INF727
Systèmes Répartis pour le Big Data

Nicolas Gallay

-

TELECOM Paris

Table des matières

Introduction :

La croissance exponentielle du volume de données disponibles surpasse aujourd'hui les capacités de certaines machines permettant de les exploiter. Cette évolution rapide a obligé les entreprises de ce secteur à s'adapter et à repenser la façon de traiter ces données (stockage, lecture, calculs…).

Une partie de la réponse à ce problème a été apporté par Google par l'intermédiaire de leur modèle de programmation « MapReduce » permettant la parallélisation et la distribution sur plusieurs machines des calculs sur des données volumineuses.

Cependant, cette parallélisation et cette mise en commun des ressources ne résout pas tout. Le gain maximal atteignable par une parallélisation des tâches est limité par la partie non parallélisable de l'algorithme. C'est ce qui est visible notamment grâce à la loi d'Amdahl utilisée pour prédire l'accélération théorique en fonction du nombre de processeurs.

D'autres facteurs peuvent rentrer en jeux, distribuer les tâches implique des échanges entres les machines d'un même cluster. Ces communications ont aussi un impact sur le temps d'exécution total comme nous le verrons dans plus tard dans ce rapport.

Objectifs :

Le but de ce projet est d'implémenter une version simple du concept de MapReduce permettant de compter l'occurrence des mots d'un texte en utilisant le langage Python et de comparer les performances de notre programme distribué avec un programme séquentiel non parallélisé.

Est-il toujours plus avantageux de paralléliser les calculs ? La loi d'Amdahl est-elle visualisable dans notre cas ? Quels paramètres influencent la performance du modèle ? Quels sont les améliorations possibles ?

Ce rapport a pour but d'expliquer cette implémentation, les problèmes rencontrés, les solutions trouvées et d'essayer de répondre à ces questions.

Le code et disponible sur GitHub à l'adresse suivante :
https://github.com/nicolasgallay/INF727_projet.git

Implémentation de MapReduce :

En partant de notre programme séquentiel non parallélisé permettant de compter les occurrences des mots, il est impossible d'arriver à une version pouvant être distribuée sur différentes machines. C'est là que le modèle 'MapReduce' est utile.

MapReduce va permettre de paralléliser certaines étapes de notre calcul et donc de pouvoir les exécuter sur des machines différentes. Tout ceci a pour but de diminuer le volume de données traitées par chaque machine et d'augmenter la capacité de calcul totale à notre disposition.

Notre programme sera divisé en plusieurs parties selon le schéma suivant :

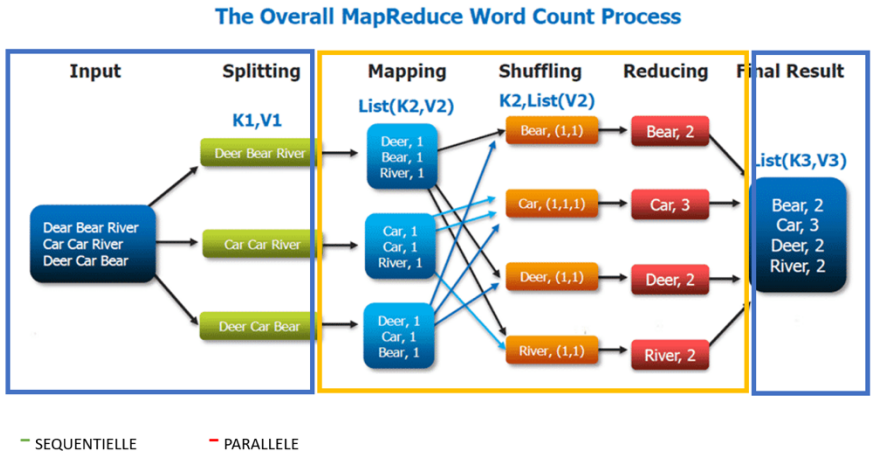

Figure 1 : Étape du MapReduce pour le Word Count

Source : https://www.lebigdata.fr/mapreduce-tout-savoir

Le calcul est lancé par une machine dite « Master » qui possède le fichier d'entrée. Ce Master s'occupe de découper le fichier d'entrée en autant de fichiers que de machines disponibles pour le calcul (appelées « Workers »). C'est aussi cette machine qui récupérera et agrègera les résultats des Workers.

