Projet INF727 – Systèmes Répartis

Sara Boutigny

2021-2022

Code source

https://github.com/SaraBoutigny/MS-Big-Data-2021-2022/tree/main/INF727%20-%20Syst%C3%A8mes%20R%C3%A9partis/MapReduce

Structure du code optimisé

Dans les étapes qui vont suivre, on considère la première machine distante comme le master. Celle-ci exécute les principales étapes du programme. Les passages en surbrillance sont les plus importants : ils constituent une amélioration par rapport aux 13 étapes constitutives du projet.

Etape 0 : Clean

Effacement de tous les dossiers /tmp/<login> sur les machines utilisées.

Etape 1 : Deploy

- Création du dossier dans /tmp et des sous-dossiers sur chaque machine. (P)[1]
- Envoi des scripts python sur chaque machine. (P)

Notons que chaque commande identique effectuée sur plusieurs machine est parallélisée, mais que le programme attend ensuite la fin d'exécution avant d'en relancer d'autres.

Etape 2 : Split

- Le master créé les splits et les déploie simultanément sur chaque machine. (P)
- Le master attend la fin du déploiement sur toutes les machines avant de passer à la suite. Pour cela, on lance les commandes ssh/scp successivement grâce à la méthode Popen de subprocess et seulement ensuite, on applique la méthode communicate aux process en cours d'éxécution.

Etape 3 : Map

- Le master organise l'exécution simultanée du script slave.py sur chacune des machine afin de créer les map. Ces map sont au format .txt. (P)
- La master attend la fin d'exécution de tous les slaves (de façon similaire à l'étape précédente)

Etape 4 : Shuffle

- Le master organise l'exécution simultanée du script slave.py sur chacune des machines (P) afin de créer les shuffle. Le fait de créer un fichier par hashcode était gourmand en mémoire, on décide donc de créer autant de fichiers que de machines. Cela réduit drastiquement la mémoire utilisée et le temps d'exécution. On utilise le hashcode seulement pour déterminer

[1] (P) : action parallélisée sur les différentes machines

quel clé est envoyée vers quelle machine. Les clés sont stockées dans des dataframes pour faciliter le traitement puis enregistrées au format pickle. Chaque DataFrame correspond à un fichier pickle et se réfère à une seule machine, sa machine de destination.

- On note que cette étape est parmi les plus chronophages du map reduce : plus le cluster est grand, plus il y a de fichiers à créer et à envoyer aux machines. On bride alors le nombre de fichiers shuffle à 4 machines maximum. Le reste des étapes est donc parallélisé sur maximum 4 machines.
- Chaque fichier pickle est envoyé à sa machine de destination. (P)

Etape 5 : Reduce

- Le master organise l'exécution simultanée du script slave.py sur toutes les machines afin de créer les fichiers reduce (P). La tâche est optimisée puisque l'on transforme les fichiers pickle en DataFrame, conservant ainsi leur format d'origine et nous donnant accès aux méthodes de la classe DataFrame. En un simple groupby, on obtient alors le wordcount partiel pour la machine concernée, sous forme d'un DataFrame, puis enregistré au format pickle.
- Le master attend la fin de l'exécution de l'étape précédente puis envoie les wordcounts partiels à la machine 0 (P), qui sera chargée du wordcount final.

