Hadoop MapReduce from scratch

Télécom Paris | Mastère Spécialisé Big Data | INF727 – Systèmes répartis

---

Introduction

Ce projet a lieu dans le cadre du cours INF727 – Systèmes répartis, dispensé par Rémi Sharrock au sein de Télécom Paris.

Le détail de celui-ci est disponible sur la page personnelle de Mr Sharrock à l'adresse suivante : https://remisharrock.fr/courses/simple-hadoop-mapreduce-from-scratch/

En synthèse ; au travers de plusieurs séances de travaux pratiques guidés, il s'agira d'implémenter une version simple du concept MapReduce (i.e Hadoop). Tout au long de l'implémentation, une attention particulière sera portée sur la justification de la loi d'Ahmdal. Pour cela, des chronométrages seront effectués afin de calculer les différents « speedups » possibles dans les implémentations.

Ce document a pour objectif de synthétiser les méthodes mises en place pour cette implémentation, les choix retenus ainsi que les résultats observés sur certains cas pratiques. Il contient également des réflexions et pistes d'améliorations envisageables pour résoudre les problèmes rencontrés.

Auteur : DELARUE Simon Octobre 2020

Étape 1 – Approche séquentielle

Objectif : Faire un programme **séquentiel non parallélisé** qui compte le nombre d'occurrences des mots dans un fichier

Remarque : Les programmes qui seront présentés ici sont développés en **Python**.

Pour la structure de stockage des nombres d'occurrences de chacun des mots du fichier, j'ai choisi un **dictionnaire<String, Integer>**.
Ce format permet de conserver chaque mot d'un fichier passé en paramètre comme une **clé** du dictionnaire (identifiant unique), et de lui affecter une valeur entière correspondant au total de ses occurrences dans le fichier.

Exemple d'output pour un dictionnaire de comptage de mots :
Car 3
Beer 2
Deer 2
River 2

➔ Pour chaque ligne, le premier élément est un mot du fichier (c'est également une clé unique du dictionnaire) et le second, son nombre d'occurrences dans le fichier

Pour s'assurer que le programme de comptage de mots fonctionne sur des encodages spécifiques, je passe dans la fonction de lecture du fichier un paramètre spécifiant celui-ci (utf8 par exemple).

Le Tableau 1 présente les différents temps de chronométrage pour l'application de la fonction de comptage d'occurrences de mots (+temps de lecture du fichier), ainsi que pour le tri de l'output, pour des fichiers de tailles différentes.

Remarque : La fonction de tri effectue d'abord un tri par valeurs décroissantes, puis par ordre alphabétique des clés si valeurs égales.

FICHIER INPUT	TAILLE FICHIER	5 MOTS LES PLUS FREQUENTS	TEMPS OUVERTURE FICHIER + COMPTAGE DES MOTS (SECONDES)	TEMPS TRI DE L'OUTPUT (SECONDES)	TEMPS TOTAL (SECONDES)
INPUT	43 octets	Car 3 Beer 2 Deer 2 River 2	0.0002	1.5e-5	0.00035
FORESTIER MAYOTTE	1 Ko	de 12 biens 8 ou 8 forestier 6 des 5	0.0007	5.3e-5	0.00123
CODE DE DEONTOLOGIE DE LA POLICE NATIONALE	7 Ko	de 86 la 40 police 29 et 27 à 25	0.0007	0.0004	0,0011

DOMAINE PUBLIC FLUVIAL	71 Ko	de 621 le 373 du 347 la 330 et 266	0.0114	0.0017	0.0131
SANTE PUBLIQUE	18,1 Mo	de 189699 la 74433 des 66705 à 65462 et 60940	1.5222	0.1340	1.6562
ARCHIVE PAGES INTERNET	398,8 Mo	the 578580 to 434203 and 432210 - 382045 of 376385	42.694	16.802	59.496

Tableau 1 Chronomètres du comptage d'occurrences de mots et du tri de l'output pour des fichiers de tailles différentes

Les résultats pour la version séquentielle nous permettent de constater plusieurs choses :

- Jusqu'à une certaine taille de fichier input, les temps de traitements augmentent, mais restent considérablement bas ; entre 1 et 2 secondes pour le temps de traitement total du second plus gros fichier

- Passé une taille critique, les ordres de grandeurs des étapes de comptage de mots et de tri augmentent de manière significative ; temps total de traitement multiplié par **37**, quand la taille du fichier est multipliée par **22**. Le coût du traitement n'est donc plus linéaire en fonction de l'augmentation du volume du fichier.