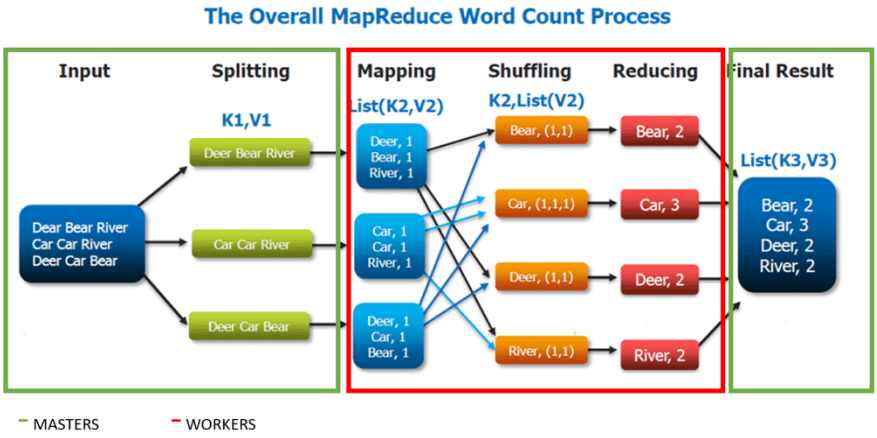

Figure 2 : Répartition des étapes entre le Master et les Workers

Voici une explication rapide de chaque fonction utilisée par le Master et les Workers. Ces fonctions représentent les différentes étapes de notre MapReduce vu sur le schéma ci-dessus. Nous verrons ensuite laquelle de fonctions demandes le plus de temps pour être exécutées., c'est sur ces étapes que nous chercherons des améliorations.

MASTER :

Les fonctions du Master correspondent à la partie verte du schéma.

- ***split_equal*** **:** Cette fonction prend en entrée un fichier texte qui sera ensuite divisé en plusieurs fichiers de plus petites tailles. L'envoie des « Splits » se fait en parallèle sur les Workers.
- ***reiceived_reduceFiles*** **:** Une fois le calcul effectué sur tous les Workers, le Master récupère des fichiers contenant l'occurrence des mots de chaque splits et les agrège par ordre décroissant dans un unique fichier de sortie.

WORKERS :

Les fonctions du Workers correspondent à la partie rouge du schéma. Ces fonctions sont distribuées sur plusieurs machines en même temps et peuvent donc être exécutées en parallèle. Chaque fonction à cependant besoins du résultat de l'étape précédentes parfois d'une autre machine pour pouvoir s'exécuter, Cette gestion se fait à la fonction ***communicate() et scp().***

- ***map_function*** **:** Chaque Worker se retrouve avec un fichier « Split » correspondant à un bout de fichier texte. La fonction « map_function » va s'occuper de séparer chaque mot et de les réécrire dans un fichier avec un mot par ligne.
- ***shuffle_function*** : La fonction shuffle va ensuite récupérer le fichier de Map créé par la fonction précédente et calculer un code de Hachage pour chaque mot. Ce code sera identique pour les mots similaires. Cette fonction gère ensuite l'envoie et la répartition des mots aux différents Workers. Les mots identiques sont envoyés au même Workers afin d'être comptés.
- ***reduce_function*** : Chaque Worker récupère ensuite les fichiers contenant des listes de mots identiques. Elle s'occupe de compter le nombre d'occurrence de ces mots et d'en écrire le résultat dans un fichier qui sera ensuite envoyé au Master.

