, split):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        client_socket.connect((host, port))
        # Envoie map start à host
        send_one_message(client_socket, '------> map start <------'.encode())
        send_one_message(client_socket, split.encode())
        response = recv_one_message(client_socket).decode()

    except Exception as e:
        print(f"Cannot connect with {host}: {e}")
        traceback.print_exc()

    return response
```

```python
def shuffle_start(host, port):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        client_socket.connect((host, port))
        # Envoie shuffle start à host
        send_one_message(
            client_socket, '------> shuffle start <------'.encode())
        response = recv_one_message(client_socket).decode()

    except Exception as e:
        print(f"Cannot connect with {host}: {e}")
        traceback.print_exc()

    return response
```

```python
def reduce_start(host, port):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        client_socket.connect((host, port))
        # Envoie recduce start à host
        send_one_message(
            client_socket, '------> reduce start <------'.encode())
        response = recv_one_message(client_socket).decode()

    except Exception as e:
        print(f"Cannot connect with {host}: {e}")
        traceback.print_exc()

    return response
```

On a aussi la fonction bye() qui permet de clôturer le système. Celle-ci va simplement fermer la socket du server (machine et port) pour lequel elle est appelée.

```python
def bye(host, port):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        client_socket.connect((host, port))
        message = '------> Bye !! <------'
        send_one_message(client_socket, message.encode())
        client_socket.close()

    except Exception as e:
        print(f"Cannot connect with {host}: {e}")
        traceback.print_exc()
```

- Le main() du module client_vf.py

C'est ici que qu'il va y avoir le script du déroulement de l'exécution et les appels des fonctions. Globalement, il est constitué d'une succession de 8 étapes :

i. Initialisation des structures de données et des variables

ii. Première connexion avec les machines et envoie du fichier machines.txt

iii. Instruction qui ordonne à chaque machine de se connecter et dire bonjour aux autres machines

iv. Premier threading : lancement du Map sur toutes les machines

```python
def map_start(host, port, split):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        client_socket.connect((host, port))
        # Envoie map start à host
        send_one_message(client_socket, '------> map start <------'.encode())
        send_one_message(client_socket, split.encode())
        response = recv_one_message(client_socket).decode()

    except Exception as e:
        print(f"Cannot connect with {host}: {e}")
        traceback.print_exc()
```

```python
# Premier threading pour lancer en simultanée le map sur chaque host
threads1 = []
map_debut = time.time()
for server in servers:
    thread = threading.Thread(target=map_start, args=(
        server[0], server[1], splits[servers.index(server)]))
    thread.start()
    threads1.append(thread)

for thread in threads1:
    thread.join()
map_fin = time.time()
map_total = map_fin - map_debut
print(f"map operation finished ! -------> {map_total} seconds")
time.sleep(2)
```

v. Deuxième threading : lancement du Shuffle sur toutes les machines

```
# Deuxième threading pour lancer en simultanée le shuffle sur chaque host
threads2 = []
shuffle_debut = time.time()
for server in servers:
    thread = threading.Thread(
        target=shuffle_start, args=(server[0], server[1]))
    thread.start()
    threads2.append(thread)

for thread in threads2:
    thread.join()
shuffle_fin = time.time()
shuffle_total = shuffle_fin - shuffle_debut
print(f"shuffle operation finished ! -------> {shuffle_total} seconds")
time.sleep(2)
```

vi. Troisième threading : lancement du Reduce sur toutes les machines

```
# Troisième threading pour lancer en simultanée le reduce sur chaque host
threads3 = []
reduce_debut = time.time()
for server in servers:
    thread = threading.Thread(
        target=reduce_start, args=(server[0], server[1]))
    thread.start()
    threads3.append(thread)

for thread in threads3:
    thread.join()
reduce_fin = time.time()
reduce_total = reduce_fin - reduce_debut
print(f"reduce operation finished ! -------> {reduce_total} seconds")
time.sleep(2)
```

vii. Récupération des résultats du MapReduce et des temps de calcul

viii. Quatrième et dernier threading qui clôt le système

SERVER_VF.PY

- Les fonctions d'envoi et de réception de messages (communes à client_vf.py)
- La fonction start_server()
- La fonction consistant_hash()

Cette fonction est importante car c'est elle qui va permette d'attribuer un mot à une machine unique et permettre ainsi le bon déroulement de la phase de shuffle. La fonction de hashage de base ne fonctionnait pas dans notre cas, il nous a fallu définir notre propre fonction de hashage :