Etape 2 – Approche distribuée

Objectif : Implémenter une approche de calcul distribuée (sur le format Hadoop MapReduce) pour compter le nombre d'occurrences des mots d'un fichier.

Dans l'approche distribuée, le programme nécessite d'être divisé en plusieurs étapes clefs :

- **Split des données :** le fichier input est splitté en plusieurs sous-fichiers. Chaque split est envoyé sur une machine distante (worker)
- **Map :** Chaque worker mappe les données des fichiers splits qu'elle a reçu, c'est-à-dire ; pour chaque mot rencontré dans le split est associée la valeur 1.
- **Shuffle :** Les workers s'échangent entre eux les fichiers mappés, afin de regrouper sur une même machine les fichiers contenant les mêmes mots.
- **Reduce :** Chaque machine agrège les fichiers qu'elle a reçu des autres workers, en additionnant les occurrences des mots similaires.

Finalement, chacune des machines envoie son résultat à la machine maître, qui agrège le tout afin de donner une vision du nombre d'occurrences des mots dans le fichier.

Choix d'implémentations communs aux différentes versions proposées :

- **Stockage des informations :** Dans les différentes versions ci-dessous, le programme MASTER possède en permanence la vision des exécutions de chaque étape (split, map, shuffle, reduce). Il existe donc au sein du MASTER, un mapping entre les machines SLAVES et les jobs. Cette information est conservée dans un dictionnaire liant les inputs aux machines dont la connexion a **réussi**.

- **Gestion des connexions aux machines du réseau** : La gestion des machines disponibles est faite au travers d'un TimeOut de 10 secondes, qui permet au MASTER de ne pas être trop contraint par les disponibilités du réseau. Ainsi, après la 1ère tentative de connexion sur une liste de machines donnée, le MASTER stocke une liste des machines disponibles, à utiliser pour la suite des calculs.

- **Lecture de l'input et split des données (fonction implémentée « from scratch »):** le split du fichier de données se fait sur la base d'un nombre de caractères à considérer. Cette méthode permet en effet de ne pas tester chaque caractère du fichier (ce qui est le cas d'une lecture ligne à ligne par exemple). Si le split du fichier provoque la coupure d'un mot, on teste chaque caractère du mot en question, jusqu'à trouver un espace, et on module la taille du split en fonction. Ce paramétrage est implémenté au sein du MASTER. Dans la suite, on testera plusieurs tailles de splits pour comparer les performances obtenues en fonction des cas.

- **Envoi des reduce.txt au MASTER :** Les SLAVES utilisent la **sortie standard** pour envoyer (écrire) leurs résultats respectifs au MASTER. Le MASTER lit cette sortie au fur et à mesure des traitements de chacune des machines, et écrit les

données dans un fichier resultat, permettant une sauvegarde (et un élément de comparaison) pour l'utilisateur.