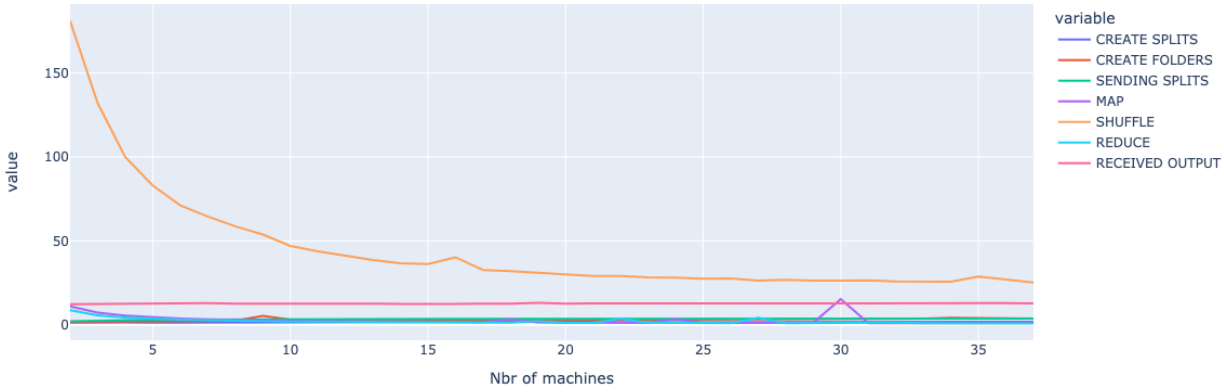

Figure 3 : Temps d'exécution de chaque étape

Le graphe ci-dessus représente le temps de chaque étape de notre MapReduce en fonction du nombre de machines utilisées pour paralléliser notre calcul. Il apparait clairement que l'étape la plus chronophage est l'étape de « Shuffle ». Diviser et distribuer cette étape est donc très utile et permet de diminuer de 86 % le temps d'exécution passant de 181s à 26s.

Loi d'Amdahl :

Nous allons maintenant essayer de visualiser la loi d'Amdahl. Pour rappel cette loi donne l'accélération théorique que l'on peut attendre d'un système dont on améliore les ressources. L'une des conclusions de cette loi est que l'accélération dépend du taux de parallélisation du programme et que cette accélération ne peut pas dépasser un certain seuil dépendant de ce taux de parralélisation.