```
def consistent_hash(word):
    return int(hashlib.md5(word.encode()).hexdigest(), 16)
```

- Initialisation des structures de données utilisées pour le MapReduce (listes et dictionnaires)

C'est ici que toutes les structures de données qui vont permettre d'enregistrer les mots à mapper ainsi que les résultats des shuffle et des reduces sont initialisée :

```
map_results = []
shuffle_results = []
reduce_results = {}
time_storage = {}
forbidden_signs = '.,!?":;()[]{}><\/@'
```

- La fonction handle_client()

Cette fonction est la fonction clé de pour la réussite du job de MapReduce car c'est ici que sont définies les actions à effectuer en fonction des ordres reçus du Master

Ici, sont inscrits des scripts voués à répondre à 10 messages différents captés via la socket.. Les 10 messages sont listés ci-après mais uniquement les étapes de map, shuffle et reduce seront explicités :

i. '------> hello !! <------'

ii. 'file'

iii. '------> go <------'

iv. '------> Tout le monde a répondu, super ! <------'

v. '------> map start <------'

Après avoir reçu le message map start, on rentre dans cette boucle qui stocke le deuxième message envoyé par le client qui contient soit le chemin d'accès encodé du split géré par le Worker, soit plusieurs chemins d'accès dans une liste encodée en format JSON :

```
# Reception du fichier a mapper
split = recv_one_message(client_socket).decode().strip()
# splits_json = recv_one_message(client_socket).decode().strip()
# splits = json.loads(splits_json)
```

Après avoir pris connaissance et lu du ou des splits attribués, le Worker stock un par un dans une nouvelle liste les mots qu'il va devoir shuffle :

```
# On lit déchiffre les bytes du split reçu et ignore les bytes qui ne sont pas valide en UTF-8 pour empêcher qu'ils ne créent d'erreurs
# for split in splits:
with open(split, 'r', encoding='utf-8', errors='ignore') as file:
    for line in file:
        words = line.split()
        for word in words:
            word = word.strip(forbidden_signs)
            word = word.lower()
            map_results.append(word)
```

Après avoir ajouté les mots de tous ses splits, le Worker envoie un message au Master pour lui dire qu'il a terminé.

vi. '------> shuffle start <------'

Le shuffle est la phase qui prend le plus de temps car elle va nécessiter que les Workers se connectent entre eux pour s'envoyer des mots.

D'abord le Worker qui a reçu l'ordre de commencer le shuffle va lire le machine.txt envoyé au début de la connexion pour prendre connaissance de ses collègues. Puis on utilise un dictionnaire de listes proportionnelles au nombre de Workers.

```
with open("machines.txt", 'r', encoding='utf-8') as f:
    hosts = [line.strip() for line in f.readlines()]

# Créer n listes de mots à envoyer à chaque host indépendemment
lists_to_send = {}
for host in hosts:
    lists_to_send[host] = []
```

Chaque liste associée à un Worker va contenir tous les mots gérés uniquement par ce Worker. On utilise la fonction de hashage personnalisée évoqueé avant pour déterminer le Worker associé au mot de la liste issue du mapping précédent.

```
for word in map_results:
    # On définit la machine qui va traiter le mot en utilisant une fonction de hashage sur ce mot
    selected_host = hosts[consistent_hash(
        word) % len(hosts)]

    lists_to_send[selected_host].append(word)
```

À la fin du processus, le Worker qui aura créé toutes les listes va se connecter à chaque autre Worker un par un pour lui envoyer sa liste.