- **Log utilisateur :** Les logs affichées au fur et à mesure de l'exécution permettent à l'utilisateur de suivre le déroulement des étapes. La Figure 1 donne un exemple simpliste d'exécution sur le fichier forestier_mayotte.txt (dans l'exemple, il y a plus de machines que de splits du fichier input initial).

```
simondelarue@MacBook-Pro-de-simon:~/Documents/MS_BGD-Telecom_PARIS/INF727_Systemes_repartis/Hadoop_Map_Reduce_from_scratch$ ./MASTER.py -clean forestier_mayotte.txt -sort
================================================
 CLEANING machines ...
Machine : tp-1a207-15 | CLEANING OK
Machine : tp-1a207-18 | CLEANING OK
Machine : fausse-mach | CLEANING Echec
Machine : tp-1a207-16 | CLEANING OK
Machine : tp-1a207-17 | CLEANING OK
Machine : tp-1a201-29 | CLEANING TimeOut Echec
------------------------------------------------
Connexions réussies : 4
Connexions en échec : 2

================================================
Deploiement du SLAVE.py sur les machines ...
Machine : tp-1a207-15    | Deploiement OK
Machine : tp-1a207-18    | Deploiement OK
Machine : fausse-mach    | Deploiement Echec
Machine : tp-1a207-16    | Deploiement OK
Machine : tp-1a207-17    | Deploiement OK
Machine : tp-1a201-29    | Deploiement TimeOut Echec
------------------------------------------------
Connexions réussies : 4
Connexions en échec : 2
================================================
 SPLITS | Partition des données forestier_mayotte.txt
S1.txt  -> tp-1a207-15
S0.txt  -> tp-1a207-18
S2.txt  -> tp-1a207-16
------------------------------------------------
 SPLITS | Deploiement
Connexion machine : tp-1a207-15 | données S1.txt copiées OK
Connexion machine : tp-1a207-18 | données S0.txt copiées OK
Connexion machine : tp-1a207-16 | données S2.txt copiées OK
------------------------------------------------
 Copie de la liste des machines sur les workers
Machine : tp-1a207-18 | copie available_machines.txt OK
Machine : tp-1a207-16 | copie available_machines.txt OK
Machine : tp-1a207-15 | copie available_machines.txt OK
================================================
 MAP
Machine : tp-1a207-15 | MAP S1.txt OK
Machine : tp-1a207-18 | MAP S0.txt OK
Machine : tp-1a207-16 | MAP S2.txt OK
 MAP finished : 0.73172 secondes
================================================
 SHUFFLE
```

```
Machine : tp-1a207-15 | SHUFFLE UM1.txt OK
Machine : tp-1a207-18 | SHUFFLE UM0.txt OK
Machine : tp-1a207-16 | SHUFFLE UM2.txt OK
 SHUFFLE finished : 1.33275 secondes
================================================
 REDUCE
Machine : tp-1a207-15 | REDUCE OK
Machine : tp-1a207-18 | REDUCE OK
Machine : tp-1a207-16 | REDUCE OK
 REDUCE finished : 0.82547 secondes
================================================
 SORT RESULT (see 'sorted_result.txt')
 SORT finished : 0.01043 secondes
Temps total : 2.90036
```

Figure 1 Exemple de log d'exécution

1ère version – MASTER parallélisé

En synthèse, la 1ère version (naïve) de l'approche distribuée donne des résultats beaucoup moins bons que la version séquentielle (cf Tableau 2), pour les différentes tailles de fichiers testées.

D'après les résultats du Tableau2, on aperçoit d'emblée que c'est la phase de shuffle qui est extrêmement gourmande, du fait de la grande quantité de données à échanger entre les workers.
Dans cette version, le MASTER affecte déjà les jobs aux SLAVES de manière **parallèle**, mais chaque SLAVE exécute ses jobs de manière séquentielle, ce qui n'est pas optimal en terme de temps de traitement.
De plus l'étape SHUFFLE est implémentée sur le format défini dans le projet ; 1 fichier shuffle par *hash* de mot, ce qui implique une grande quantité de fichiers à transférer sur le réseau.

Choix d'implémentation de la 1ère version :

- **Parallélisation des tâches du MASTER** :
  Le programme maître conserve l'association entre chaque split de données et machine dans un dictionnaire. Grâce à cette structure, il lui est possible de d'envoyer des données ou des jobs à effectuer aux machines SLAVE, de manière **parallélisée**. Étant donné l'indépendance des traitements entre chaque machine, ce choix d'implémentation est naturel pour optimiser le temps de calcul :
  - Déploiement des splits de données en parallèle
  - Déploiement des fichiers de machines valides en parallèle
  - Envoi des demandes de **MAP** aux slaves en parallèle
  - Envoi des demandes de **SHUFFLE** aux slaves en parallèle
  - Envoi des demandes de **REDUCE** aux slaves en parallèle

- **Pas de parallélisation des tâches du SLAVE :** Aucune tâche n'est parallélisée au sein de chaque SLAVE à ce stade.

- La création des fichiers shuffles est implémentée sur la base des instructions du projet ; un fichier nommé par le résultat du hashage d'un mot et de la machine ayant effectué le job. Dans la suite, cette étape sera optimisée afin de réduire le nombre de fichiers à transférer sur le réseau.

- La création du répertoire destiné à la réception des fichiers shuffles sur chaque machine (*shufflesreceived/*) est faite de manière séquentielle au moment de l'étape de shuffle d'une machine. Une optimisation de cette étape sera effectuée dans les versions suivantes, afin de réduire le nombre de demandes de connexions total entre machines du réseau.

- La fonction **REDUCE** n'est pas optimisée pour les fichiers de grande taille, en effet, on effectue à ce stade une première étape de comptage des mots de chaque fichier *shuffle* reçu, puis on alimente un dictionnaire de la forme <valeur de hash, [mot, occurences]>, qui permet le comptage des mots et qui sera parcouru pour le transfert des résultats au MASTER.