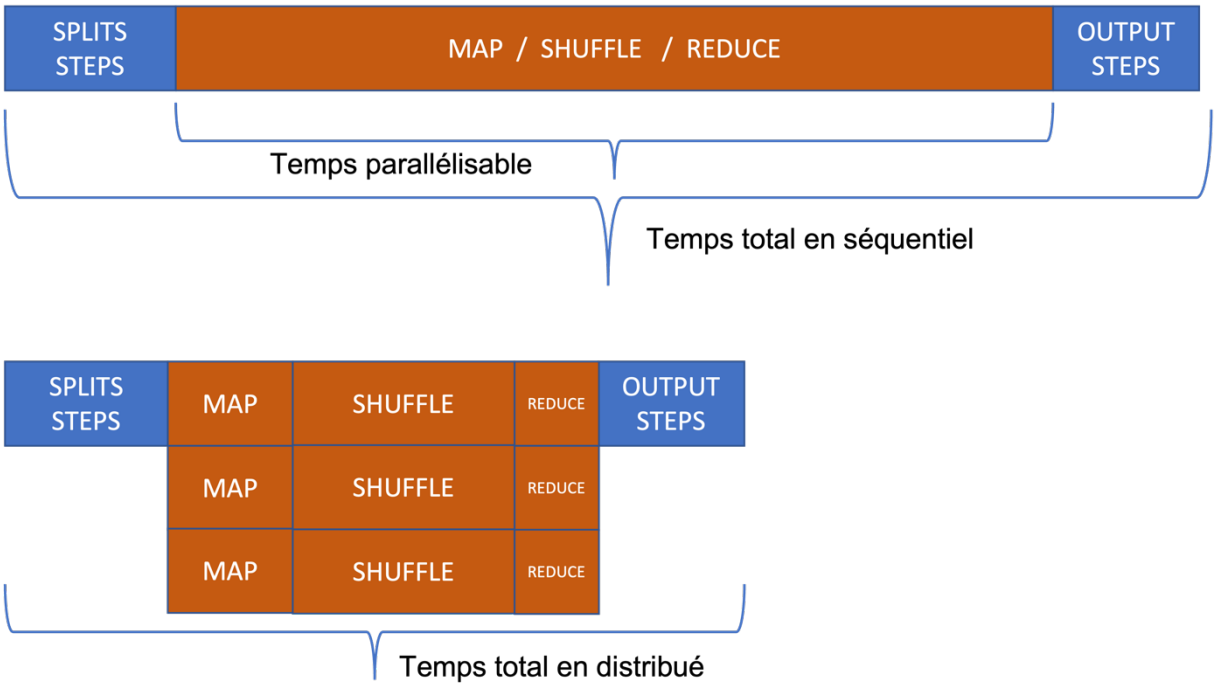

Figure 4 : Schéma de la parallélisation du programme

Dans notre cas il est difficile de définir le taux de parralélisation de notre programme. Nous testons notre programme sur une seule machine et utilisons le temps d'exécution comme référence. Les temps moyens obtenus sont les suivants :

TOTAL TIME	FOLDERS CREATION and SPLITS STEPS	SSH MAP	SSH SHUFFFLE	SSH REDUCE	CREATING OUTPUT
329.989767	14.307583	28.491416	264.248877	12.999258	8.249895

Nous parallélisons ensuite les étapes encadrées en orange sur la figure n°1. En prenant comme référence les temps obtenus au test précédant nous pouvons en déduire qu'il serait possible au maximum de paralléliser 92% de notre programme.

En effectuant maintenant le calcul en mode distribué et parallélisé et en augmentant le nombre de machine. En moyennant les résultats de 5 tests, nous obtenons les résultats suivants.

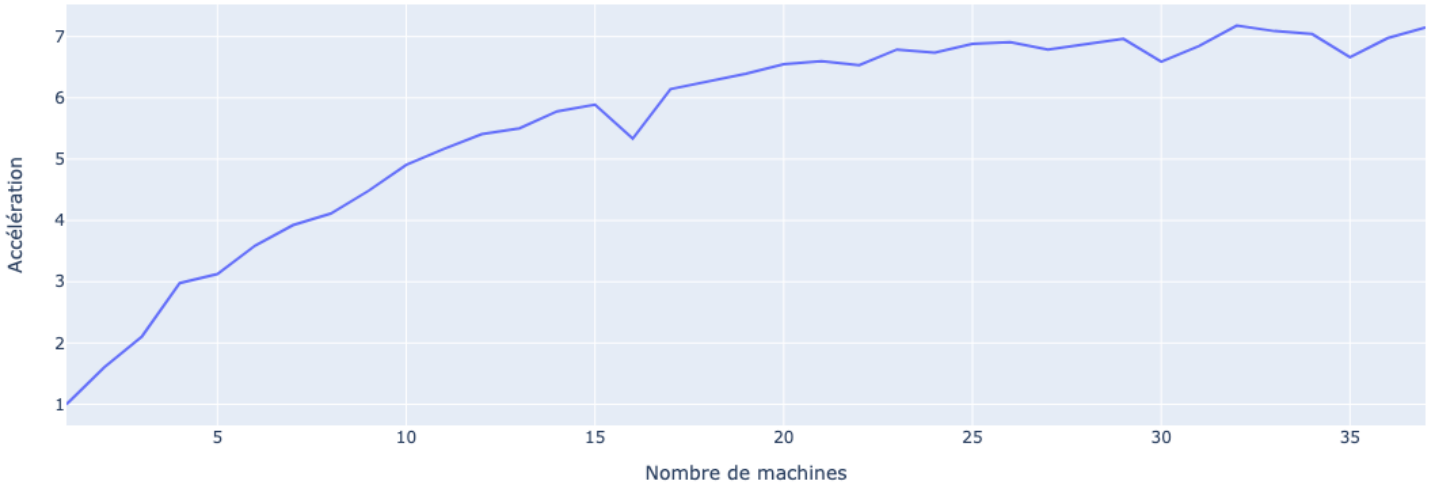

Figure 5 : Accélération observée en fonction du nombre de machines

L'accélération moyenne semble se stabiliser au tour de 7. Il faudrait cependant pouvoir les tester sur beaucoup plus de machine afin d'être certains de ce seuil. Comme énoncé dans la loi d'Amdahl, ce résultat met donc en évidence une limite à la parallélisation de notre programme.

