Projet : Hadoop optimisation

Chaimaa Lotfi

Département HPC-BigData - Deuxième année
Mai 2021

Table des matières

1 Introduction

Notre plateforme comporte un système de fichiers adapté au traitement concurrent de masses de données, ainsi qu'un noyau d'exécution axé sur l'ordonnancement et la gestion de tâches selon le schéma (map-reduce). L'architecture et les fonctionnalités de cette plateforme reprendront celles de la plateforme Hadoop. Nous avons consolider notre Hidoop dans la version précédente, mais nous avons continue nos tests sur des fichiers plus volumineux (car nous avions des problèmes dans X2Go, mais on a résolu tous ces problèmes.).

2 Une version améliorée de Hidoop :

2.1 Changement d'architecture

Voici notre nouvelle architecture pour l'amélioration de hidoop :

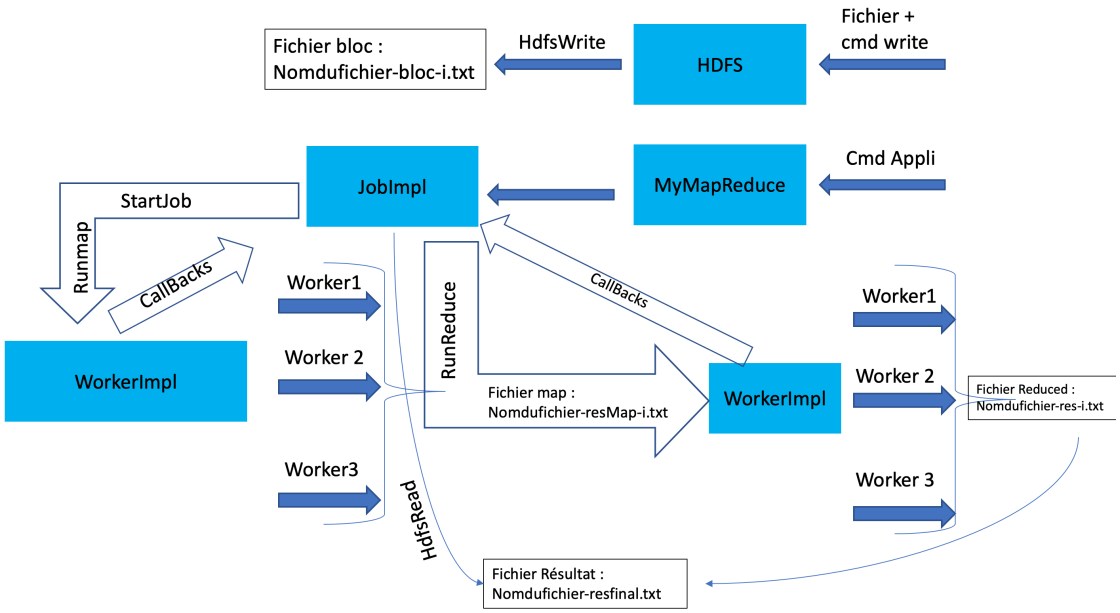

FIGURE 1 – Architecture de notre application

2.2 Modification des algorithmes ou protocoles existants

1. **Maps Parallèles :** les maps sur les blocs se font maintenant en parallèle grâce à des threads lancés par la classe WorkerImpl.java.
2. **Reduces Parallèles :** nous effectuons maintenant plusieurs reduces en parallèle. Le problème qui s'est posé alors était qu'il était obligatoire de

faire un reduce final qui va réduire les autres résultats des autres reducers. En effet, avec la configuration initiale des maps, un mot peut figurer dans plusieurs résultats de maps. Afin de remédier à cela, nous avons choisi d'attribuer à chaque fichier résultat de map une partie de l'alphabet correspondant à la première lettre des KV qui s'y trouvent. Le fichier nomdufichier-resMap1.txt contiendra les KV commençant par un A, nomdufichier-resMap2.txt par un B, nomdufichier-resMap3.txt par un C... etc.
Ainsi nous pouvons lancer plusieurs reduces avec la classe WorkerImpl en parallèle sur les fichiers resMap et concatener les résulats pour obtenir le résultat final.

