Travaux pratiques - implémentation de Hadoop MapReduce "from scratch" en Java

Table des matières

Présentation du projet :

L'objectif de ce projet est de recréer en Java un système MapReduce basique, afin d'en comparer l'efficacité face à un système classique. Ici l'objectif est d'effectuer une tâche de comptage de mots dans un fichier texte. La méthode classique consiste à parcourir ce fichier pour en compter les mots un à un. Le système MapReduce consiste quant à lui à splitter le fichier initial en sous-fichiers qui sont alors distribués à différentes machines. Celui-ci effectue alors une phase de map et de shuffle. Cette phase va ainsi permettre de séparer les mots et d'envoyer toutes les occurrences d'un même mot sur la même machine. Une fois ceci-fait commence la phase de reduce durant laquelle chaque machine va compter les mots sur les shuffles reçus. Enfin les résultats des reduces de chaque machine sont renvoyés sur la première qui les concatène.

Toutefois le système MapReduce trouve ses limites en temps d'exécution à cause des temps de communication et de transfert entre les différentes machines du système. A cela s'ajoute une seconde limite de lecture et d'écriture sur disque. L'objectif est ici de comparer ces deux systèmes mais aussi de visualiser la loi d'Ahmdal.

Par ailleurs ce projet est aussi pour moi une découverte de java que je n'avais pratiqué alors que pendant l'initiation du Mastère spécialisé. Il m'a permis de découvrir les différents objets de ce langage et de mieux comprendre leur fonctionnement.

Problématiques rencontrées et solutions :

Présentation du système :

Le système MapReduce fonctionne à l'aide de deux éléments principaux. Le premier est une Classe Master située sur le pc de départ (le Master Node). Cette classe est celle permettant de lancer les exécutions sur les différentes machines du système. Une seconde Class appelée slave est-elle exportée en jar exécutable et est distribuée sur les différentes machines du système par le Master. Le Master pourra ainsi lancer les exécutions des slaves sur les machines du système au travers de commandes ssh.

Afin de vérifier la validité des résultats une fonction permet de comparer les fichiers de résultats du système MapReduce avec un système de wordcount classique. De plus les temps d'exécution des différentes parties seront affichés ce qui permettra de comparer les performances des deux systèmes.

Choix des machines :

Bien que lors des premières versions une liste de machines était donnée en Input, il est plus simple de laisser le programme sélectionner les machines auxquelles se connecter. En effet, certaines machines plus sollicitées que d'autres par les utilisateurs peuvent parfois s'avérer plus lentes à répondre au commande SSH. L'un des inputs du la classe master est ainsi un fichier csv contenant le nom de

machines de TP de l'école. Une fonction va alors être appelée et va tester la vitesse de réponse à une requête ssh. Selon le nombre de machines souhaité, mis en input aussi, la fonction va renvoyer une liste des x machines les plus rapides à la réponse, ainsi qu'une liste contenant la liste des machines ayant répondu en moins de n secondes et qui pourront servir de backup en cas de défaillance d'une des machines.

Pour stocker cette liste de machines j'ai créé une classe machine_cluster composée de deux éléments, les machines utilisées et celle non utilisées. Les premières sont stockées dans un HashMap<String, String> les associant au numéro du split sur lequel elle devra travailler. La seconde est une simple ArrayList contenant l'ensemble des autres machines ayant répondu, classées du plus au moins rapide.

Enfin dans le cas où le nombre de machines disponibles était inférieur au nombre de machines souhaitées, celui-ci est revu à la baisse et un message informe de l'évènement. Dans tous les cas le cluster sélectionné est affiché au début de l'exécution.

Split des données :

La seconde étape est donc le split des données, connaissant désormais le nombre de machines dans notre cluster.

Au sein de la classe deux idées sont ressortis sur la façon de procéder lors de cette étape. D'un côté certains ont décidé de créer un fichier de splits par ligne du fichier texte, puis de les distribuer selon une table de correspondance sur les pc. Pour moi cette solution n'était pas optimale car créant une multitude de fichier et donc autant de commande ssh afin d'effectuer les transferts scp. J'ai pour ma part décidé de partir sur un fichier de splits par machine dans le cluster.

Pour splitter les données la méthode la plus intuitive a d'abord été de parcourir le fichier d'input et de copier les lignes dans les fichiers de retour selon la valeur de numéro ligne modulo nombre d'output. Le problème est que cela est très couteux en temps du fait de la lecture et de l'écriture des fichiers, et le temps de split pouvait assez vite approximer le temps de calcul obtenu par un système classique. Une solution que j'ai pu trouver a été l'utilisation de la ligne de commande de linux split. Celle-ci permet de découper un fichier texte en sous fichiers selon une taille en bytes spécifiée. Après avoir récupéré la taille du fichier d'input et connaissant le nombre de splits il est aisé de découper le fichier texte à l'aide de cette fonction et du processbuilder. Ces splits sont ensuite renommés en récupérant leur nom par la commande 'ls' puis en bouclant et en utilisant la commande 'mv'.

Par la suite j'ai développé une nouvelle méthode de split en Java. Ainsi à l'aide d'un MappedByteBuffer appliqué sur un InputStream, lui-même appliqué sur un FileReader du fichier il m'était possible d'accéder à un endroit précis dans le fichier. Ainsi en récupérant la taille du fichier, il ne restait plus qu'à découper celui-ci en n splits en cherchant le byte séparant en taille équivalente. Il m'a fallu prendre en compte qu'un byte pouvait se trouver au milieu d'un mot en cherchant les bytes se terminant contenant un espace. Enfin une des difficultés sur les plus gros fichiers a été de ne pas dépasser la taille du MappedByteBuffer, celui-ci ne pouvant pas dépasser l'INTEGER._MAX_VALUE. Pour ce faire plusieurs threads sont affectés à l'écriture d'un split si la taille de celui-ci dépasse la taille maximum.

Cette seconde méthode a été pour moi l'occasion de tester de nombreux objets et de comparer leur performance. Je me suis ainsi rendu compte que l'utilisation du transferTo de la classe FileChannel était moins efficace que l'utilisation du write sur celle-ci prenant en paramètre le MappedByteBuffer.

Les résultats avec ces deux méthodes seront affichés dans la partie résultats.

Déploiement des données, gestion des slaves et multiprocessing :

Le déploiement des données comme la gestion des slaves s'effectue par commande ssh et transfert scp initié par le master. Toutefois il est nécessaire de s'assurer que le processus est terminé avant de passer au suivant. En effet si le map est lancé avant la réception du fichier par la machine, celle-ci ne pourra fonctionner correctement. Pour attendre la fin de l'exécution de la commande l'élément process contient une méthode waitfor() qui est bloquante sur le thread. Le problème étant alors que si l'on attend que la machine ait fini avant de passer à la suivante, alors l'avantage de lancer les tâches sur plusieurs machines est perdu. Pour remédier à ce problème j'ai donc fait appel au framework ExecutorService de Java. Celui-ci via des classes runnables et callables permet de lancer en simultané des actions sur plusieurs threads. De plus la méthode InvokeAll des callables permet d'attendre la fin de l'exécution de l'ensemble des threads.

Ainsi cela m'a permis de lancer les tâches sur l'ensemble des machines et d'attendre jusqu'à ce que toutes celles-ci soient terminées. Bien évidemment lors de la phase de recherche de cluster, étant donné que l'ensemble des machines ne répond pas il est nécessaire d'imposer un timeout au waitfor.

Enfin l'utilisation de la classe Callable permet d'obtenir des valeurs de retour sur chacun des process via la classe Futurs. Ainsi cela permet de vérifier que le process en plus d'être terminé, a bien fonctionné sans soucis. Cela permet notamment d'entamer la procédure de reprise sur panne si l'une des machines est défaillante.

Limitation de l'accès au disque et des transferts SCP :

Les opérations les plus coûteuses en temps dans un système MapReduce sont celles d'IO sur disque et de transfert. L'objectif pour optimiser le système a donc été de minimiser celles-ci. Pour ce faire j'ai donc modifié le slave initial qui écrivait le résultat de la phase de Map avant de le lire durant la phase de shuffle, afin que celui-ci écrive directement le shuffle en mappant.

Par ailleurs l'idée initiale donnée dans le TP était d'écrire des shuffles dont les noms seraient <HashCode>-<Host>, ce qui impliquait l'écriture de nombreux fichiers différents (un pour chaque mot différent), puis leur envoi. Pour éviter cela j'ai décidé d'écrire les fichiers sous la forme <receiver>-<sender>, ce qui limite le nombre de fichiers au nombre de machines.

Enfin, ma dernière idée a été d'appliquer un reduce avant le shuffle pour réduire considérablement la taille des fichiers. Cette idée est développée dans la partie suivante.