Les résultats ci-dessous sont obtenus en travaillant avec 4 machines distantes.

FICHIER INPUT	TAILLE FICHIER	MAP (SECONDES)	SHUFFLE (SECONDES	REDUCE (SECONDES	TEMPS TRI DE L'OUTPUT (SECONDES)	TEMPS TOTAL (SEDONDES)	*VERSION SEQUENTIELLE (SECONDES)*
INPUT	43 octets	0.83827	2.91107	0.63885	0.00396	4.39215	0.00035
FORESTIER MAYOTTE	1 Ko	0.93439	30.76825	0.52008	0.01806	32.24078	0.00123
CODE DE DEONTOLOGIE DE LA POLICE NATIONALE	7 Ko	4.59431	162.50963	0.53338	0.07868	167.71590	0,0011
DOMAINE PUBLIC FLUVIAL	71 Ko	15.58748	1603.68281	0.52895	0.24589	1620.04513	0.0131
SANTE PUBLIQUE	18,1 Mo	-	-	-	-	-	1.6562
ARCHIVE PAGES INTERNET	398,8 Mo	-	-	-	-	-	59.496

Tableau 2 Chronomètres des étapes de calcul réparti – Version non optimisée

Note : les calculs ci-dessous ont été effectués sur ma machine personnelle (performance moindres vs machines de l'école). Néanmoins, la qualité des résultats laisse supposer que des optimisations sont nécessaires avant d'homogénéiser les environnements de traitement pour comparaison.

Au-delà des optimisations à implémenter pour que le programme supporte des fichiers plus volumineux que les petits exemples, il ressort de ces premiers résultats que l'étape de shuffle est le frein majeur en terme de temps de traitement pour un framework de ce type.

2ème version – Réduction du nombre d'interactions sur le réseau

La seconde version du programme permet une accélération considérable des temps de traitement, notamment de la phase de Shuffle. Néanmoins, les résultats restent très en dessous de ceux observés pour l'implémentation séquentielle (voir Tableau 3), et les performances sont toujours contraintes par l'étape de communication des workers sur le réseau.

Nous avons vu grâce à la version naïve que les communications sur le réseau posaient les limites de la performance du système. Les modifications dans les implémentations de la V2 ont donc pour objectif principal d'alléger cette étape. Ci-dessous les optimisations déployées :

- **Réduction du nombre de connexion SSH** sur les machines distantes grâce à une centralisation de la création de répertoires distants lors de l'appel du programme **DEPLOY**. Cette manœuvre très simple permet en effet d'utiliser la connexion ssh initiale (déploiement des slaves) pour créer tous les répertoires utiles sur les machines distantes *(auparavant, un worker envoyait ses fichiers shuffle à ses voisins, et créait via une seconde commande le répertoire pour accueillir ces fichiers si nécessaire)*.

- Les données lues dans les fichiers UM.txt (en sortie du MAP) ne sont plus enregistrées dans des fichiers de shuffle indexés par le résultat de hashage du mot, mais par le **nom de la machine qui recevra les mots** (on conserve néanmoins le suffixe donnant le nom de la machine qui a effectué la tâche). Cela permet de **réduire nettement le nombre de fichiers à transférer** dans le réseau des workers, et ainsi allège la phase de shuffle. En effet, on passe d'un nombre de fichiers à transférer égal au nombre de mots, à un nombre de fichiers au maximum égal au nombre de machines du réseau. Dans ce cas, chaque fichier à transférer est plus volumineux, puisqu'il contient tous les mots destinés à une machine donnée.

- L'écriture dans les fichiers shuffle se fait **sans étape intermédiaire** après la lecture des données dans les fichiers UM.txt (pas de stockage dans un dictionnaire). Cela permet de réduire le temps de traitement.

- **Parallélisation de l'envoi des SHUFFLES au sein d'un SLAVE:**

    - L'envoi des fichiers suffle via scp se fait **de manière parallèle** sur un même worker, grâce à la liste des fichiers créés et indexés selon la machine vers laquelle transférer *(auparavant, le nœud worker balayait et envoyait les fichiers sur le réseau de manière séquentielle).*

- Etape **REDUCE** ; les données réduites de chaque worker sont stockées dans un dictionnaire puis **affichées dans la sortie standard** (sans écriture dans un fichier au préalable).