3. **Classe MapFormat :** La classe MapFormat nous permet de construire la liste des fichiers nomdufichier-resMapi.txt afin de choisir celui dans lequel nous allons mettre la KV selon sa première lettre.
4. **CallBack :** comme les reduces se font dorénavant en parallèle, il fallait revoir comment on acceptait les callbacks et faire un simple lock n'a pas suffi. Nous avons donc ajouté une classe CallBackImpl qui instancie un sémaphore. Ce sémaphore est libéré quand un worker a terminé son travail (map ou reduce) et est pris dans la boucle d'attente de terminaison des workers.

3 Outils de déploiement, test et l'évaluation des performances

Pour le déploiement nous avons choisi d'écrire des scripts pour lancer et arrêter notre script ainsi que pour la suppression des fichiers. Pour le lancement on change le directory ou on travail et on modifie aussi le nombre de worker en les ajoutant en utilisant des boucles on a pu automatiser les lancements des workers et des serveurs dans le fichier lanceur.sh.
Ce fichier nous a permis de

— déployer votre plateforme avec une simple ligne de commande .

— configurer notre plateforme (en ajoutant le nom des machines).

— Lancer notre platforme en toute simplicité .

La commande pour le lancement de notre plateforme après modification selon le besoin des paramètres est : ./lanceur.sh

Pour l'étude de performance nous avons eu l'idée d'écrire un script qui duplique notre fichier et en faisant de simple calcul on peut savoir le nombre de répétition qu'il faut pour travailler sur de gros fichier. c'est une simple boucle en changer le nombre d'itération on obtient le fichier souhaité dans /home/mwissad/nosave/test.txt . Le fichier lanceur et aussi un plus pour l'étude de performance car il permet d'ajouter et de supprimer des workers.

La commande pour dupliquer les fichiers est : ./dupliquer.sh

Après le lancement avec le lanceur nous avons aussi utiliser un script d'arrêt pour arrêter est un script pour supprimer les fichiers bloc qui ont été générer lors de la période d'activité de notre programme. **La commande pour dupliquer les fichiers est : ./arret.sh**

Pour les résultats nous avons rediriger la sortie dans le fichier nohup.out dans ce fichier on peut avoir le temps après la fin de notre programme ainsi les différentes infomations de l'échange entre le client et le serveur. **Pour avoir des informations sur les étapes de l'exécution de notre programme veuillez consulter le fichier nohup.out et pour encore plus de détails nous avons mis les remarques importantes dans le fichier ReadMe.**

4 Étude de performances

Nous avons suivi la méthode suivante pour l'évaluation des performances, nous voulons comparer le temps mis par l'application Hidoop pour les différentes taille de fichiers, et différents nombre de workers.

Tout d'abord une comparaison en terme de temps,en essayant avec les même paramètres, dans. le but d'obtenir un temps d'exécution plus petit avec un plus grand nombre de worker.

Comme expliquer précédemment nous avons utiliser le script duplicate.sh pour créer des fichier volumineux avec 100000 itérations a donné un fichier de 200Mo.pour l'agrandir un simple copier coller suffit. La duplication nous a permis aussi de connaitre la cohérence des résultats même avec des fichiers trés grand il suffisait de multiplier le résultat de notre fichier d'origine par le nombre de répétition avec le changement de notre architecture et c'était très utile pour connaitre la cohérence.

Nous avons comme dans la première itérations fixer la taille du fichier et on fait varier le nombre de worker en fixant les autres paramètres. On remarque que avec un petit fichier le nombre de worker importe peu à notre test de la plateforme en ajoute 5 worker avec un gros fichier on remarque que le temps diminue effectivement en laissant pensé que en ajoutant d'avantage de worker cela permettera de diminuer encore le temps. c'est faux puisque notre platforme stagne à un moment avec un nombre de worker une fois dépassé le rendement en terme ne bouge quasiment plus.