Enfin pour écrire plus vite dans ces fichiers j'ai décidé d'utiliser un BufferedWritter plus rapide qu'un FileWritter classique. De plus pour ne pas avoir à ouvrir fermer le BufferedWritter à chaque mot, j'ai créé un objet HashMap<Integer, BufferedWritter> qui permet de sélectionner quel BufferedWritter utiliser directement.

Changement de la méthode MapReduce proposée en cours :

Là où la phase de déploiement est difficilement optimisable du fait de la bande passante limitée, les phases de split, map, shuffle et reduce le sont beaucoup plus. Après avoir passé l'étape de merging des phases de map et shuffle j'ai cherché où se situait la plus grosse perte de temps sur cette partie et il s'agissait bien évidement du shuffle. Celui-ci dépendant principalement de la limitation de la bande passante l'idée a été de réduire la taille de ces shuffles. Comment ? En transformant la phase de map par un wordcount. Le dictionnaire créé et non copié sur disque est alors distribué dans les fichiers de shuffle selon le hashcode module nombre de machine, et contient sur chaque ligne le mot et son occurrence. L'avantage ? La taille des fichiers de shuffles passe de plusieurs centaines de Mo, voire de Go, à quelques dizaines au plus centaines de Ko. Une version modifiée du reduce sur les slaves va ensuite concaténer les shuffles reçus en une fraction de secondes et les renvoyer au master. Les résultats de cette méthode seront aussi affichés dans les résultats.

Gestion de la reprise sur panne :

L'utilisation de la classe Callables comme indiqué au-dessus nous permet de récupérer un retour sur l'exécution de chaque thread. Ainsi selon le retour obtenu il est possible de savoir si l'une des machines ne fonctionne plus. Dans ce cas et selon la phase à laquelle nous nous trouvions il est possible d'entamer une reprise. Si l'erreur survient lors de la phase de déploiement des données alors il suffit de sélectionner une nouvelle machine dans la liste des backups pour remplacer la machine défaillante. Si l'erreur survient durant la phase de map, il faut alors remplacer la machine défaillante comme décrit précédemment puis relancer cette étape dessus. Si nous en sommes au shuffles, la méthode est la même que précédemment mais implique de relancer l'étape de shuffles sur l'ensemble des machines. C'est d'ailleurs l'un des désavantages des méthodes map_shuffle et map_reduce_shuffle qui évitant l'écriture sur disque nécessite donc de relancer l'étape sur l'ensemble des machines après avoir redéployé sur la nouvelle. Cependant bien que cela puisse paraitre limitant, à condition de considérer que ces étapes prennent un temps similaire sur l'ensemble des machines, alors dans le cas d'une panne le fait de relancer sur l'ensemble des machines ne coute pas plus de temps que de le relancer sur une seule.

L'organisation du code et son appel dans le main du master :

L'un des objectifs a aussi été pour moi de créer un projet bien organisé avec une structure cohérente, lisible et facilement utilisable du code. Pour ce faire j'ai donc séparé l'ensemble des fonctions en de nombreuses classes comme ci-dessous :

1: Organisation du code

Par ailleurs l'appel aux fonctions par le main se fait de façon très simple comme suit :

```
public static void main(String[] args) throws IOException, InterruptedException, ExecutionException {
    //main to launch for the project

    //find who is the user to adapt folder name during the whole project, only the Inputs for slave jar need to be modify
    String current_user=functions.get_user();

    //create a txt file, if already have one comment this part but change the Input path
    //use the file bible.txt that makes roughly 4Mo, a 500 multiplier in the create_file_cmd represent a 1.9Go output file
    String[]create_file_cmd= {"5000"};
    make_big_txt_file.main(create_file_cmd);

    //select your Inputs bellow
    ArrayList<String> machines= functions.get_machine("Machines_TP.csv");
    String slave_jar_path="/cal/homes/rgenet/slave.jar";
    String Input="/tmp/data_file/bible_time"+create_file_cmd[0]+".txt";

    //select your parameters :
    //first : spliting function takes "multiproc" or "linux"
    //second: compression take true or false
    //third: mode takes "classic", "map_shuffle" or "map_reduce_shuffle"
    mapreduce_parameters my_parameters= new mapreduce_parameters("multiproc", false, "map_reduce_shuffle");

    //launch map reduce
    int machines_getted=launch_map_reduce(machines, slave_jar_path, 12,Input, my_parameters, current_user);


    //launch classic system with sequential version
    System.out.println("Using wordcount sequential");
    launch_classic_system(Input,"classic", null, current_user);
    //sleep to let CPU cool
    Thread.sleep(20000);
    //launch classic system with multiprocessing version
    System.out.println("Using wordcount multiprocessed");
    launch_classic_system(Input,"multi", "12", current_user);

    //check if all files are equals
    System.out.println("Résultat test de similarité des resultat du systeme classique multithread et séquentiel: "+file_comparator.compare_file("/tmp/"+current_user+"_resultat/resultat_sequential_multi.txt", "/tmp/"+current
    System.out.println("Résultat test de similarité des resultat du systeme classique multithread et MapReduce: "+file_comparator.compare_file("/tmp/"+current_user+"_resultat/resultat_sequential_multi.txt", "/tmp/"+current_
    System.out.println("Résultat test de similarité des resultat du systeme classique séquentiel et MapReduce: "+file_comparator.compare_file("/tmp/"+current_user+"_resultat/resultat_sequential_classic.txt","/tmp/"+current_

}
```

2: Organisation du main

Comme vous pouvez le voir le main est très réduit et ne sert qu'à appeler les fonctions principales du projet. Toutefois il laisse la possibilité de modifier très simplement les paramètres des fonctions. Tout d'abord il prend en compte automatiquement le nom d'utilisateur, ce qui signifie que n'importe quelle personne se connectant sur un pc de Telecom sera en mesure de lancer ce programme en modifiant

uniquement le chemin d'accès au jar, le fichier contenant le texte de base et la liste des machines étant inclus dans le projet.

De plus une classe mapreduce_parameters permet d'appliquer très simplement le type d'algorithme à utiliser, pour un nombre de combinaison totale de 12 possibilités différentes.

Enfin le main contient aussi un appel à la fonction créant un fichier d'input, et lance à la fois les systèmes MapReduce et classique pour ensuite vérifier l'égalité des résultats.

Les essais qui ont échoué :

Ayant pu voir qu'une des limites du système MapReduce résidait dans le temps nécessaire au transfert des données, j'ai eu l'idée de compresser celle-ci avant l'envoi. Ma première idée a été d'utiliser la compression des données directement au moment du split via l'option de commande –filter et l'utilisation de la fonction de compression gunzip pour sa rapidité. Cependant le temps additionnel dû à la compression des fichiers ne compensait pas le gain lors du transfert. Afin d'accélérer cette étape j'ai implémenté une classe Callable afin de pouvoir appliquer en même temps sur l'ensemble des splits une compression et ainsi répartir la charge de travail. Bien que cela accélère grandement le processus de compression, en divisant le temps d'exécution par 3 durant la phase, le temps additionnel dû à la compression et à la décompression sur les machines receveuses n'était pas compensé par la réduction de la durée du déploiement. L'option reste tout de même disponible en changeant uniquement le paramètre compression de l'objet mapreduce_parameters. Cela peut notamment servir dans le cas de l'utilisation du programme sur un réseau à bande passante plus limité. Les résultats seront affichés dans la partie suivante.

Par ailleurs j'ai aussi fait plusieurs essais afin d'accélérer le processus du système classique, notamment en remplaçant le HashMap<String, Integer> en un HashMap<Integer, HashMap<String, Integer>. L'idée était ainsi de stocker des sous dictionnaires en fonction du HashCode modulo une taille de split, afin de chercher les éléments dans des dictionnaires plus petits. Bien que cela n'ait pas amélioré la vitesse d'exécution, cela ne l'a toutefois pas diminué, les performances des deux systèmes étaient similaires. Une des hypothèses est que ce genre de découpage est peut-être déjà effectué au sein même de la classe HashMap de façon sous-jacente.

J'ai aussi essayé de passer par une autre méthode que l'utilisation de l'ExecutorService pour lancer le processus de déploiement et comparer les temps d'exécution. Pour ce faire j'ai créé un Hashset de ProcessBuilder appelé dans un second HashSet de process lancé en séquentiel. Ensuite avec une seconde boucle les processus sont attendus avec waitfor() pour attendre l'exécution de chaque commande puis détruit. Le temps d'exécution est alors deux fois plus long avec cette méthode qu'avec le multiprocessing utilisé grâce à l'ExecutorService. Cette méthode n'a donc pas été sélectionnée et n'est plus disponible dans la dernière version du projet.