FICHIER INPUT	TAILLE FICHIER	MAP (SECONDES)	SHUFFLE (SECONDES	REDUCE (SECONDES	TEMPS TRI DE L'OUTPUT (SECONDES)	TEMPS TOTAL (SEDONDES)	*VERSION SEQUENTIELLE (SECONDES)*
INPUT	43 octets	0.63504	1.15561	0.63256	0.0036	2.42688	0.00035
FORESTIER MAYOTTE	1 Ko	0.72476	1.45448	0.73835	0.01662	2.93421	0.00123
CODE DE DEONTOLOGIE DE LA POLICE NATIONALE	7 Ko	0.93301	1.25536	0.62292	0.07107	2.88235	0,0011
DOMAINE PUBLIC FLUVIAL	71 Ko	1.64744	2.67393	0.62469	0.22892	5.17499	0.0131
SANTE PUBLIQUE *	18,1 Mo	1.11582	**24.78976**	0.81453	0.74663	**27.46674**	1.6562
ARCHIVE PAGES INTERNET *	398,8 Mo	5,18943	**95,07498**	3,98440	**32,00447**	**136,25329**	59.496

Tableau 3 Chronomètres des étapes de calcul réparti – Version 2

(*) Taille de split : 8Mo, nombre de machines : 21

La faible performance (en comparaison de la version séquentielle) de l'étape de tri saute aux yeux. En effet, dans cette version, les SLAVES envoient les données de l'étape *reduce* au MASTER, au travers de la sortie standard. Puis, dans un second temps, le MASTER compile ces sorties dans un dictionnaire qui sera trié afin d'obtenir le résultat au format souhaité.
En revanche, dans la version séquentielle, le dictionnaire est construit **au cours** des étapes qui précèdent la phase finale, ce qui est un gain de temps au moment de l'agrégation. En décomposant et chronométrant les étapes de tri de la version séquentielle, nous avons les résultats suivant :

- Création de dictionnaire ~ 16s
- Tri du dictionnaire ~ 16s

Qui sont cohérents avec les résultats constatés pour le tri de la V2, qui comprend ces 2 phases ; **32 secondes**.

On constate également que l'étape de **SHUFFLE** est toujours très gourmande, même si l'écart de temps d'exécution avec la version séquentielle diminue avec l'augmentation de la taille du fichier d'input :

- **sante_publique.txt** : l'étape de shuffle représente **90% du temps d'exécution total.** Le temps d'exécution du comptage de mots est **17 fois supérieur** à celui de la version séquentielle

- **Archives pages internet :** l'étape de shuffle représente **70% du temps d'exécution** (mais 91% du temps total hors phase de tri). Le temps d'exécution total pour le comptage des mots est **2,3 fois supérieur** à celui de la version séquentielle.

On a donc une performance croissante du système distribué en fonction de l'augmentation de la taille du fichier input. On constate de plus en plus que les

bénéfices d'un tel système (i.e répartit), ne sont visibles qu'à partir une taille minimale des données initiales.

3ème version – Parallélisation totale du SLAVE

La troisième version du programme permet une **nette réduction du temps de l'étape SHUFFLE** (voir Tableau 4), grâce à l'introduction de calcul parallèle au sein de celle-ci. La version précédente nous avait déjà permis de nous affranchir d'une certaine volumétrie de communications sur le réseau, nous nous concentrons donc ici sur l'optimisation de la construction des fichiers à transférer.
Cette nouvelle implémentation permet notamment d'avoir un temps de traitement **en dessous de celui pour la version séquentielle, et ce pour le fichier le plus volumineux** (en ne prenant en compte que les étapes MAP-SHUFFLE-REDUCE-SORT, c'est-à-dire sans considérer les étapes de cleaning des machines et de déploiement des splits de données).

Les optimisations apportées au code sont les suivantes :

- Modification de la structure des fichiers en sortie de la phase de MAP : un fichier UM.txt ne contient plus une colonne de mots (avec répétitions) et une colonne de '1', mais seulement une liste des mots rencontrés dans le split de données initial.

  Exemple du contenu d'un fichier UM.txt (ancienne version vs nouvelle version) :

  UM.txt Versions précédentes

  Car 1
  Beer 1
  Car 1
  River 1

  → **UM.txt Version V3**

  Car Beer Car River

  Cette modification dans la structure de l'output du MAP permet de simplifier la lecture de l'input dans la phase de SHUFFLE.
  En effet, plutôt que de lire un fichier ligne à ligne, de convertir chaque ligne en liste de chaînes de caractères et de retenir le 1er élément de cette liste, on appelle la fonction *read().split()* sur tout le fichier. Cette fonction renvoie une liste de mots (avec répétitions).