4.1 Pour les fichiers de petite taille :

Pour la nouvelle version d'Hidoop avec optimisation :

Nombre de workers	Taille de fichier (Ko)	Temps mis en (ms)
5	100	625
5	260	1026
5	650	1728
8	650	1925
3	1000	4000
8	1000	2498
8	2000	4061
6	1000	2650
5	10000	26928
6	10000	27720
8	10000	27383

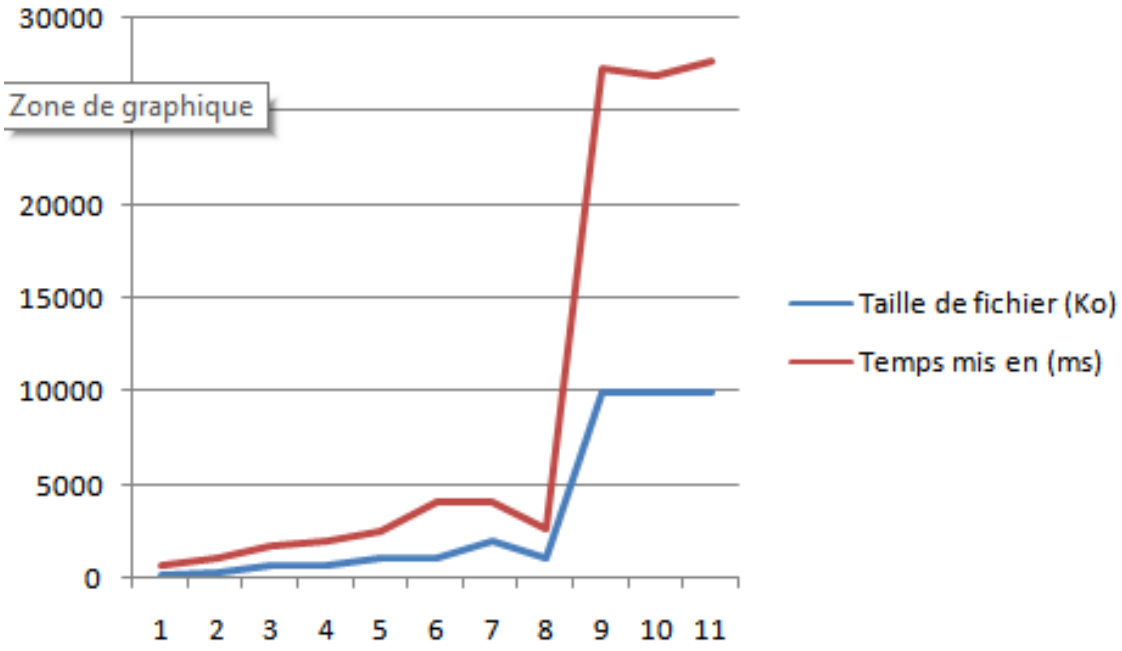

FIGURE 2 – Le temps mis en fonction de taille de fichier (new version of Hidoop)

Les résultat qu'on a eu avec l'ancienne version d'Hidoop :

Nombre de workers	Taille de fichier (Ko)	Temps mis en (ms)
3	1000	2044
6	1000	1979
1	10000	10422
3	10000	9539
6	10000	9052
5	100000	1000000

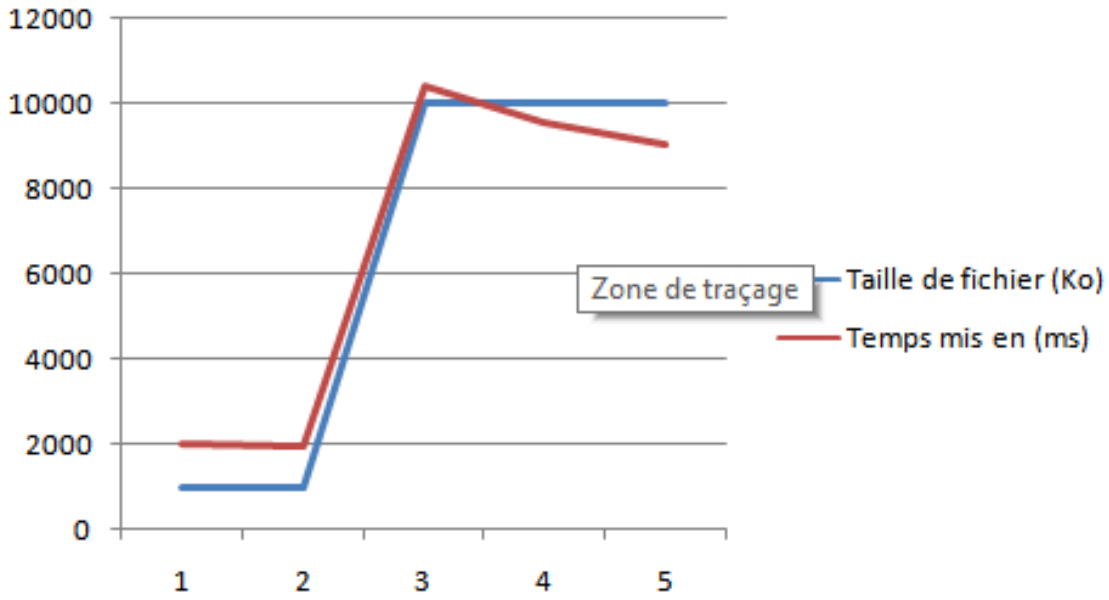

FIGURE 3 – Le temps mis en fonction de taille de fichier (old version of Hidoop)