Enfin j'ai essayé afin d'accélérer les splits, d'associé multithreading et multiprocessing. En effet, en comparaison du wordcount en multiprocessing qui tirait pleinement parti de la puissance du CPU et qui sera discuté plus tard, la phase de split en multiprocessing n'en tirait pas le maximum. Ainsi j'ai appelé sur chaque process des threads copiant des parties différentes du fichier mais utilisant le même FileChannel in et out. Le problème ici est que cette méthode n'est pas thread safe, et bien que beaucoup plus rapide (3 fois), les bytes envoyés par les différents threads du process dans le même fichier se chevauchaient parfois ce qui mélangeait certains mots.

Résultats obtenus :

Le fichier d'Input :

Les systèmes MapReduce n'ont d'intérêt que lors de l'utilisation de fichiers volumineux. En effet, certaines exécutions du système MapReduce ayant des temps d'exécution indépendants de la taille du fichier, leur temps d'exécution seul dépasse facilement le temps d'exécution sur un système classique. Afin de créer ce fichier j'ai pris un fichier texte initial assez volumineux contenant la bible. Ce fichier faisant uniquement 4Mo, j'ai créé une fonction qui selon un multiplicateur x créait un fichier contenant x fois le nombre de lignes du fichier initial en sélectionnant aléatoirement les lignes dans le fichier. J'ai ensuite pu obtenir plusieurs fichiers d'input afin de tester l'efficacité du système MapReduce. La taille des fichiers utilisés a donc été de 1.8Go, 5.7Go, 18Go, et 113Go.

Le système classique de référence

Le système de référence est assez important ici car l'objectif est de comparer des performances. Ainsi j'ai passé un peu de temps pour trouver quelle optimisation permettait de rendre l'exécution la plus rapide possible. J'ai ainsi remarqué que l'utilisation de lecteur de type BufferedReader permettait de diviser par 4 à 5 la durée d'exécution par rapport à un scanner. J'ai aussi pu voir que le fait d'imposer une taille de buffer au BufferedReader à sa création rendait l'exécution plus lente, et ce même après avoir cherché la taille minimisant le temps d'exécution. Enfin pour le stockage et le calcul mon choix s'est tourné vers une HashMap. Ce type d'objet permet d'accéder facilement à l'endroit où est rangé un mot et donc d'effectuer très rapidement le Word count.

Par la suite j'ai cherché à encore optimiser ce programme, notamment après le développement de la méthode de split sur plusieurs threads. Ainsi l'idée a été la même, découper la lecture du fichier grâce au MappedByteBuffer, puis en récupérer les mots via un CharBuffer pour ensuite créer des dictionnaires sur chacune des parties. Les dictionnaires créés sont renvoyés grâce aux futurs de la classe Callable et sont ensuite mergés dans un dictionnaire final. Cette étape a été très longue à réaliser sans erreur, notamment du fait des limitations de taille du CharBuffer et de la nécessité de la méthode toString à utiliser ici en lieu et place d'une lecture des éléments. Cela m'a permis de diviser le temps par trois sur les fichiers de 2Go, un peu moins sur les plus gros. Cela sera discuté dans les résultats.

J'ai aussi par curiosité reproduit le même programme en python (séquentiel), et pour rester au plus proche j'ai utilisé un BufferedReader pour la lecture (avec io.open et « rb » en paramètre) ainsi qu'un dictionnaire à la place du HashMap.

L'affichage des résultats :

Les résultats sont affichés de la façon suivante.

```
creating a input file from bible.txt with a multiplier of 5000 estimated output size: 18,85Go
file created in 94 seconds
Launching map reduce system
The parameters of the mapreduce system are:
split function: multiproc, compression: false, mode: map_reduce_shuffle
number of machine asked: 12
Creating the cluster
not enougth machine find, cluster size reduce to 11
the cluster is:
{0=rgenet@tp-1a207-12, 1=rgenet@tp-1a201-10, 2=rgenet@tp-1a201-39, 3=rgenet@tp-1a201-03, 4=rgenet@tp-1a207-18, 5=rgenet@tp-1a260-31, 6=rgenet@tp-1a201-25, 7=rgenet@tp-1a201-31, 8=rgenet@tp-1a207-27, 9=rgenet@tp-1a207-10, 10=rgenet@tp-1a201-34}
Creating cluster elapsed time in s: 10
start splitting
Preparing folder and splits elapsed time in s: 372
launch deployment
Deployment on cluster elapsed time in seconds: 292
launch map reduce shuffle
finish map reduce shuffle
mapping reducing shuffling on cluster elapsed time in seconds: 73
launch reduce
reduce finished
reducing elapsed time in seconds: 21
creating_result_file
creating result file elapsed time in seconds: 0
finish
All mapreduced process elapsed time in seconds: 773


Using wordcount sequential
time elapsed (ms) : 350290
Using wordcount multiprocessed
time elapsed (ms) : 174107
Résultat test de similarité des resultat du systeme classique multithread et séquentiel: true
Résultat test de similarité des resultat du systeme classique multithread et MapReduce: true
Résultat test de similarité des resultat du systeme classique séquentiel et MapReduce: true
```

3: Affichage des résultats

Cette présentation permet à la fois de décomposer les temps d'exécution du système MapReduce, mais aussi de les comparer au système classique tout en vérifiant que les fichiers d'output sont bien équivalents. Par ailleurs, afin de comparer des éléments comparables, entre chaque exécution d'un Word count (MapReduce, système unique utilisant le multiprocessing puis séquentiel) un temps de pause est imposé, selon la taille du fichier traité auparavant, afin que le processeur se retrouve dans le même état qu'auparavant (pour ne pas subir d'effet de thermal throttling dû à l'exécution précédente).

J'ai aussi pu remarquer que le nombre important de commandes ssh opéré par MapReduce entrainait un ralentissement de celles-ci par la suite, que je ne saurais expliquer. J'ai bien évidement vérifié que tous les process était bien clos. Pour prévenir d'un effet dû à cela j'ai donc changé de machine entre chaque lancement de calcul.

Les résultats chiffrés :

Afin de vous présenter les résultats obtenus j'ai préparé ici quelques tableaux Excel afin de les résumer. Je les commenterai dans la partie suivante.

1,9Go File				
Système classique		**Système MapReduce en Java (split linux)**		
Total en Java	33,6	Cluster size : 5 machines	Create cluster	20
Total en Python	150		Split	11
Total en java (multiprocessing)	8,3		Deploy	20
			Map-Suffle	45
			Reduce	14
Total	2127		Total	112
		Cluster size : 10 machines	Create cluster	20
			Split	11
			Deploy	31
			Map-Suffle	105
			Reduce	20
Total	2127		Total	188
		Cluster size : 23 machines	Create cluster	20
			Split	11
			Deploy	38
			Map-Suffle	206
			Reduce	43
			Total	319
		Système MapReduce version Map-Reduce-Shuffle-Reduce (split multithread)		
		Cluster size : 3 machines	Create cluster	10
			Split	0
			Deploy	38
			Map-Reduce-Suffle	26
			Reduce	5
			Total	80
		Cluster size : 6 machines	Create cluster	10
			Split	0
			Deploy	23
			Map-Reduce-Suffle	59
			Reduce	13
			Total	106
		Cluster size : 9 machines	Create cluster	10
			Split	0
			Deploy	33
			Map-Reduce-Suffle	83
			Reduce	25
			Total	152

4 : Résultat sur un fichier de 1.9Go

5,7Go File				
Système classique		**Système MapReduce**		
Total en Java	95	Cluster size : 2 machines	Create cluster	4
Total en Python	455		Split	67
(multiprocessing)	24		Deploy	60
			Map-Suffle	225
			Reduce	44
Total	2127		**Total**	402
		Cluster size : 3 machines	Create cluster	4
			Split	53
			Deploy	56
			Map-Suffle	95
			Reduce	35
Total	2127		**Total**	244
		Cluster size : 5 machines	Create cluster	10
			Split	34
			Deploy	58
			Map-Suffle	69
			Reduce	24
Total	2127		**Total**	197
		Cluster size : 10 machines	Create cluster	10
			Split	55
			Deploy	65
			Map-Suffle	102
			Reduce	29
			Total	263
		Système MapReduce (split multithread)		
		Cluster size : 14 machines	Create cluster	10
			Split	54
			Deploy	56
			Map-Suffle	144
			Reduce	38
			Total	303
		Système MapReduce version Map-Reduce-Shuffle-Reduce (split multithread)		
		Cluster size : 3 machines	Create cluster	10
			Split	46
			Deploy	58
			Map-Reduce-Suffle	26
			Reduce	6
			Total	147
		Cluster size : 6 machines	Create cluster	10
			Split	38
			Deploy	60
			Map-Reduce-Suffle	51
			Reduce	14
			Total	176
		Cluster size : 9 machines	Create cluster	10
			Split	33
			Deploy	56
			Map-Reduce-Suffle	93
			Reduce	19
			Total	212