- **Parallélisation de l'étape SHUFFLE au sein d'un SLAVE :**

  La modification de structure du fichier en sortie du MAP a permis de créer une liste de mots en une seule étape, au début de la phase de SHUFFLE. Il est alors possible de **paralléliser les traitements** à effectuer sur chacun des mots pour créer le fichier de shuffle. La parallélisation porte sur les éléments suivants :

- Attribuer une machine receveuse pour le fichier de shuffle, en appliquant une fonction de hashage au mot (i.e modulo le nombre de machines). C**'est la parallélisation de cette étape qui permet un gain de temps considérable.**

- Écrire le mot dans un fichier shuffle.txt, nommé avec le nom de la machine de destination et le nom de la machine ayant effectué le calcul.

FICHIER INPUT	TAILLE FICHIER	MAP (SECONDES)	SHUFFLE (SECONDES	REDUCE (SECONDES	TEMPS TRI DE L'OUTPUT (SECONDES)	TEMPS TOTAL (SEDONDES)	*VERSION SEQUENTIELLE (SECONDES)*
INPUT	43 octets	0,43425	1,27117	0,83779	0,00063	2,54384	0.00035
FORESTIER MAYOTTE	1 Ko	0,43440	1,67430	0,83690	0,00119	2,94680	0.00123
CODE DE DEONTOLOGIE DE LA POLICE NATIONALE	7 Ko	0,43622	1,57197	0,84323	0,00333	2,85475	0,0011
DOMAINE PUBLIC FLUVIAL	71 Ko	0,43422	1,77639	0,83719	0,00938	3,05448	0.0131
SANTE PUBLIQUE	18,1 Mo	0,94203	**4,70416**	0,74538	0,19033	**6,78190**	1.6562
ARCHIVE PAGES INTERNET	398,8 Mo	3,48614	**15,80059**	3,39174	30,04959	**52,72806**	**59.496**

Tableau 4 Chronomètres des étapes de calcul réparti – Version 3

Taille de split : 8Mo, nombre de machines : 21

Les différentes optimisations, ainsi que la parallélisation des tâches du MASTER et des SLAVEs ont permis de réduire le temps de traitement de la version distribuée, et de passer sous le chronomètre du temps séquentiel.

Il est toutefois important de noter que dans le tableau ci-dessus, les temps de cleaning des machines et de déploiement des données ne sont pas pris en compte. Ceux-ci peuvent varier en fonction des disponibilités du réseau de l'école (un TimeOut de 10s est paramétré afin de ne pas rester bloqué si une machine ne répond pas), et de la taille du fichier de données à splitter. La somme de ces temps se situe en moyenne autour de 15s (une optimisation considérable peut être d'utiliser la fonction *split* de Linux, beaucoup plus efficace que la version « from scratch » développée pour ce projet).

Comme pour la version V2, on note que l'étape de SHUFFLE constitue toujours un frein majeur dans l'optimisation des temps de traitement. Néanmoins, l'étape de tri de l'output devient la plus consommatrice de temps de traitement pour le fichier input le plus volumineux *(ceci est directement du aux choix d'implémentation de la V3, ou il est nécessaire de reconstruire un dictionnaire des données de sorties avant de le trier)* :

- **sante_publique.txt** : l'étape de shuffle représente **70% du temps d'exécution.** Le temps d'exécution du comptage de mots est **4,1 fois supérieur** à celui de la version séquentielle

- **Archives pages internet :** l'étape de shuffle représente **30% du temps d'exécution** (mais 70% du temps total hors phase de tri). Le temps d'exécution total pour le comptage des mots est **11% inférieur** à celui de la version séquentielle.

On note à nouveau que les performances du système répartit augmentent nettement à mesure que la masse des données en input augmente, ce qui est principalement dû à la diminution de la part du shuffle dans le temps total.

Résultats en fonction des paramétrages

Nous avons vu le coût important de la phase de communications des nœuds du cluster sur le réseau, et nous avons remarqué que pour obtenir des performances intéressantes (vs une version séquentielle), il est nécessaire que le jeu de données ait une certaine taille minimale.

Nous pouvons à présent nous intéresser aux paramètres de notre système ; la **taille des splits de données** et **le nombre de machines dans le réseau**. En effet, ces deux valeurs jouent un rôle direct dans le volume d'échanges entre machines, et il est primordial d'étudier leurs impacts pour optimiser les performances finales.