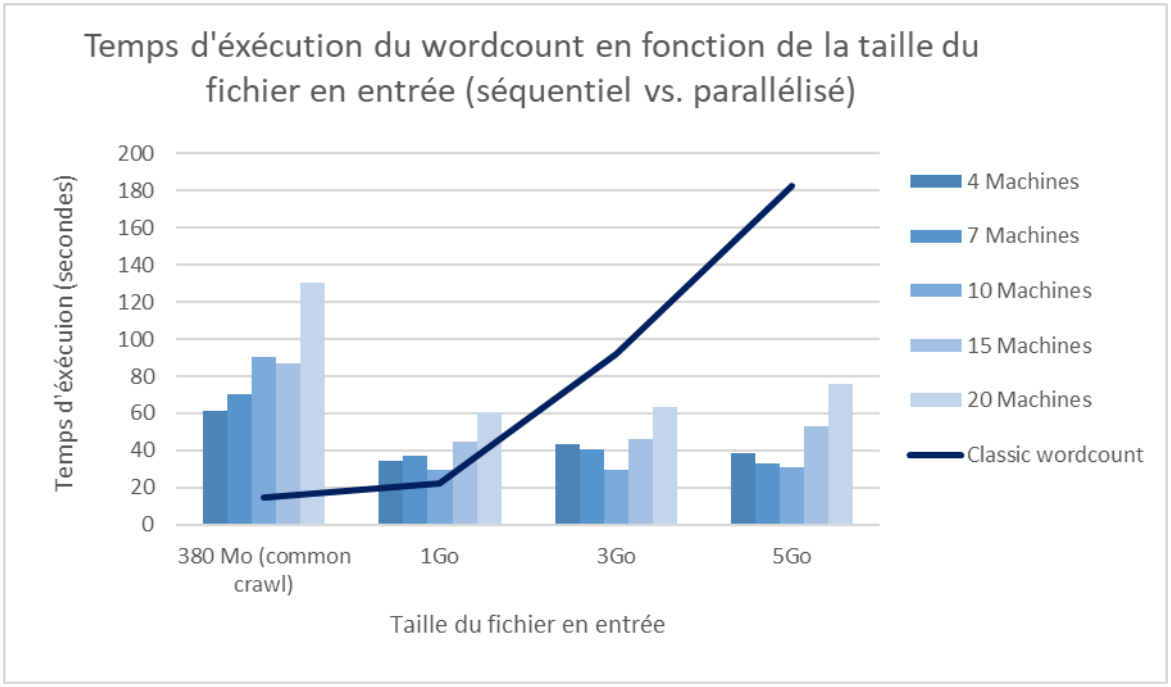
Etape 6 : WordCount final

- La machine 0 recueille tous les wordcount partiels et exécute le wordcount final à partir de tous les fichiers pickle reçus. Une fois de plus, l'objet DataFrame et ses méthodes aident à obtenir le résultat final en un temps optimal. Cette étape n'est pas parallélisée puisque s'effectue sur une seule machine.
- Le résultat du wordcount est au format .txt dans le dossier « wordcount » de la machine 0.

Temps d'exécution, taille du fichier input, et nombre de machines

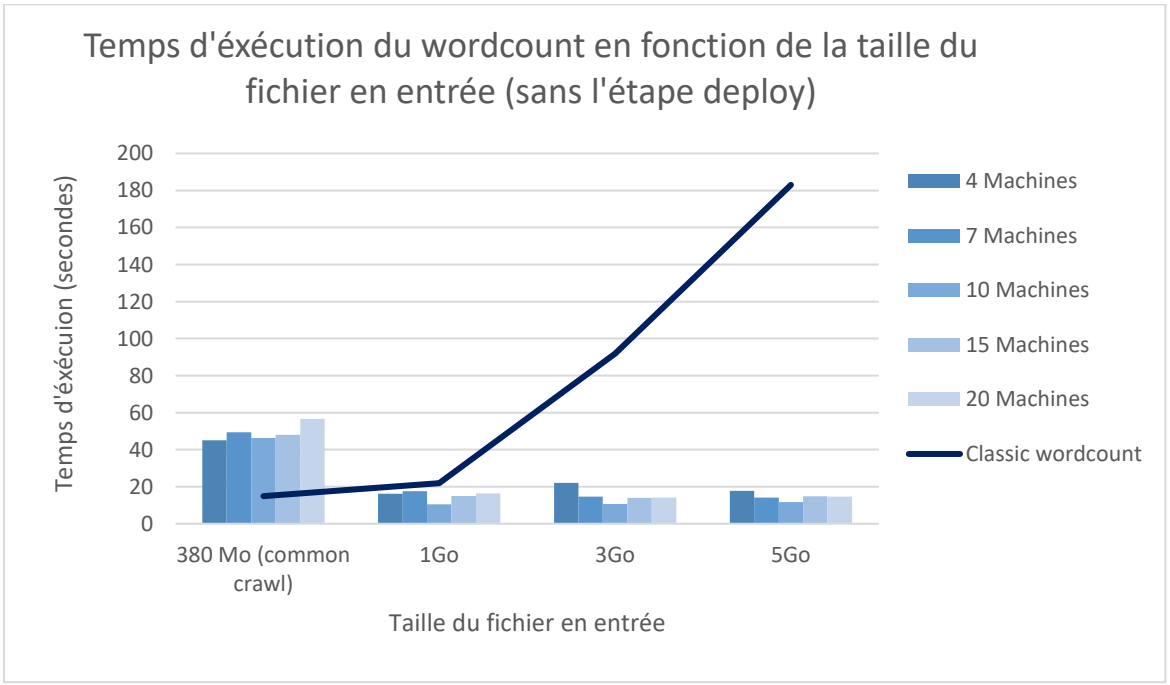
Pour cette étape, on utilise des fichiers .txt de différentes tailles, allant de 380Mo à 6Go. Le fichier de 380 est celui du Common Crawl. Les autres fichiers ont été générés à partir d'un script python et contiennent des mots aléatoires du dictionnaire anglo-saxon.