Cependant, les valeurs obtenues sont différentes. En comparant les résultats obtenus expérimentalement et les résultats attendus théoriquement, il est possible d'observer une grande différence.

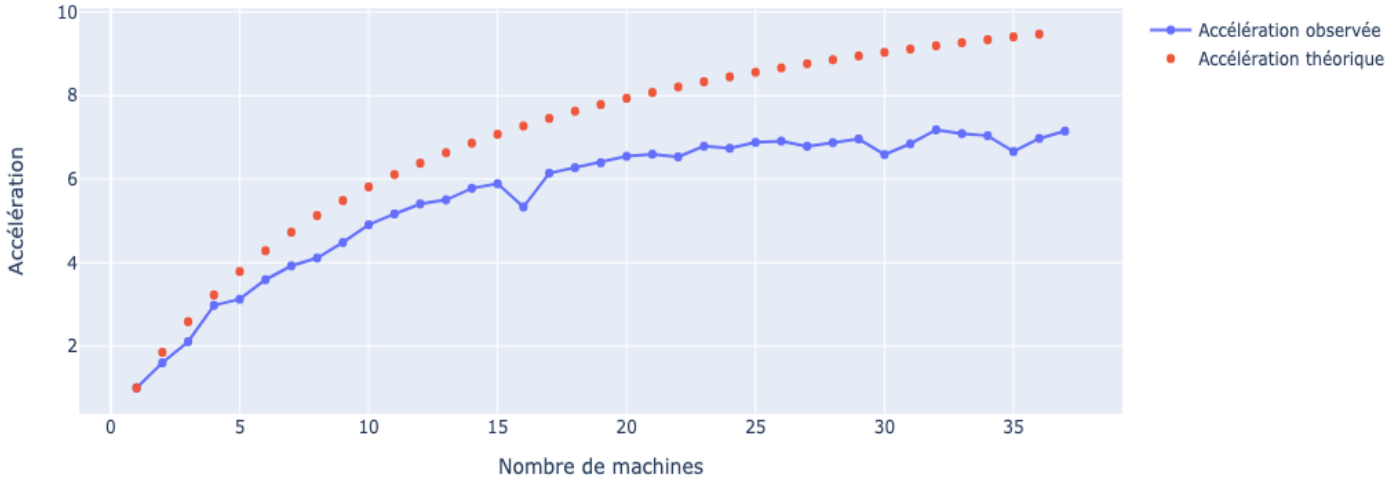

Figure 6 : Accélération théorique vs observée pour un taux de parallélisation de 92%

Cette différence peut s'expliquer par différents phénomènes. Tout d'abord le calcul du taux de parrallélisation qui dépend des temps d'exécution de chaque étape et qui peut donc varier en fonction des performances de la machine. Ensuite, la communication entre les machines ne doit pas être négligée. Plus le nombre de machines augmente, plus le nombre de fichier échangé augmente. Bien que le volumes de ces fichiers diminue, ces échanges peuvent être couteux en temps.

Il est possible donc de partir de ces observations et des résultats obtenus et d'en déduire les taux de parallélisation théorique correspondant à la courbe théorique se rapprochant le plus de notre courbe observée.