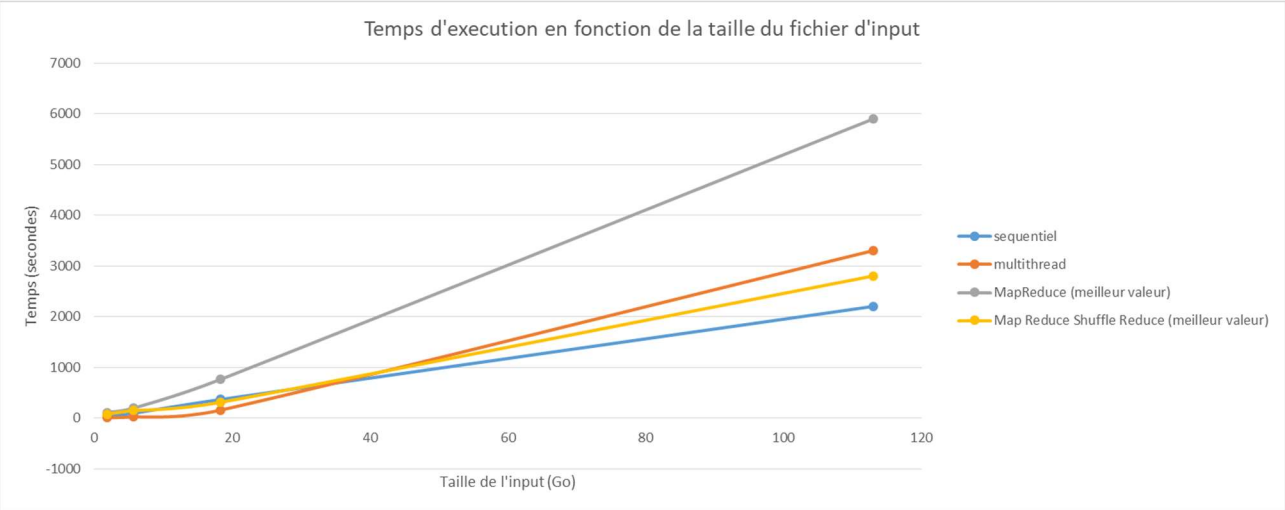
5 : Résultat sur un fichier de 5.7Go

18Go File				
Système classique (avec Scanner et non BufferedReader)		**Système MapReduce**		
Total:	1300-1500	Cluster size : 49 machines	Create cluster	20
Système classique (avec multiprocessing)			Split	254
Total	156		Deploy	296
Système classique			Map-Suffle	139
Total	365		Reduce	52
[redacted]			**Total**	761
[redacted]		**Système MapReduce version Map-Reduce-Shuffle-Reduce (split multithread)**		
[redacted]		Cluster size : 8 machines	Create cluster	10
[redacted]			Split	349
[redacted]			Deploy	220
[redacted]			Map-Reduce-Suffle	74
[redacted]			Reduce	21
[redacted]			**Total**	673
[redacted]		Cluster size: 11 machines	Create cluster	10
[redacted]			Split	372
[redacted]			Deploy	292
[redacted]			Map-Reduce-Suffle	73
[redacted]			Reduce	21
[redacted]			**Total**	773
[redacted]		**Système MapReduce version Map-Reduce-Shuffle-Reduce (split linux**		
[redacted]		Cluster size : 12 machines	Create cluster	10
[redacted]			Split	249
[redacted]			Deploy	256
[redacted]			Map-Reduce-Suffle	75
[redacted]			Reduce	20
[redacted]			**Total**	610
[redacted]		Système MapReduce (map et shuffle séparés)		
[redacted]		Cluster size : 9 machines	Split	223
[redacted]			Deploy	215
[redacted]			Map	47
[redacted]			Shuffle	166
[redacted]			Reduce	54
[redacted]			**Total**	705
[redacted]		Système MapReduce compression multithread		
[redacted]		Cluster size : 6 machines	Create cluster	10
[redacted]			Split and compress	455
[redacted]			Deploy	73
[redacted]			Map-Suffle	220
[redacted]			Reduce	65
[redacted]			**Total**	825
[redacted]		Système MapReduce compression multithread		
[redacted]		Cluster size : 24 machines	Create cluster	10
[redacted]			Split and compress	609
[redacted]			Deploy	73
[redacted]			Map-Suffle	231
[redacted]			Reduce	41
[redacted]			**Total**	966
[redacted]		Système MapReduce compression multithread		
[redacted]		Cluster size : 3 machines	Create cluster	10
[redacted]			Split and compress	843
[redacted]			Deploy	73
[redacted]			Map-Suffle	758
[redacted]			Reduce	101
[redacted]			**Total**	966

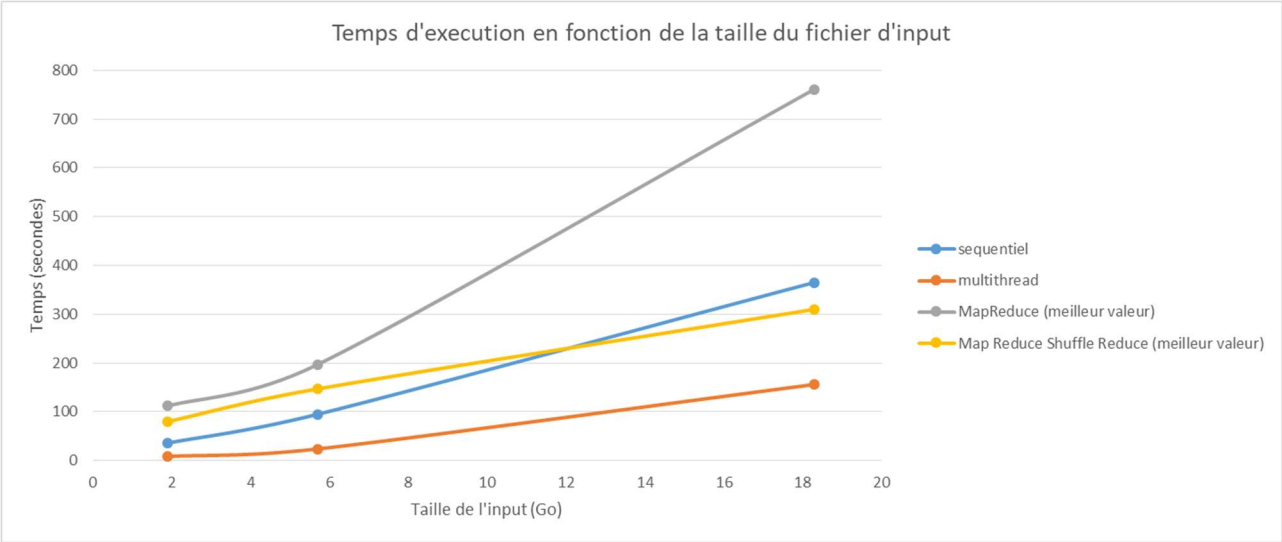
6 : Résultat sur un fichier de 18GO

113Go File				
Système classique		**Système MapReduce**		
Total en Java	2201	Cluster size : 49 machines	Create cluster	20
Total en java (multiprocessing)	3300		Split	1678
			Deploy	3621
			Map-Suffle	320
			Reduce	256
			Total	5896
		Système MapReduce (compression multithread + split linux		
		Cluster size : 10 machines	Create cluster	10
			Split	4227
			Deploy	446
			Map-Suffle	1114
			Reduce	230
			Total	6029
		Système MapReduce version Map-Reduce-Shuffle-Reduce (split multithread)		
		Cluster size : 3 machines	Create cluster	10
			Split	2900
			Deploy	2015
			Map-Reduce-Suffle	156
			Reduce	27
			Total	5109
		Système MapReduce version Map-Reduce-Shuffle-Reduce (split linux)		
		Cluster size : 10 machines	Create cluster	10
			Split	1462
			Deploy	1098
			Map-Reduce-Suffle	214
			Reduce	16
			Total	2801

7 : Résultat sur un fichier de 113Go



8: Résultats globaux (1.9-113Go)



9: Résultats globaux (1.9-18Go)

Description des résultats

Les résultats semblent assez décevants mais sont tout à fait compréhensibles et explicables. Il apparait sur le premier tableau pour un fichier de 2Go que la version séquentielle en Java est bien plus rapide que la version MapReduce, elle-même plus rapide que l'exécution en python. En effet il ne faut que 36 secondes pour le système classique séquentiel contre au minimum 112 pour le système MapReduce. L'utilisation de la méthode map reduce shuffle associé au split en multiprocessing permet de gagner un peu de temps d'exécution et d'atteindre 80 secondes au minimum, toujours plus qu'un système séquentiel classique. L'utilisation d'un système classique associé à l'utilisation du multiprocessing permet de battre tous les records en descendant à 8.5 secondes, soit moins que le temps nécessaire uniquement à la création du cluster.