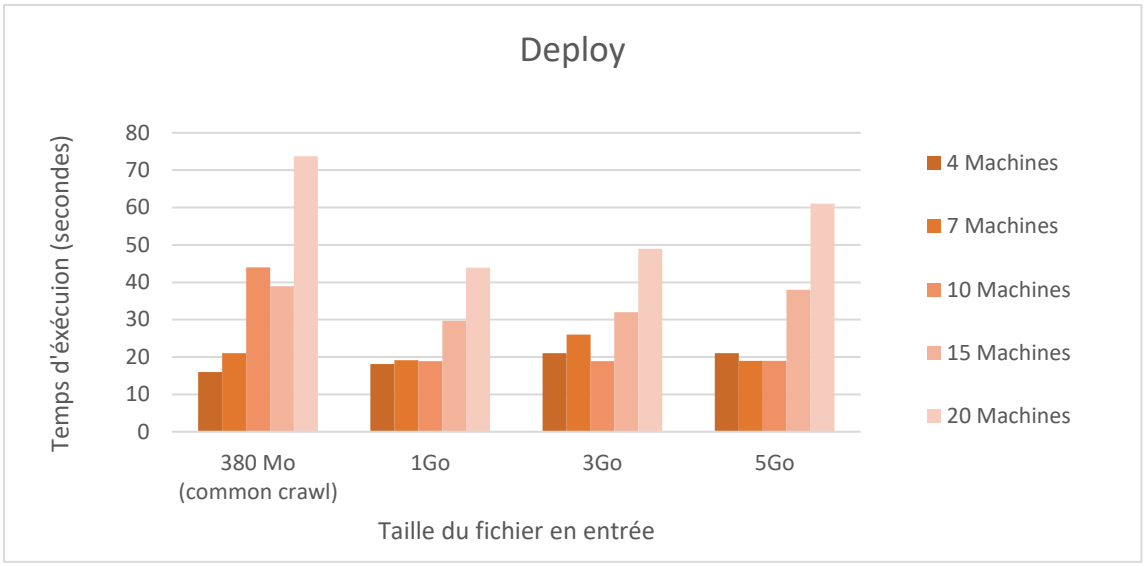
Résultats

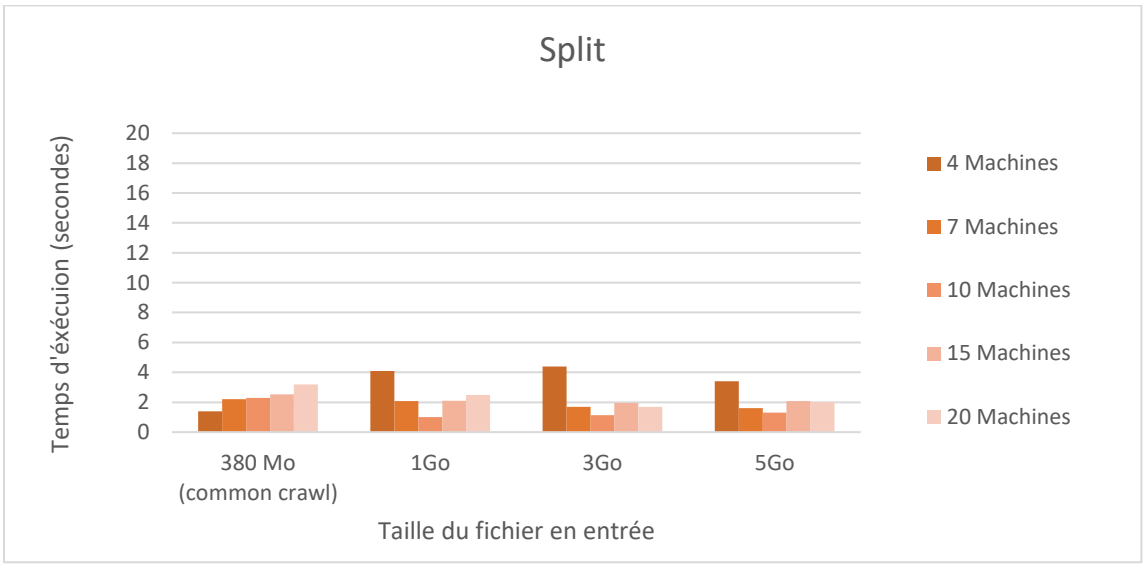


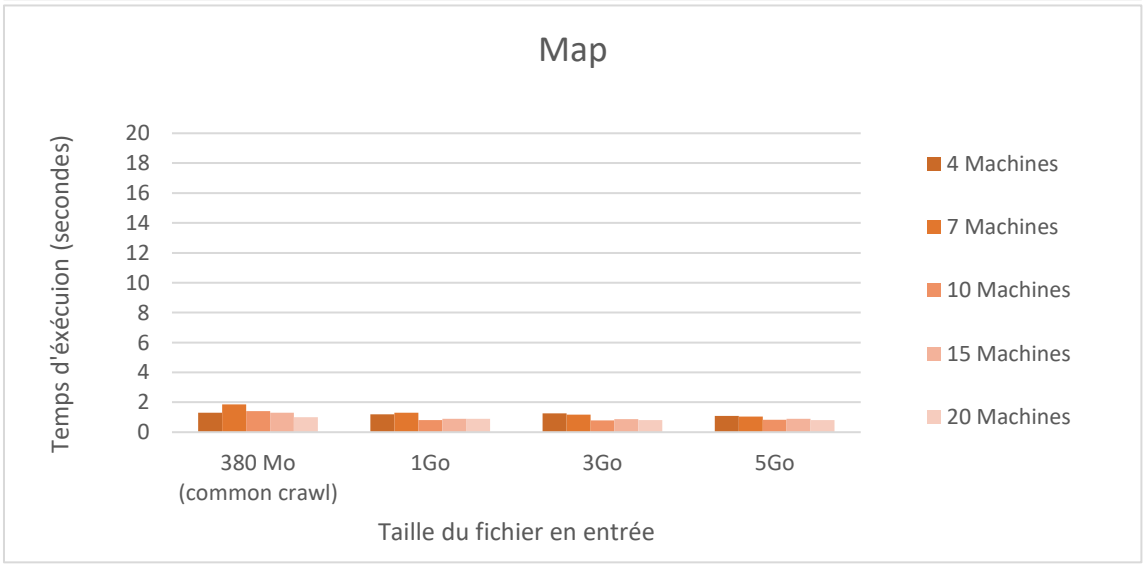
On peut voir que lorsque le fichier en entrée est de taille supérieure à environ 1Go, le wordcount parallélisé obtient de meilleures performances que le wordcount séquentiel.