```
for selected_host, words in lists_to_send.items():
    json_words = json.dumps(words)
    port = 6900
    server = socket.socket(
        socket.AF_INET, socket.SOCK_STREAM)
    server.connect((selected_host, port))

    message = 'operating shuffle'
    send_one_message(server, message.encode())
    send_one_message(server, json_words.encode())

    # Fermeture de la connexion avec la machine sélectionnée
    server.close()
```

Comme le Worker va se connecter à d'autrres Worker, à chaque connexion il va envoyer un message précis qui est 'operating shuffle' pour le prévenir qu'il a une liste à lui envoyer pour qu'il se prépare à la recevoir.

vii. 'operating shuffle'

Lors de la réception de ce message, le Worker qui doit recevoir la liste, la reçoit, la décode et la retire du format JSON avant de la lire et d'ajouter mot par mot les mots de cette liste au dictionnaire qui viendra compter les occurrences de ces mots dans l'étape d'après.

```
elif message == 'operating shuffle':
    words_json = recv_one_message(client_socket).decode().strip()
    words = json.loads(words_json)
    for word in words:
        shuffle_results.append(word)
```

viii. '------> reduce start <------'

De la même manière que dans la première partie avec la programmation séquentielle, le worker compte les occurrences des mots dont il dispose et envoie un message au Master dès qu'il a fini cette tâche :

```
elif message == '------> reduce start <------':
    # Chronométrage du reduce
    total_reduce_time = 0
    start_time_reduce = time.time()
    for word in shuffle_results:
        if word in reduce_results:
            reduce_results[word] += 1
        else:
            reduce_results[word] = 1
```

```
message = f"reduce operation is done in {total_reduce_time} seconds"
send_one_message(client_socket, message.encode())
```

ix. '------> envoie moi les résultats <------'

x. '------> bye !! <------'

4) Procédure d'exécution

Étape 1 : Nombre de machines

La première étape consiste à définir le nombre de machines avec lesquelles nous souhaitons travailler. Soit l'on connaît le nom des machines de TP et nous les rentrons directement dans le fichier texte machines.txt, soit nous voulons exécuter la procédure automatique qui check l'état de toutes les machines de TP disponible puis qui sélectionne de manière aléatoire un nombre n de machines définit à l'exécution. La deuxième méthode est menée en exécutant la commande « bash getMachines.sh n (nombre de machines) »

Étape 2 : Choix du fichier de travail et splitting de celui-ci

- Une fois les machines de travail définie, on détermine le fichier avec lequel on veut faire fonctionner le système distribué (on regarde plus précisément la taille de fichier qui nous intéresse).

Une fois le fichier récupéré localement, nous allons le séparer en plusieurs morceaux de tailles égales afin de les envoyer séparément aux machines correspondantes.

Ici, je mentionne le split en local du fichier mais on peut récupérer des fichiers directement dans le commoncrawl que nous pouvons considérer comme des splits en tant que tel pour nous éviter de télécharger de gros volumes de données. Cette gestion des splits directement sur le dossier common/crawl n'est pas évoquée bien que je l'aie implémentée dans le module client_splits_commmoncrawl.py.

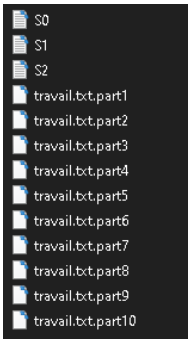
- Une fois le fichier sélectionné et mis dans le dossier « adeployer », on actualise manuellement les chemins d'accès à ce fichier dans les deux commandes du module split.py afin qu'il puisse réaliser le split sans erreur puis supprimer le fichier initial afin qu'il ne soit pas déployer.

```
splits = split_file(
    "/home/matsvr/projet_systrep/adeployer/splits/grosfichier6.warc.wet.part1", n)
# splits = split_file(
#     "/home/matsvr/projet_systrep/adeployer/splits/travail.txt", n)

print(f"Fichier séparé en {n} parties")

# Supprime le fichier original
os.remove("/home/matsvr/projet_systrep/adeployer/splits/grosfichier6.warc.wet.part1")
# os.remove("/home/matsvr/projet_systrep/adeployer/splits/travail.txt")
```

- On exécute le module split.py et on observe que les splits à déployer sont bien apparus :



Le fichier travail.txt a ici été split.

Étape 3 : Actualisation du code

Un fois le fichier split, nous avons besoin d'actualiser le les noms des splits à envoyer à chaque Worker dans le script du module client_vf.py déployé sur le Master.

On modifie donc manuellement le chemin d'accès aux splits stockés dans la liste d'initialisation comme montré dans l'exemple ci-dessous pris pour les splits de travail.txt.