On peut supposer qu'une taille de split trop petite aura tendance à créer un goulot d'étranglement sur le réseau, au niveau de l'étape de shuffle ; trop de fichiers à transférer pour le volume d'information contenu dans chaque.
On peut également supposer qu'un grand nombre de machines sur le réseau, va permettre un grand nom de calculs en parallèle, mais le gain d'optimisation pour l'ajout d'une machine est-il toujours significatif ? Ou encore, n'existe-t-il pas un effet délétère du fait de la masse de données à communiquer une fois un certain nombre de machines dépassé ?

La Figure 2 (ci-dessous) montre les résultats obtenus pour des exécutions de la version V3 en fonction du nombre de machines et de la taille des splits de données. Les chronomètres sont effectués sur les étapes de MAP – SHUFFLE – REDUCE ainsi que MAP – SHUFFLE – REDUCE – SORT.

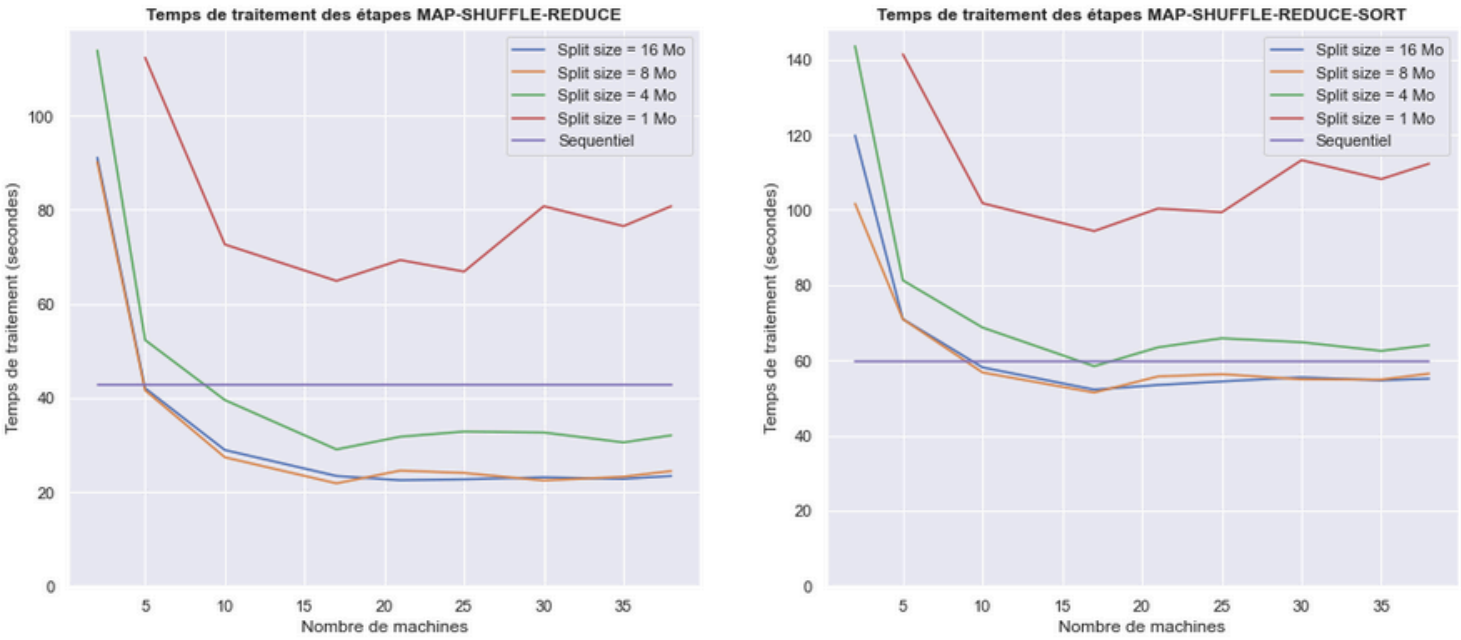

Figure 2 Temps de traitements en fonction du nombre de machines et de la taille des split de données

La figure de droite (y compris phase de tri de l'output) montre que la version calcul distribué dépasse la performance de la version séquentielle lorsque le **nombre de machines dépasse 10**, pour une taille de splits de 8 ou 16Mo.

Si on diminue la taille de splits en dessous de 4Mo, les performances sont nettement plus faibles. C'est le reflet du goulot d'étranglement au niveau des transferts de fichiers.