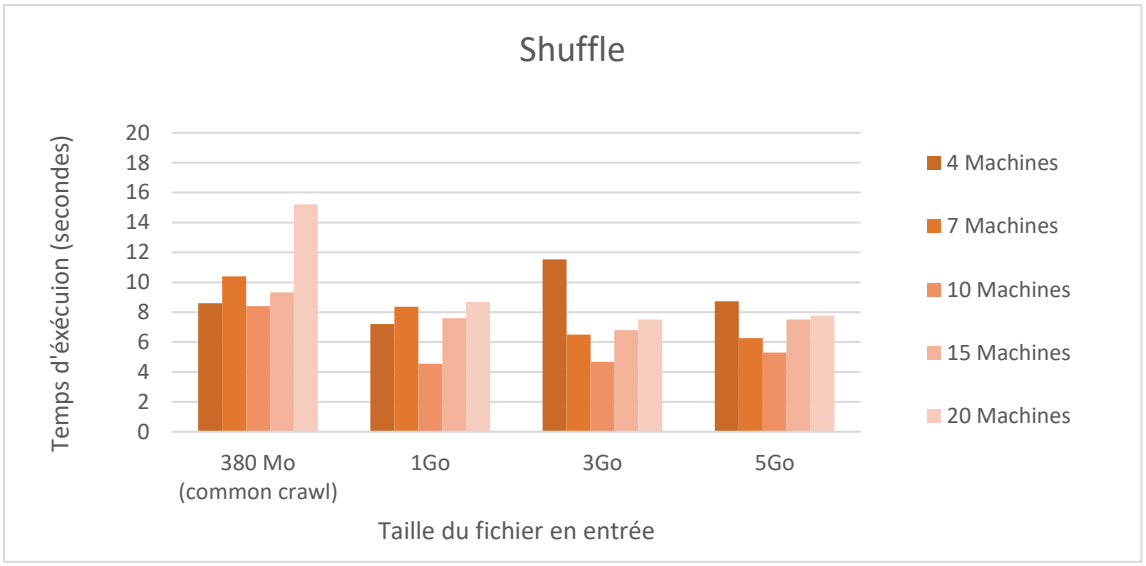
De plus, l'étape qui prend le plus de temps est l'étape *deploy.* En effet, les envois de fichier du poste personnel sur une machine distante sont chronophages.