```
def main():
    # Open the hosts file and read the hosts into a list
    with open('machines.txt', 'r') as f:
        # replace with the server IP address or hostname
        hosts = [line.strip() for line in f.readlines()]
        splits = []
        for i in range(len(hosts)):

            splits.append(
                f"/tmp/msauveur-23/adeployer/splits/travail.txt.part{i+1}")

        port = 6900
        servers = []
```

Lorsque le master va commander le début du mapping, il enverra chaque élément de la liste splits au Worker associé.

Étape 4 : Déploiement général

À ce stade, tous les fichiers du dossier « adeployer » sont prêt à être envoyés aux Workers. On va donc exécuter «bash deploy.sh ». Comme vu précédemment, cela va tuer les processus en cours, créer en écrasant un dossier temporaire avec « adeployer » dans chaque machine (composé du module server_vf.py et des splits) et lancer automatiquement le server_vf.py qui va faire rentrer les Workers dans une phase d'attentes d'ordres de la part du Master.

Étape 5 : Lancement du scénario par le Master

Une fois que les fichiers sont déployés sur toutes les machines et que tous le monde écoute, nous allons lancer client_vf.py sur la machine Master ce qui va d'abord envoyer le fichier machines.txt à toutes les machines afin que chacune d'elles prenne connaissance de l'identité de tous les Workers sur le job.

Ensuite une séquence se lance durant laquelle chaque machine ira une par une, dire bonjour à toutes les autres machines pour s'assurer qu'elles puissent bien communiquer entre elles, action nécessaire pour l'étape du shuffle dans le MapReduce.

Enfin, l'opération MapReduce va se lancer en envoyant en simultanée (à l'aide d'un threading) à toutes les machines les ordres de début de map, de shuffle et de reduce. À noter que le shuffle démarre uniquement si toutes les machines ont fini leur map, et le reduce démarre seulement si tous le monde a fini son shuffle.