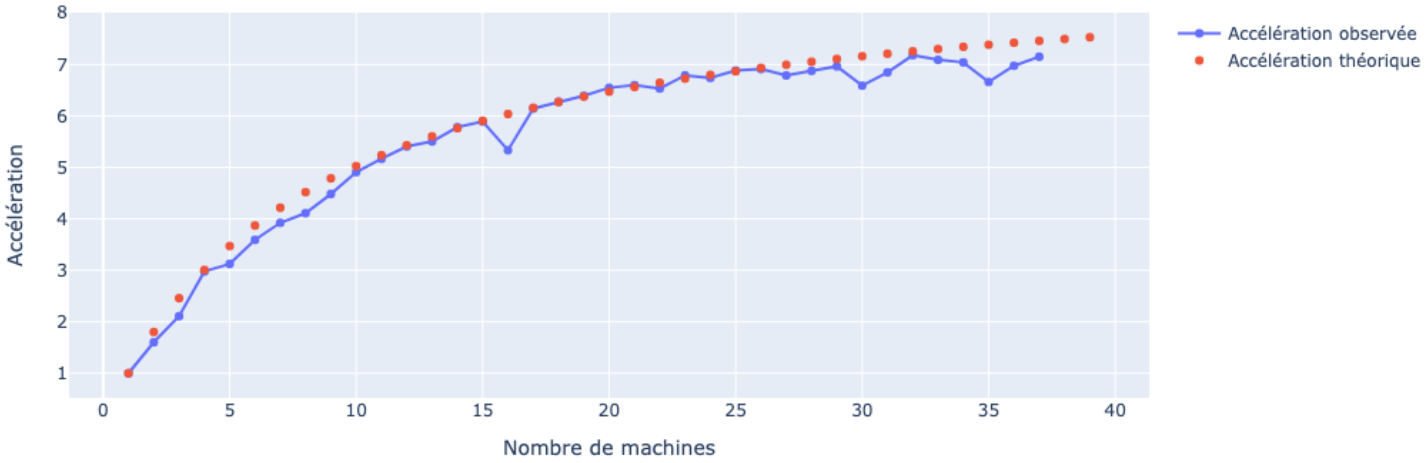

Figure 7 : Accélération théorique vs observée pour un taux de parallélisation de 88%

Il semblerait donc que le taux de parallélisation de notre programme soit de l'ordre de 88%.

Améliorations :

La première étape ayant subi une amélioration a été l'étape de split. Elle été réalisé en lisant une par une chaque ligne du fichier d'input et en réécrivant ces lignes dans différents fichiers. Lorsque le fichier commence à être volumineux, cette technique n'est pas très efficace. Les splits sont donc créés en divisant notre fichier en plusieurs fichiers de même taille. Cela a pour conséquence de diviser certains splits au milieu d'un mot, sans que cela ait beaucoup d'impact sur un fichier dont certains mots apparaissent plusieurs milliers de fois.

Le « Count Word » distribué a d'abord été lancé depuis un Master local (machine personnelle). Les résultats obtenus étaient très dépendants du réseau depuis lequel il était lancé (réseau Télécom, VPN). L'envoi des Splits du Master aux Workers demande beaucoup de temps. La solution à ce problème a été de choisir une des machines à notre disposition comme Master. Le Master est donc devenu la machine « tp-4b01-00 ». Cela a permis de passer d'une moyenne de 145 secondes pour envoyer les Splits à 3,5 secondes.

L'étape de « shuffle » est celle qui demande le plus de temps. La première version du shuffle consistait à rassembler un type de mot par fichier. Il y avait donc autant de fichiers que de mots différents. Lorsque le fichier d'entrer est peu volumineux, cela fonctionne bien mais dans le cas d'un fichier texte plus grand, le calcul ne se termine pas, bloqué par les envois des fichiers. Afin de palier à ce problème, chaque machine rassemble les mots destinés à une autre machine dans un seul fichier, chaque machine doit donc envoyer un maximum de 40 fichiers si les 40 machines à notre disposition sont utilisées.

Il existe encore plusieurs pistes d'améliorations possibles notamment sur la gestion des pannes. Dans le cas ou une machine n'est plus disponible, le split qui lui est envoyé est perdu. Il serait donc intéressant de pouvoir gérer cela pour toutes les étapes de calcul.

Comparaison :

Passons maintenant à la comparaison avec notre programme séquentiel. Comme vu précédemment, la communication entre les machines freine l'exécution de notre programme. L'avantage du programme séquentiel est qu'il n'a pas besoin de communiquer avec d'autre machine.