En augmentant la taille des fichiers on peut voir que le système séquentiel augmente de façon linéaire là où les autres systèmes beaucoup moins.

-Le système map reduce classique voit sa durée augmenté en pourcentage moins vite dans un premier temps que le système classique du fait de l'amortissement des taches non évitable (création du cluster par exemple). Toutefois en continuant d'agrandir les fichiers cette croissance en pourcentage devient plus grande que le système séquentiel classique, et cela s'explique par l'augmentation des temps de transfert du fait de l'augmentation de la taille des fichiers

-Le système map reduce shuffle lui voit aussi dans un premier temps sont temps d'exécution croitre moins vite que le système séquentiel classique, mais de façon plus marqué que le système map reduce classique. En effet comme décrit précédemment l'utilisation de ce système fait des phases de shuffle et de reduce des phases à durée presque indépendante de la taille du fichier. Ainsi la durée de ce système bat le système séquentiel arrivé à une certaine taille (18Go). Cependant sur les très gros fichiers (113Go ici) l'accroissement du temps de split et de déploiement dépasse alors la durée du système séquentiel.

-Le système classique utilisant le multiprocessing voit ses résultats battre tous les autres systèmes sur les fichiers assez petits, mais avec l'accroissement de la taille des fichiers l'effet du

thermal throttling réduit grandement son efficacité et le fait ralentir au point de devenir plus long qu'un système séquentiel classique et même que le système map reduce shuffle.

Explication des résultats

Il convient maintenant d'expliquer ces résultats et leur cohérence. Je vais donc expliquer le temps nécessaire à chaque phase pour ensuite voir pourquoi dans le cas du système MapReduce ici présent il est physiquement impossible à notre système de battre le système classique, mais aussi en conclure qu'il existe bien des cas où celui-ci aurait des avantages.

Phase de création du cluster :

Comme vous avez pu le voir, durant cette phase le temps est similaire sur la plupart des exécutions et fixe. En effet, le temps d'exécution de cette phase est paramétré manuellement. Il s'agit en réalité du temps accordé à la requête ssh cherchant à voir si un pc est bien disponible (autrement dit le Timeout du ProcessBuilder. Le choisir petit permet de gagner du temps, toutefois cela limite par la suite le nombre de machines à disposition, ce qui peut réduire la taille du cluster demandé.

Utiliser un fichier d'inputs contenant moins de machine permet aussi de réduire la durée du TimeOut car cela réduit le nombre de connexion ssh en parallèle et donc la durée nécessaire à l'obtention des réponses.

Phase de split :

La phase de split est effectuée de deux manières, soit par la commande de linux soit par l'utilisation du multiprocessing.

Lors de l'utilisation de la fonction de Linux il apparait que cette phase croit de manière assez linéaire avec la taille du fichier, mais dépend tout de même de l'état du système (des autres tâches en cours d'exécution). Cette phase qui n'est pas évitable et dont le temps n'est pas diminuable est d'autant plus gênante que son temps d'exécution représente déjà 2 tiers du temps nécessaire au système linéaire pour effectuer le comptage des mots.

Lors de l'utilisation du multiprocessing les résultats sont différents. Pour les petits fichiers (<5Go) le split est bien plus rapide et son temps devient négligeable. Toutefois sur les fichiers plus gros la durée d'exécution de la fonction croit plus vite que celle utilisé par linux ce qui amène à des temps d'exécution de split plus important au-delà d'une certaine taille.

Phase de déploiement :

La phase de déploiement est aussi l'un des facteurs explicatifs majeurs du temps d'exécution. En effet cette phase ne peut être accélérée car elle ne dépend que de la bande passante entre les machines. Sur les machines de TP celle-ci s'élève à 100MB/s. De plus il apparait que cette phase est ralentie lors de l'utilisation de cluster plus massif, du fait de la multiplication des commandes scp, ce qui a un impact

négatif sur les performances. De plus, si l'une des machines du cluster dispose d'une moins bonne connexion, ou à d'autres tâches en cours, le temps d'exécution de cette phase est celui de la moins bonne des machines du cluster.

L'utilisation d'une compression décompression des splits peut ici accélérer cette phase en réduisant la taille des fichiers, mais il est visible dans les résultats que le gain sur cette phase ne compense pas la perte durant la phase de création des splits et de map dû à la compression décompression. Toutefois en cas de bande passante plus faible cette option peut s'avérer intéressante.

Phase de map_shuffle :

Dans mon projet ces deux phases sont mélangées afin de limiter les écritures sur disque. La partie la plus longue dans cette étape reste toutefois le shuffle que l'on peut voir sur le fichier de 18Go où l'une des parties représente un de mes systèmes où les deux étapes étaient séparées. Ce qui impacte grandement cette étape est le temps nécessaire à l'échange de tous les fichiers entre les machines du cluster. En effet pour 2Go de fichier en Input, il faudra renvoyer et recevoir en moyenne 2,5Go sous la forme de plusieurs fichiers. L'augmentation de la taille des fichiers provenant du split des lignes dans les fichiers de données. La limitation redevient ici encore la bande passante.

Phase de map_reduce_shuffle :

Comme discuté plus tôt, une des solutions pour limité le temps de la partie shuffle a cette étape a été de réduire la taille des fichiers avant le shuffle avant de les échanger entre les machines, en procédant un premier reduce. On peut voir ici que le gain est très important sur cette étape. Cette étape pour le fichier de 18Go ne prend que 74 secondes contre 213 pour un système map reduce classique et 137 pour un système map_shuffle combinés.

Phase de reduce :

Cette phase est celle la plus proche du système classique, en effet les fichiers renvoyés ne représentent alors plus qu'une taille négligeable et la partie de transfert des données y est négligeable. Sur cette phase on peut voir deux choses intéressantes :

-Sur les petits fichiers (1.9Go) l'augmentation du nombre de machines allonge cette phase, en réalité l'augmentation vient du fait que l'on augmente le nombre de requêtes scp simultané vers le master Node.

-Sur les fichiers moyen (5.7Go), une augmentation du nombre de machine s'accompagne dans un premier temps d'une réduction du temps d'exécution dû au gain sur la partie reduce effectuée par les machines supérieur à la perte de temps dû à la multiplication des transfert. Toutefois si l'augmentation devient trop importante alors le temps s'allonge, le gain de temps sur la phase de reduce effectué par les machines ne compense alors plus la perte du à la multiplication des requêtes.

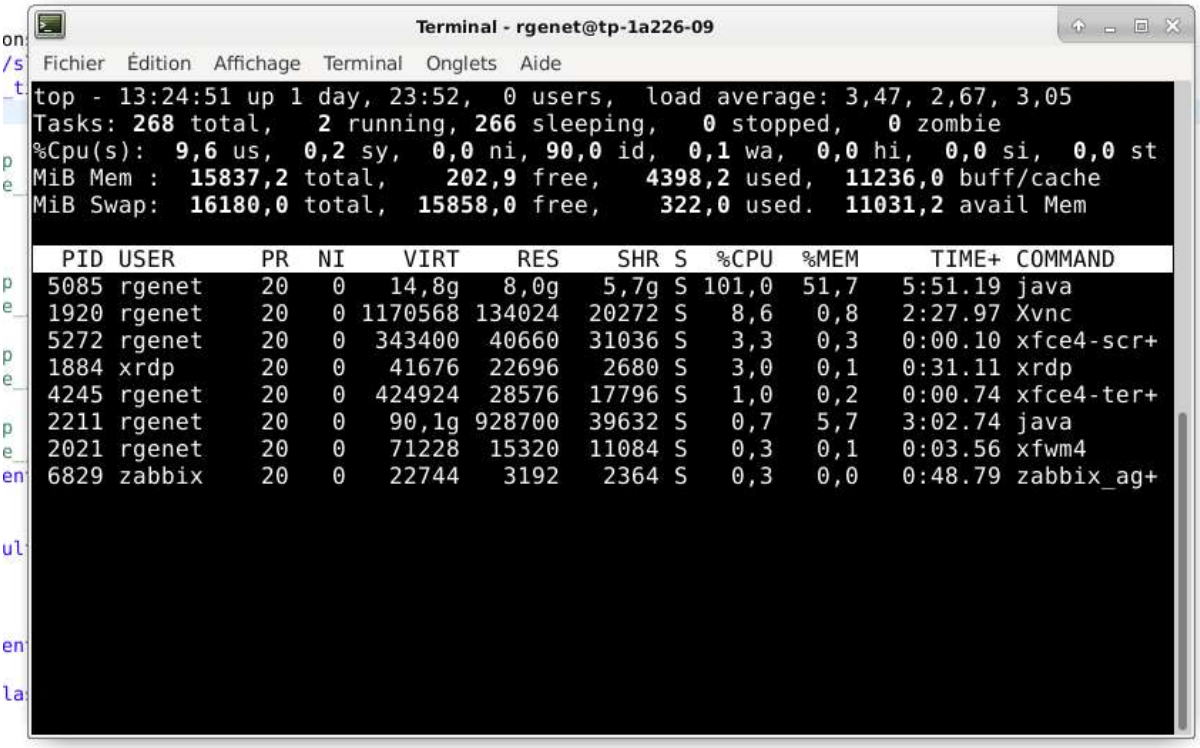
Phase de reduce suivant un map reduce shuffle :