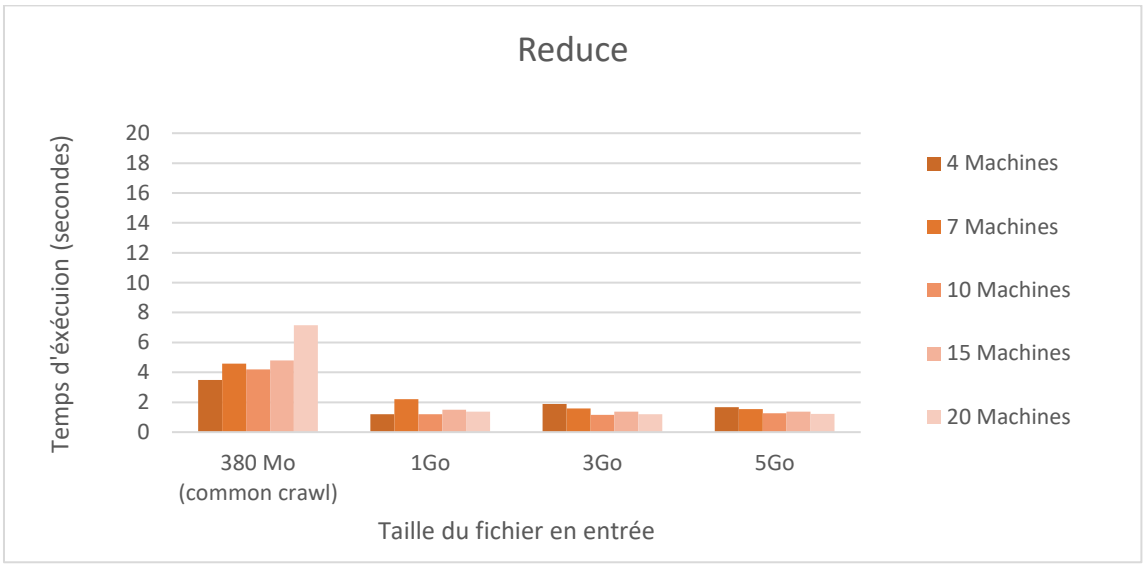


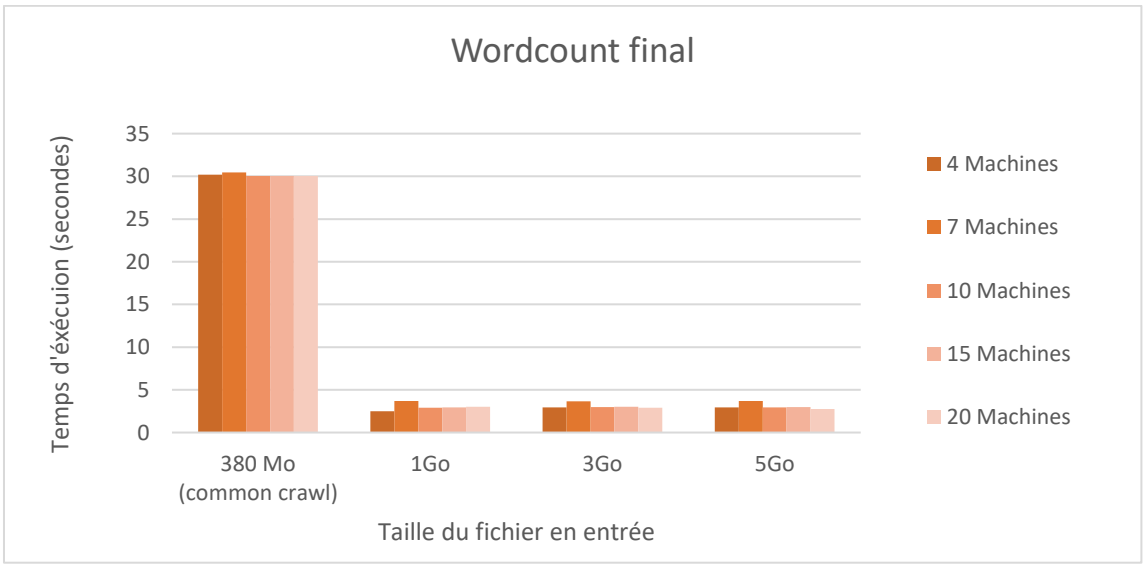

Deploy
Temps d'éxécuion (secondes)
80
70
60
50
40
30
20
10
0
380 Mo (common crawl)
1Go
3Go
5Go
Taille du fichier en entrée
4 Machines
7 Machines
10 Machines
15 Machines
20 Machines


Split
Temps d'éxécuion (secondes)
20
18
16
14
12
10
8
6
4
2
0
380 Mo (common crawl)
1Go
3Go
5Go
Taille du fichier en entrée
4 Machines
7 Machines
10 Machines
15 Machines
20 Machines


Map
Temps d'éxécuion (secondes)
20
18
16
14
12
10
8
6
4
2
0
380 Mo (common crawl)
1Go
3Go
5Go
Taille du fichier en entrée
4 Machines
7 Machines
10 Machines
15 Machines
20 Machines







En ce qui concerne le détail des étapes, on remarque que certaines d'entre elles sont plus indépendantes que d'autres de la taille du fichier et du nombre de machines. Par exemple, quelle que soit la taille du fichier ou le nombre de machines, les étapes *split*, *map*, *reduce*, et *wordcount final*

restent sensiblement les mêmes, bien que les graphiques montrent qu'augmenter le nombre de machines réduit très légèrement le temps d'exécution. A l'inverse, les étapes *deploy* et *shuffle* sont très sensibles au nombre de machines. Plus le nombre de machines est élevé, plus les étapes de *shuffle* et de deploy prennent de temps. L'étape *deploy* montre plus de linéarité dans son augmentation face au nombre de machines. Tandis que l'étape *shuffle* semble montrer qu'il existe un nombre optimal de machine (non situé aux extrêmités) qui permet de minimiser le temps d'exécution. En effet, avec peu de machine, on a peu de fichiers shuffle mais on traite davantage de données par fichier. Et lorsque le nombre de machines augmente, on a trop de fichiers shuffle.

Pistes d'amélioration

- Améliorer l'étape *deploy* de sorte à gagner du temps sur le déploiement des fichiers nécessaires au fonctionnement du cluster. Dans notre cas, le problème réside dans le temps de latence dans l'envoi de fichiers depuis la machine personnelle vers la machine distante. L'avantage d'utiliser la machine personnelle était de piloter plus facilement le déroulement du code. Il serait avantageux de travailler directement depuis la machine distante et d'y initier tous les envois.
- Le programme parallélisé réalise de moins bonnes performances sur le fichier common crawl. Il peut être intéressant de comprendre pourquoi certains fichiers sont mieux traités que d'autres et faire en sorte que le programme tienne compte de ces disparités : les lignes sont-elles plus longues ? Y a-t-il plus de caractères spéciaux ?
- L'étape *shuffle*, déjà améliorée, contient autant de fichiers que de machines. Augmenter le nombre de machines augmente le temps de *shuffling* et donc le temps d'exécution. Il peut être intéressant de diminuer le nombre de machines recevant les *shuffle*, pour limiter les envois de fichiers et l'allocation de mémoire dans la création de multiples fichiers