4.1.1 Analyse de résultat :

Analyse de temps d'exécution en fonction de la taille de fichier D'après la comparaison de deux figures précedentes on peut remarquer que le temps mis pour l'exécution de fichiers est supérieur au temps d'exécution d'Hidoop sans la partie optimisation, c'est parce que la gestion de fichiers se fait en parallèle ce qui prend plus de temps pour l'exécution des fichiers de petites taille.
Car, les performances d'un calcul parallèle s'évaluent à l'aide de plusieurs mesures comme l'accélération ou l'équilibrage, donc pour des fichiers de petites taille, il induit des attentes inutiles sur les processeurs les moins chargés, ce qui prend plus de temps aussi pour diviser les tâches entre plusieurs reducers pour un calcul simple.

L'impact de nombre de workers sur le résultat : On remarque que pour un fichier de même taille (10Mo) en changeant le nombre de workers entre (5, 6 et 8) la différence entre le temps mis pour chaque nombre de workers est non significatif, le temps varie entre 26928ms et 27780ms. On peut conclure que pour des fichiers de petites taille le changement de nombre de workers ne joue pas vraiment un rôle important. Même des fois il a un effet négatif, puisqu'il a besoin de plus de temps pour lancer plusieur workers.

4.2 Pour les fichiers de grande taille :

Pour la nouvelle version d'Hidoop avec optimisation :

Nombre de workers	Taille de fichier (Mo)	Temps mis en (s)
8	100	190,067
5	100	190,651
5	200	311,541
5	500	673,245
5	1000	1313,845
8	1000	913,845
5	2000	1201,2
8	4000	1623,7

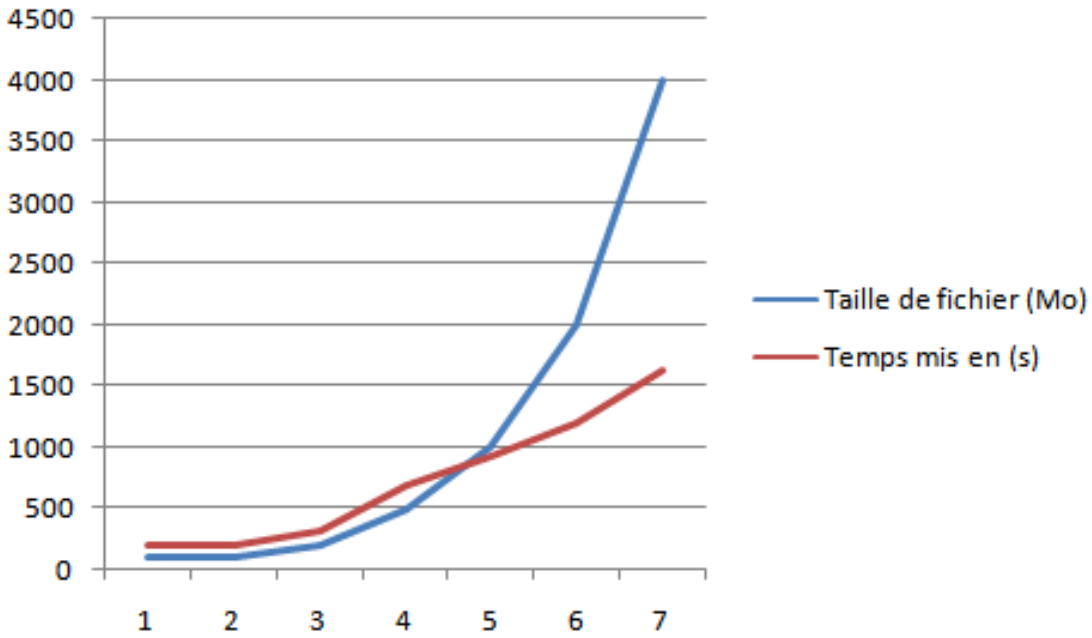

FIGURE 4 – Le temps mis en fonction de taille de fichier (new version of Hidoop)