Étape 6 : Affichage des résultats

À la fin de l'exécution, toutes les machines ont pour ordre d'envoyer leurs résultats pour les grouper, mais aussi leur temps de calcul pour chaque phase effectuée (map, shuffle, reduce).

Le master calcul de son côté le temps global pris pour ces phases en calculant le temps entre l'ouverture du thread envoyant les ordres de démarrage aux Workers et la fermeture du thread en captant la réponse du dernier Worker à avoir fini sa tâche.

En conséquence, nous avons dans le terminal 4 zones :

- Une première dans laquelle tous les Workers confirment d'avoir bien reçu le fichier machine.txt

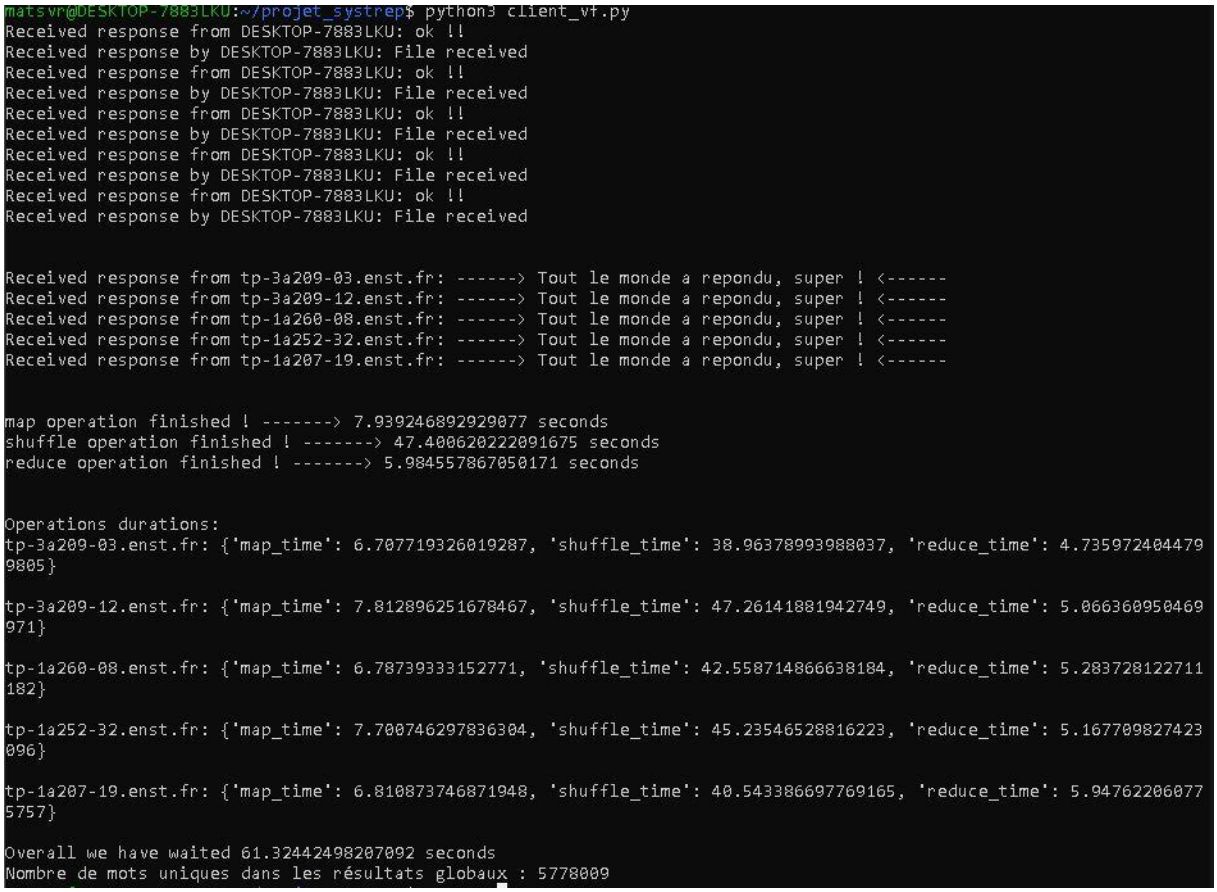
- Une seconde dans laquelle chaque Worker confirme s'être bien connecté à tous les autres et que tous le monde a confirmé cette connexion

- Une troisième dans laquelle on est informé de la fin de la phase de map, de reduce et de shuffle avec les temps associés calculés par le Master

- Une quatrième dans laquelle chaque Worker donne le temps propre pris pour chaque phase.

À la fin, le temps global attendu pour le MapReduce est affiché (somme du temps des 3 phases). Le nombre de mots uniques est relevé également en prenant la longueur de la liste résultat créée (et qui n'est pas affichée car trop longue pour des gros fichiers)

Un visuel obtenu est présenté en exemple ci-après

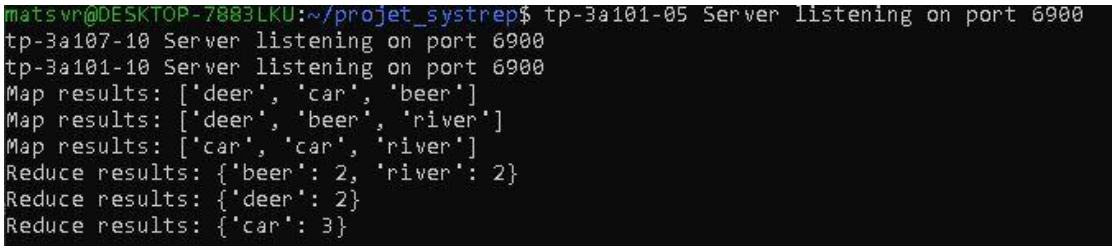


Exemple pour 5 Workers avec un fichier de 780Mo.

5) Premiers résultats

On utilise une première fois le système distribué pour faire un MapReduce sur les fichiers de base test, notamment avec les splits S0, S1 et S2.

Le résultat obtenu est le suivant :



Sur ce premier terminal, on affiche les résultats du mapping des 3 Workers puis leur résultat de l'opération de reduce. On voit bien que chaque mot a été attribué à une machine précise, envoyé à cette dernière puis compté dans un dictionnaire.