En comparant le temps pour différents volumes de fichiers d'entrées on en conclu qu'il est inutile, dans notre cas, d'avoir recours à un programme distribué pour des petits fichiers. L'utilisation de MapReduce à un réel intérêt lorsque le volume des données d'entrée ne permet pas au programme séquentiel d'être exécuté sur la machine.

Comme montré ci-dessous, le programme distribué permet d'obtenir de meilleurs résultats que le programme séquentiel sur la première machine. Mais pour la machine n°2 (avec de meilleures ressources), distribuer le calcul sur 37 machines ne permet pas d'obtenir d'aussi bons résultats.

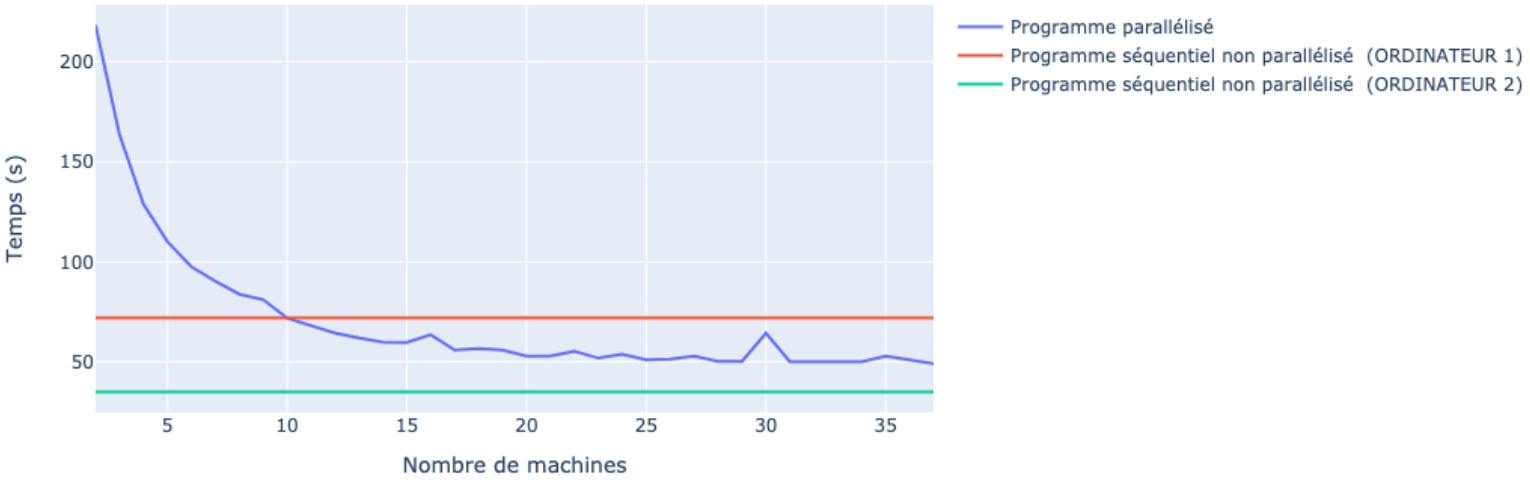

Figure 8 : Comparaison du programme séquentiel et distribué

Conclusion :

La parrallélisation et la distribution d'algorithmes permettent dans certains cas de pallier le manque de ressources des machines. La distribution des programmes par le biais du concept de MapReduce est possible grâce à la communication entre les machines et la division du calcul en tâches parallélisables.

Cette communication à un impact sur le temps d'exécution du programme. Plus le calcul est distribué, plus le nombre de communication augmente. De plus certaines parties de l'algorithme ne peuvent être parallélisées, le gain maximal atteignable par une parallélisation des tâches est alors limité par cette partie non parallélisable de l'algorithme comme le montre la loi d'Amdahl.

La visualisation de l'accélération de notre programme permet de mettre en évidence cette loi et de calculer le taux de parallélisation de l'algorithme.

Enfin, la comparaison des performances du programme distribué avec celles d'un programme séquentiel montre qu'il n'est pas toujours préférable de distribuer son calcul notamment lorsque les données traitées sont petites et facilement exploitables par la machine.