La phase de reduce après un map shuffle reduce permet de voir que celle-ci prend un temps presque fixe indépendamment de la taille du fichier. Le seul paramètre modifiant sa durée étant le nombre de machines présentent dans le cluster, du fait toujours de la multiplication du nombre de commande scp. Cela s'explique très simplement par le fait que l'utilisation d'un pré-reduce dans la phase précédente réduit considérablement la tache dans cette étape qui ne consiste en réalité plus qu'à merger les dictionnaires, un peu comme le fait le master node.

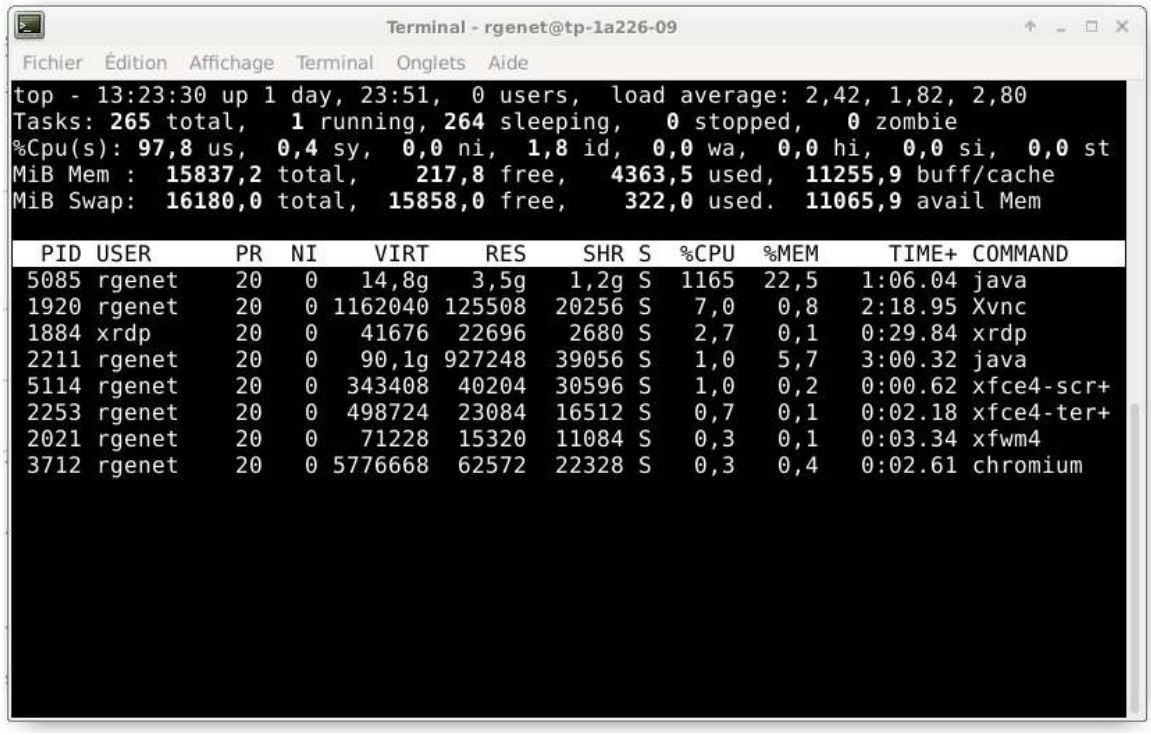
Analyse des résultats sur le système classique

Sur le système classique on peut voir tout d'abord que l'utilisation de Java par rapport à python permet des gains conséquents. Ceci est inhérent à Python et est l'une de ses faiblesses, bien que plus rapide à développer celui-ci se retrouve bien plus limité au niveau des performances. Ce qu'il est aussi intéressant de voir, c'est comment la bonne optimisation d'un système classique peut permettre des gains de temps bien supérieurs à ceux qui peuvent être obtenus en passant à un système MapReduce. L'utilisation du bon objet comme lecteur du fichier permet ainsi de réduire par 5 le temps du processus sur le fichier de 18Go.

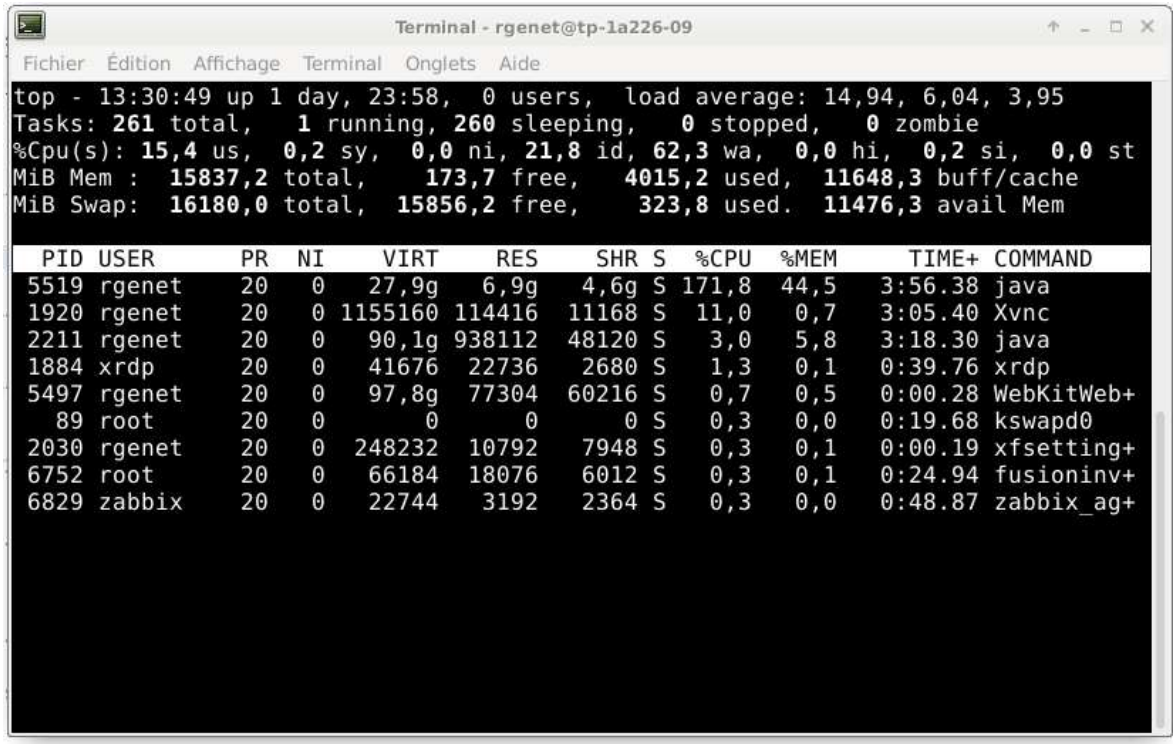
Le passage à un algorithme utilisant le multiprocessing, où là le temps d'optimisation est beaucoup plus important, permet encore de réduire le temps d'exécution. On observe toutefois que le gain est de près de 70% sur un fichier de 2Go par rapport au système séquentiel, mais qu'il n'est plus que d'environ 50% sur le fichier de 18Go. Comment expliquer alors cette différence, le temps de ces compteurs variant peu entre les exécutions, ayant sélectionné des machines sans autres utilisateurs, et n'ayant ici pas le problème d'une augmentation du nombre de commande ssh ? J'ai tout d'abord observé l'utilisation du processeur durant ces taches via la commande « top –i » (ou « htop »). -Pour l'algorithme séquentiel on peut voir que le processeur reste constamment sur les 100% (1 cœur) tout au long de l'exécution, sans accrocs (cf. image ci-dessous) :