```
Received response from DESKTOP-7883LKU: ok !!
Received response by DESKTOP-7883LKU: File received
Received response from DESKTOP-7883LKU: ok !!
Received response by DESKTOP-7883LKU: File received
Received response from DESKTOP-7883LKU: ok !!
Received response by DESKTOP-7883LKU: File received


Received response from tp-3a101-05.enst.fr: ------> Tout le monde a repondu, super ! <------
Received response from tp-3a107-10.enst.fr: ------> Tout le monde a repondu, super ! <------
Received response from tp-3a101-10.enst.fr: ------> Tout le monde a repondu, super ! <------


map operation finished ! -------> 0.046527862548828125 seconds
shuffle operation finished ! -------> 0.018739700317382812 seconds
reduce operation finished ! -------> 0.018009185791015625 seconds


Operations durations:
tp-3a101-05.enst.fr: {'map_time': 0.0002543926239013672, 'shuffle_time': 0.003827333450317383, 'reduce_time': 7.79628
7536621094e-05}

tp-3a107-10.enst.fr: {'map_time': 0.0002753734588623047, 'shuffle_time': 0.004316568374633789, 'reduce_time': 7.31945
0378417969e-05}

tp-3a101-10.enst.fr: {'map_time': 0.00026869773864746094, 'shuffle_time': 0.001682281494140625, 'reduce_time': 6.8664
55078125e-05}