De manière générale, on remarque que les performances s'améliorent lorsque le nombre de machines croît. Cependant, au-delà de 10-15 machines, les performances atteignent un plateau (voire diminuent légèrement).
Pour les tailles de splits élevées (4 à 16 Mo), cela s'explique par le fait que le nombre de splits est plus petit que le nombre de machines disponibles, donc ajouter des workers supplémentaires n'apporte rien en termes de performances.
Pour des tailles de splits plus faibles (1Mo), cela met en évidence le trade-off important entre nombre de fichiers à transférer sur le réseau / répartition des jobs.

La figure de gauche (hors phase de tri de l'output) met en évidence les performances de la version répartie par rapport à la version séquentielle.
On note que pour des tailles de splits entre 8 et 16 Mo, et au-delà de 15 machines, les temps de traitement des étapes MAP – SHUFFLE – REDUCE sont 2 fois moins élevés que ceux de la version séquentielle.
Comme pour les résultats avec le tri de l'output, les temps de traitement atteignent un plateau à partir d'un certain nombre de machines, et le gain de performances pour l'ajout de ressources supplémentaires n'est pas significatif (voire légèrement contre-productif).

Conclusion

Nous avons vu que l'implémentation d'un système de calcul répartis (simpliste) permet d'atteindre les performances d'une version séquentielle, à conditions que :

- L'input ait une taille minimale
- Le nombre de fichiers à transférer sur le réseau n'excède pas un certain nombre (conditions sur la taille des splits et nombre de machines dans le réseau)

Nous avons également constaté qu'à partir d'un certain point, l'ajout de ressources (machines) n'apporte pas de gain dans le temps de traitement global.

Au-delà de ces résultats, nous avons vu que dans une implémentation telle que celle présentée, un frein majeur se situe au niveau du transferts des données sur le réseau. Comparés aux étapes de MAP et REDUCE, la phase d'échange entre machines, SHUFFLE, est considérable et accapare la grande majorité du temps de traitement.

La phase de tri des données, immédiate dans la version séquentielle, pose un problème plus important dans la version distribuée, puisque l'entité qui agrège les données à l'obligation de stocker la totalité des éléments reçus, puis de les trier, afin d'avoir un résultat cohérent.

Finalement, cette implémentation nécessite plusieurs **points de synchronisation**, ralentissant l'avancement global du traitement. Les principaux points sont les suivants :

- Après la phase de **MAP** : le MASTER attend que tous les SLAVEs aient fini cette étape avant de lancer la suivante

- Après la phase de **SHUFFLE** : le MASTER attend que tous les SLAVEs aient fini cette étape avant de lancer la suivante

- Après la phase de **REDUCE** : le MASTER attend que tous les SLAVEs aient fini cette étape avant de lancer la suivante

Pour aller plus loin

Différentes optimisations sont donc envisageables, afin d'améliorer la qualité globale du système, et d'aller au-delà des limites qu'il impose :

- **Gestion des pannes** : un système performant se doit d'être tolérant aux pannes. En effet, la probabilité qu'une machine soit défaillante (cause « naturelle » ou attaque extérieure) augmente significativement lorsque la taille du réseau croît *(pour un projet de petite envergure comme celui-ci*, *j'ai déjà pu constater des erreurs machines lors des différents essais sur le réseau de l'école).*

Pour cela, les splits de données pourraient être dupliqués sur les différentes machines du réseau. Le job est affecté à la première machine envoyant un signe de vie au MASTER. En cas de panne de cette machine, aucun signe de vie n'est transmis au MASTER, et ce dernier peut affecter le job à la machine suivante sur la liste des machines possédant les données nécessaires.

- **Stream processing** : les limites du batch processing peuvent être contournées via l'implémentation d'un **stream** de données. Chaque SLAVE reçoit les données au fur et à mesure qu'elles sont lues par le MASTER (en fonction de la disponibilité de la machine SLAVE). Ceux-ci effectuent les étapes de MAP, SHUFLLE et REDUCE au fur et à mesure des données reçues. Ainisi, l'output est une représentation à l'instant *T* du résultat global. Cette méthode permet de supprimer les points de synchronisation et ainsi d'éviter d'attendre que la dernière machine ait fini son traitement. Dans ce cas, l'utilisateur à un accès permanent à la donnée, même si des nœuds sont à l'arrêt. Le résultat auquel il a accès est en permanence mis à jour en fonction des disponibilités du réseau. La disponibilité et la tolérance aux pannes priment sur la consistance des résultats.