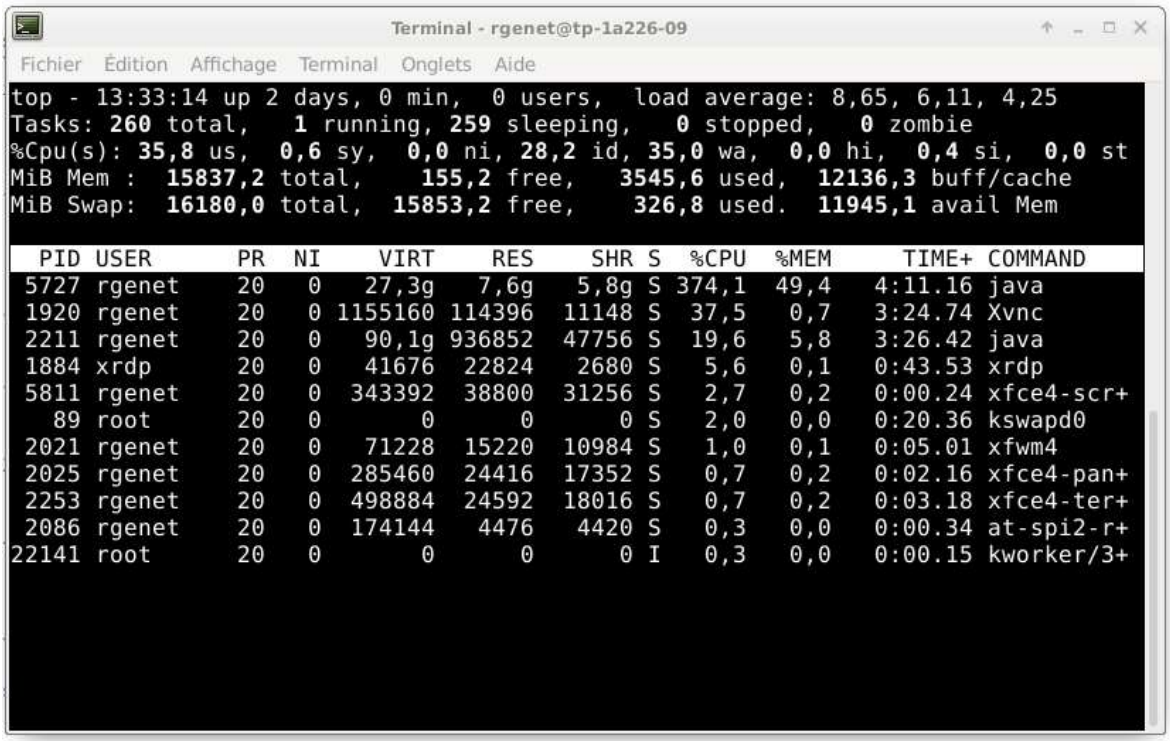


Pour le programme réparti sur plusieurs threads l'histoire est différente. En effet, sur le fichier de 2Go l'utilisation affichée se situe entre 1100% et 1200% du cpu (répartie sur les 12 cœurs, voir image ci-dessous).

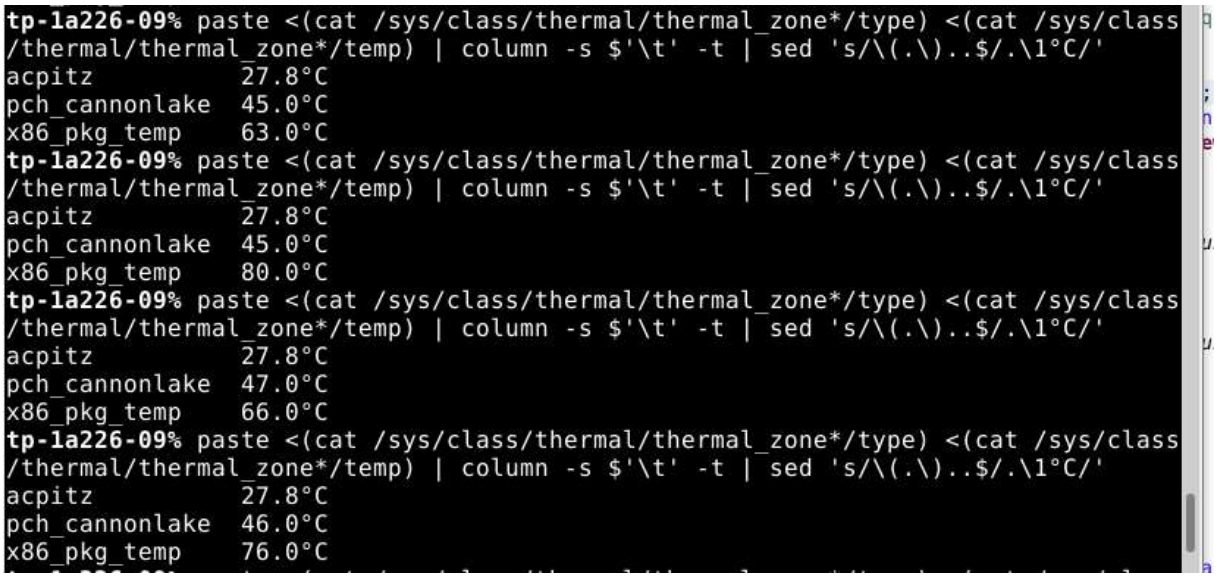


Toutefois dès lors que le fichier devient plus gros et donc l'exécution plus longue, l'utilisation processeur descend entre 150 et 600% par à-coup. (cf. deux images suivantes).

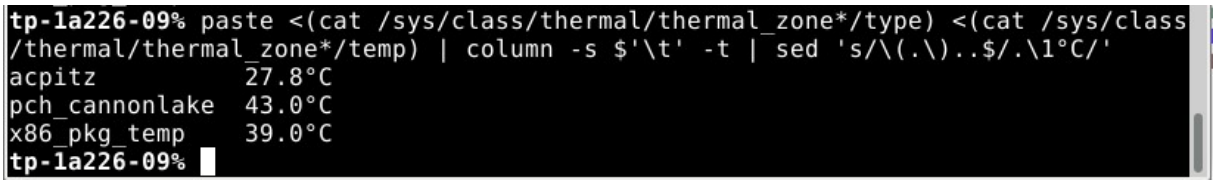




J'ai donc pensé à regarder du côté de la température du processeur via la console. Ainsi, il est apparu qu'alors que le processeur tournait à 37 degrés au repos, que la température montait rapidement du fait de l'utilisation massive du processeur. Là où celle-ci était stable lors de l'utilisation de l'algorithme séquentiel sur un seul cœur, la sécurité thermique s'activait sur le programme réparti sur plusieurs threads et entrainait un thermal throttling. On peut le voir sur les images ci-dessous :



10 : Exemple de thermal throttling



11 : température au repos

Analyse des résultats :

Les résultats nous ont permis de voir que le système MapReduce n'était ici pas en mesure de battre notre système classique. La question qu'il pourrait être intéressant de se poser est : aurait-il pu être en mesure de le battre ? Aurions-nous pu prédire facilement avant qu'il n'allait pas être aussi performant ?

En partant de notre algorithme de référence calculons le temps théorique des étapes en considérant que le temps de réponse à la commande ssh des machines est nul, ce qui est déjà un avantage donné au système réparti. Nous savons que notre bande passante est limitée à 100MB/s. Prenons désormais le fichier de 18Go employé précédemment. Pour l'envoyer la phase de déploiement prendra donc au minimum 180 secondes.

Considérons désormais la phase de map et de reduce. Celle-ci fonctionnant de façon assez similaire à notre algorithme séquentiel on peut estimer leur durée à $\frac{2*x}{n}$ où x est le temps d'exécution du système classique et n le nombre de machine dans le cluster.

A cela il faut ajouter la phase de shuffle, durant laquelle les machines envoient et reçoivent des fichiers dont la somme pour chaque machine (en réception ET en envoie) représente généralement 20% de plus que leur splits. Chaque machine prend donc un temps estimé à $\frac{2*18000}{n*100} = \frac{360}{n}$ secondes.

La phase d'envoie des reduces est considérée négligeable, la taille des fichiers étant considérablement réduite, et le merging des fichiers tout autant.

Enfin reste la phase de splits dont nous noterons s.

L'équation du temps d'exécution est donc : $s + 180 + \frac{360+2*x}{n}$, x étant la durée du système classique. Dans notre cas en prenant l'algorithme séquentiel prenant 365 secondes cela revient donc à chercher s et n tel que $s + \frac{1090}{n} \leq 185 \leftrightarrow s + 1090 \leq 185 * n$

Pour au moins éliminer le 1090 de l'équation il faut que le cluster comporte au moins 6 machines. Si l'on prend la durée du split observée il en faut au moins 8.