Overall we have waited 0.08327674865722656 seconds
Nombre de mots uniques dans les résultats globaux : 4
```

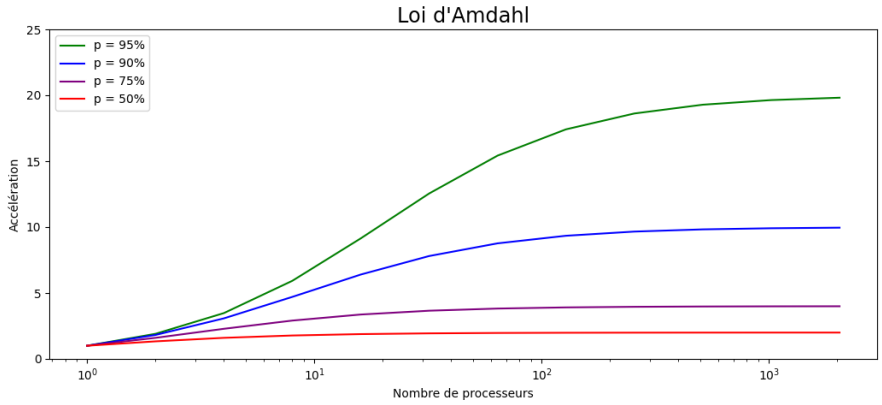
Ce second terminal nous donne bien les 4 zones évoquée en détaillant les temps d'exécution individuels.

6) Démonstration empirique de la loi d'Amdahl

1. Rappel sur la loi d'Amdahl

La loi d'Amdahl définit une limite théorique à l'amélioration de performance qu'on peut obtenir en améliorant une partie d'un système informatique. Elle est souvent utilisée pour comprendre les gains potentiels de performance d'un système en augmentant le nombre de processeurs en parallèle.

La loi d'Amdahl s'exprime généralement par la formule suivante : 1/((1-p)+(p/n), où p est la proportion du programme qui peut être parallélisée et n le nombre de processeurs ajoutés.



Les principaux objectifs de la loi d'Amdhal est

- D'évaluer les gains de parallélisation vu qu'elle aide à comprendre jusqu'à quel point l'ajout de processeurs supplémentaires va améliorer les performances d'un programme

- Identifier les limites en montrant que l'amélioration des performances a une limite théorique, surtout lorsque la majorité du programme ne peut être parallélisée.

- Optimisation des ressources car permet de savoir si l'investissement dans du matériel supplémentaire (comme des processeurs supplémentaires) est judicieux ou non.

2. Résultats

Afin de faire une démonstration empirique de cette loi d'Amdhal, j'ai choisi d'exécuter l'opération MapReduce en distribué sur différentes tailles de fichiers et avec différents nombres de Workers afin de faire varier le nombre de cœurs utilisé pour effectuer la tâche.

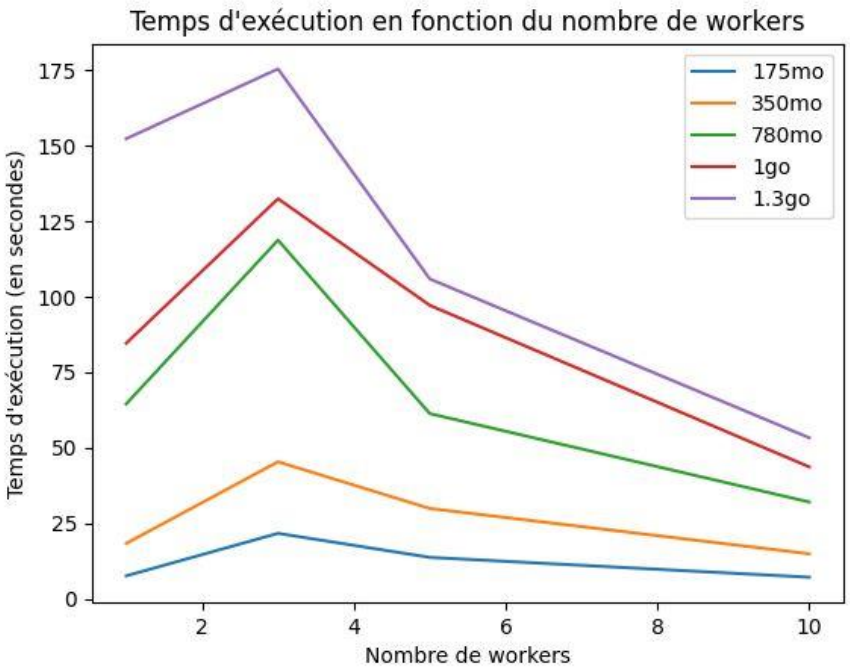
J'ai choisi de faire l'étude sur 5 fichiers respectivement de taille **175Mo**, **350Mo**, **780Mo**, **1Go** et **1.3Go** pour des Workers au nombre de 1, 3, 5 et 10.

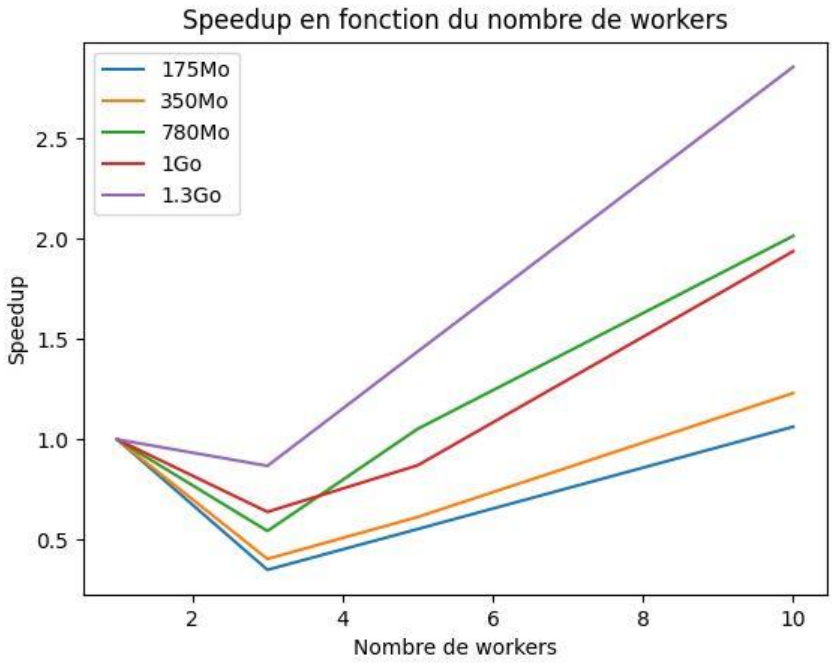
Les temps d'exécutions en fonction des tailles de fichiers et du nombre de Workers sont résumés dans le tableau ci-dessous. Les screenshots des résultats sont stockés dans le dossier « results_screenshots_Amdahl ».

	175Mo	**350Mo**	**780Mo**	**1Go**	**1.3Go**
1 Worker	7.57s	18.32s	64.54s	84.64s	152.39s
3 Workers	21.65	45.36s	118.80	132.52s	175.53s
5 Workers	13.69s	29.89s	61.32s	97.19s	105.93
10 Workers	7.12s	14.88s	32.03s	43.66s	54.31s

Pour les exécutions du script à 1 Worker, j'ai réalisé le comptage des occurrences des mots à l'aide du programme sequentiel.py utilisé dans la première partie. Je n'ai pas pris en compte le temps pris pour faire le tri par ordre alphabétique car je ne le fais pas dans le MapReduce en distribué.

Affichage graphique :





Observations :

Les différentes observations que nous pouvons faire sur nos résultats sont les suivantes :

- Tout d'abord on remarque qu'une exécution séquentielle est généralement plus avantageuse en termes de temps de calcul et de ressources lorsque le nombre de Workers est faible. En effet, la répartition d'une tâche peut se voir être trop demandante en ressources par rapport à la taille du fichier traiter et à la faible complexité du job.