Les résultats obtenus par l'ancienne version d'Hidoop sans optimisation :

5	100	1056,78
5	260	1995,23
8	550	2151,77
10	550	1845,23
10	1000	4122,14
10	2000	10711,144

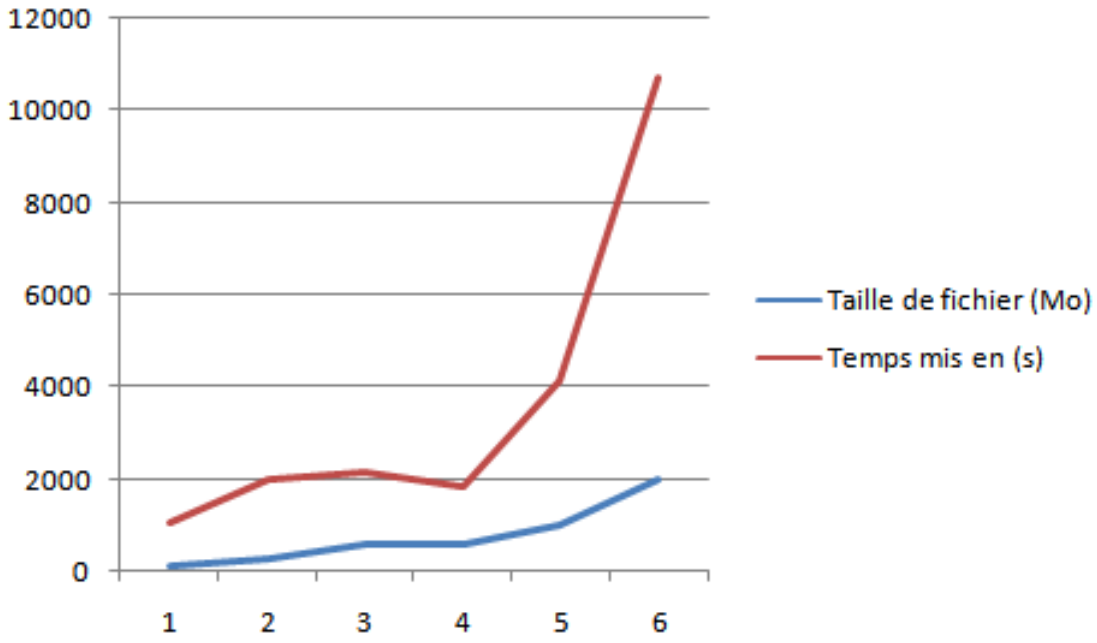

FIGURE 5 – Le temps mis en fonction de taille de fichier (old version of Hidoop)

4.2.1 Analyse de résultat :

Analyse de temps d'exécution en fonction de la taille de fichier C'est ici qu'intervient les changements effectuer.Dans les deux figures précédentes,on remarque que le temps mis pour l'exécution de fichiers est bel et bien inférieur au temps d'exécution d'Hidoop sans la partie optimisation. La différence entre les deux temps est bien significatif, ce qui montre l'importance de cette optimisation. Par exemple pour un fichier de 200Mo le temps mis par l'ancienne version est dix fois plus grand que celui mis par l'application Hidoop avec optimisation.

L'impact de nombre de workers sur le résultat : On remarque que pour un fichier de même taille (1Go) en changeant le nombre de workers entre (5 et 8) la différence entre le temps mis pour chaque nombre de workers est bien significatif, elle vaut 300s. On peut conclure que pour des fichiers de grande taille le changement de nombre de workers joue un rôle important.

5 Estimation de Π par une méthode de Monte-Carlo

Le principe est de générer des points aléatoirement dans l'intervalle $[-1, 1]$. On calcule ensuite le nombre de points dans le cercle et dans le carré. Une estimation de la valeur de Π est ainsi obtenue par : $4 \times \frac{nbPointsCercle}{nbPointsCarr}$.
La classe QuasiMonteCarlo.java du répertoire application contient l'implantation de cette méthode en implémentant l'interface MapReduce. Nous obtenons un résultat de 3.19 pour 1000 points.