Sur un algorithme pas complètement optimisé, et dans un monde hypothétique où les machines seraient équivalentes et répondraient immédiatement aux commandes ssh il serait donc possible de « battre » le système classique séquentiel. Toutefois c'est sans compter l'optimisation en passant au multiprocessing. L'équation deviendrait alors $s + \frac{718}{n} \leq -1$,s et n ne pouvant être négatif l'équation est nécessairement fausse et il est impossible de battre le système classique.

Loi d'Ahmdal :

L'un des objectifs de ce projet était aussi d'essayer d'observer la loi d'Ahmdal.

Cette loi nous dis que $s = \frac{1}{\frac{p}{n}+s}$, ce qui peut aussi se réécrire selon la convenance $speedup = \frac{1-\frac{1}{p}}{1-\frac{1}{s}}$ où s représente l'augmentation des ressources. Il semble donc facile à appliquer dans notre cas pour trouver la partie de notre code parallèle et vérifier cette loi, en comptant le nombre de machines utilisées. Cependant ici bien que les tâches s'exécutent en parallèle sur les différentes machines, la variation des conditions notamment des temps de réponse à la requête ssh rend difficile la comparaison. En effet on peut observer que pour des clusters comprenant un plus grand nombre de machines bien souvent le temps se retrouve augmenté car c'est la machine la plus lente qui bloque le processus.

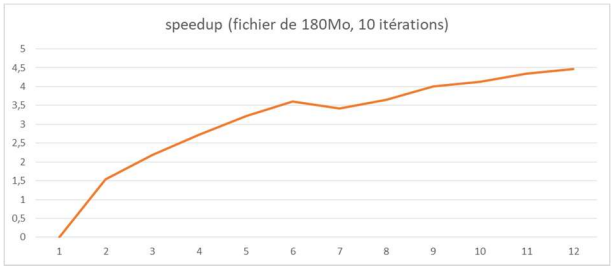
La loi d'Ahmdal peut toutefois être observée sur la partie multiprocessessé du Word count. Bien qu'elle ne soit pas aussi fidèlement vérifiée lors de l'utilisation de fichier lourd car impactée par l'effet du thermal throttling, j'ai ci-dessous reproduit l'expérience pour des nombres de threads variant sur des fichiers plus petits. Les résultats sont les suivants :

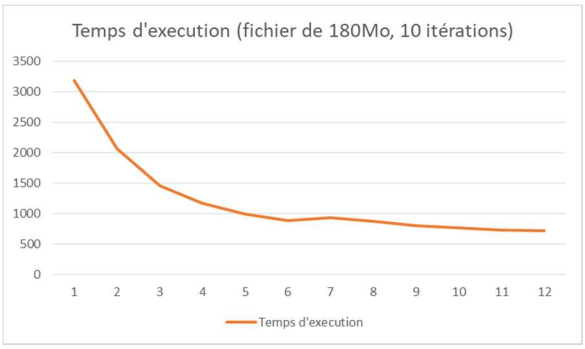
```
utilisation de : 1 threads
time elapsed (ms) : 3674
utilisation de : 2 threads
time elapsed (ms) : 2149
utilisation de : 3 threads
time elapsed (ms) : 1522
utilisation de : 4 threads
time elapsed (ms) : 1172
utilisation de : 5 threads
time elapsed (ms) : 999
utilisation de : 6 threads
time elapsed (ms) : 880
utilisation de : 7 threads
time elapsed (ms) : 951
utilisation de : 8 threads
time elapsed (ms) : 943
utilisation de : 9 threads
time elapsed (ms) : 886
utilisation de : 10 threads
time elapsed (ms) : 817
utilisation de : 11 threads
time elapsed (ms) : 791
utilisation de : 12 threads
time elapsed (ms) : 802
```

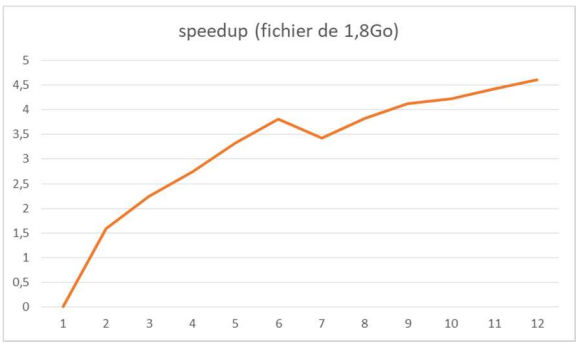
```
utilisation de : 1 threads
time elapsed (ms) : 32479
utilisation de : 2 threads
time elapsed (ms) : 20464
utilisation de : 3 threads
time elapsed (ms) : 14477
utilisation de : 4 threads
time elapsed (ms) : 11825
utilisation de : 5 threads
time elapsed (ms) : 9759
utilisation de : 6 threads
time elapsed (ms) : 8509
utilisation de : 7 threads
time elapsed (ms) : 9467
utilisation de : 8 threads
time elapsed (ms) : 8500
utilisation de : 9 threads
time elapsed (ms) : 7866
utilisation de : 10 threads
time elapsed (ms) : 7683
utilisation de : 11 threads
time elapsed (ms) : 7339
utilisation de : 12 threads
time elapsed (ms) : 7052
```

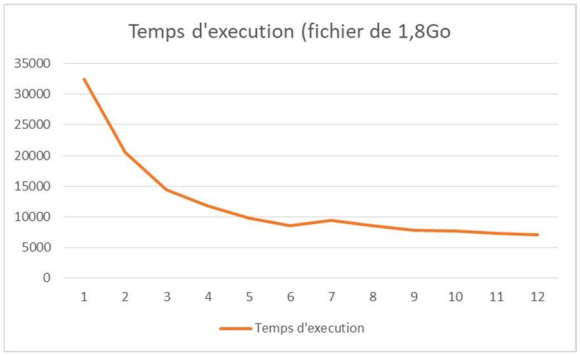
Résultats avec des fichiers de 180Mo et 1,8Go respectivement

Cela nous permet de tracer les graphes suivants :









Fichier de 180Mo	
Nombre de threads	**portion de code parrallèle calculée**
2	71%
3	81%
4	84%
5	86%
6	87%
7	83%
8	83%
9	84%
10	84%
11	85%
12	85%

Fichier de 1,8Go	
Nombre de threads	**portion de code parrallèle calculée**
2	74%
3	83%
4	85%
5	87%
6	89%
7	83%
8	84%
9	85%
10	85%
11	85%
12	85%

On peut tout d'abord voir ici que la courbe du speed up en fonction des ressources ressemble bien à celle étudiée en cours. Par ailleurs on peut voir que peu importe le nombre de cœurs la portion parallèle du code calculée par la loi reste sensiblement la même autour de 85%.

On pourrait ainsi en déduire que sur cet algorithme une augmentation du nombre de cœurs par 2 ne nous permettrait d'obtenir un speed up de seulement 17%.

Conclusion :

Bien que l'idée du système réparti pour augmenter les performances de ces algorithmes puisse paraitre attrayante en imaginant la puissance de calcul à disposition, ce TP met en lumière la nécessité de savoir choisir quand un système réparti est utile ou non. En effet pour une tâche aussi basique qu'un comptage de mots l'appel à un système réparti n'est ni nécessaire, ni efficace au niveau des performances.

Par ailleurs le développement d'un système réparti requiert un temps bien plus long qu'un système classique, temps qui aurait pu être utilisé à optimiser le système classique et qui aurait pu donner de bien meilleurs résultats en terme de gain de performance.

Toutefois le système réparti n'est pas pour autant dénué d'intérêt, et peut s'avérer fort utile. En effet, le problème ici est que la tâche répartie est trop simple, et donc trop rapide à effectuer pour que le temps de transfert des données soit amorti. L'idée est donc de savoir quand un système MapReduce est efficace et quand il ne l'est pas. Un exemple qui me vient ainsi en tête et pour lequel est très applicable est un projet en python réalisé durant mon master 2. Celui-ci effectuait la même opération sur des fichiers de données différents, mais de taille contenue (10Mo maximum). Cependant l'opération effectuée sur chaque fichier était très lourde avec notamment le test de très nombreux paramètres. Au final l'opération prenait une heure par fichier. Les résultats étaient ensuite mergés dans un tableau de résultats. Dans un cas similaire l'utilisation d'un système distribué aurait été fort judicieux. Le temps de transfert aurait été presque nul, et la division du temps d'exécution presque équivalent au nombre de machines dans le cluster.

Pour conclure, ce projet nous a permis d'appréhender les systèmes MapReduce et d'en comprendre leurs limites, mais aussi de visualiser la loi d'Ahmdal. Il m'a aussi permis de découvrir de nombreux objets et optimisations possible en Java.