  Le pic témoignant d'une moins bonne performance avec un nombre de workers très faible est retrouvée pour toutes les tailles de fichiers. On peut en déduire que l'utilisation d'un système distribué n'est donc pertinente qu'à partir d'un certain de Workers impliqués dans l'opération.
  En revanche, le pic s'atténue à mesure que la taille du fichier augmente, on peut donc naturellement supposer que pour de très gros volumes de données, une répartition du job même entre 2 ou 3 workers pourrait donner des résultats très satisfaisant. Notre étude est donc assez limitée par la taille des fichiers traités.

- Une autre observation que nous pouvons faire est que pour des fichiers de tailles assez voire très petites, il n'est pas pertinent de distribuer la tâche. Pour le fichier de 175Mo, on obtient un speed up de quasiment 1 avec 10 workers, soit un résultat équivalent au résultat séquentiel. Encore une fois, avec un nombre beaucoup plus élevé de worker, on peut surement obtenir un meilleur speed up, cependant trop couteux pour la tâche à réaliser.

Globalement, on observe une augmentation significative du speed up de l'exécution du job en fonction du nombre croissant de Workers. L'augmentation du speed up est d'autant plus flagrante qu'on augmente la taille du fichier traité.

Au vu de l'allure des courbes et au fait qu'on ne distingue pas encore de plateau, on peut supposer que le speed up peut encore beaucoup augmenter si on ajoute des Workers. En effet, on sait qu'avec une bonne parallélisation du code, on peut atteindre des speeds up allant jusqu'à du x20.

Cependant, limités par les ressources, je n'ai pas pu explorer la limite de la loi d'Amdahl en tentant d'atteindre ces plateaux de speed up pour lesquels le nombre de Workers pris pour une exécution

n'a plus d'impact sur la rapidité de cette dernière. Les résultats ci-dessus nous permettent juste de faire les premières constatations des avantages et des limites des systèmes réparti et de la parallélisation des tâches.

7) Limites atteintes, problèmes rencontrées et perspectives d'amélioration de l'architecture du code

Comme évoqué juste avant, ayant travaillé localement, j'ai eu du mal à travailler sur des fichiers de plus de 1.3 Go. J'ai entamé la procédure permettant d'accéder au dossier commoncrawl des machines de TP afin de récupérer des splits directement à l'intérieur, mais par souci de temps et des crashs de connexions à cause d'une trop grande quantité de fichiers lourds, je me suis limité à l'étude présentée.

La perspective d'évolution est le travail sur des fichiers encore plus gros et en prenant encore plus de machines. Il aurait également été intéressant de récupérer plus de points afin de mieux discrétiser les courbes et distinguer des tendances.

Par souci de temps je n'ai pas pu traiter la toute dernière question sur l'élaboration d'une architecture résiliente aux pannes avec une redistribution des ressources et des splits en cas d'échecs de connexions (comme le fait très bien le framework Hadoop dans le running de ses clusters).