Année Universitaire 2022 – 2023
Matière : Système Distribué
Encadrant(s) : R. Sharrock

MS BGD

Projet Map Reduce

Florian Guily

28 novembre 2022

Résumé

Ce projet a pour but de mettre en pratique les concepts de calculs distribués et de communication entre machines vus en cours. Ce rapport, accompagné du code, vient détailler le chemin employé pour implémenter l'algorithme map reduce de manière distribué.

Sommaire

1 Introduction

Le but de ce projet est d'implémenter l'algorithme map reduce dans le language de notre choix de façon distribué. Il devra donc pouvoir être executé sur plusieurs machines. L'objectif final est de pouvoir démontrer la loi d'Amdahl par la pratique et déterminer de taux d'accélération de l'implémentation parallèle par rapport à une version séquentielle d'une fonction. Pour ce projet, nous allons compter le nombre d'occurence de chaque mots dans un texte. C'est une tâche assez simple et qui se prête parfaitement à l'algorithme map reduce. Les machines utilisées sont celles mises à disposition par l'école ainsi que ma machine personnelle (Ryzen 4700U, 16go RAM, WSL2).

2 Algorithme

2.1 Map Reduce

Map Reduce est un concept de programmation mis en place chez Google dans les années 2000 afin de parallèliser des algorithmes. Ce concept s'articule autour de 3 étapes :

- Map : Exécute un algorithme sur une partie des données. Dans ce projet, l'algorithme du word count comptant le nombre d'occurence des mots d'un texte est utilisé.
- Shuffle : Regroupe les clés identiques dans leur noeud correspondant.
- Reduce : Aggrège les valeurs des clés identiques pour n'avoir que des clés uniques.

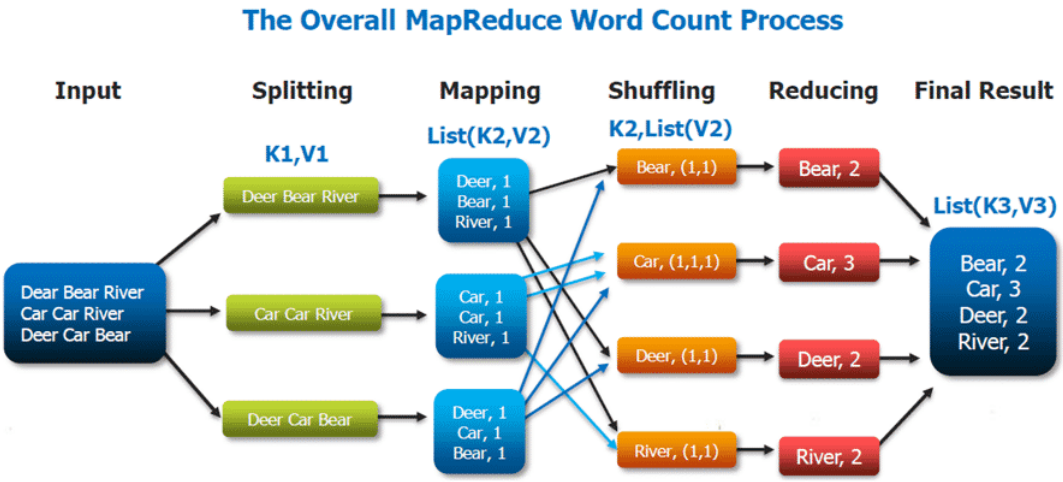

Figure 1 – Schéma de fonctionnement de Map Reduce

Le but de ce concept est donc de parallèliser l'execution d'un algorithme sur des noeuds. Ces noeuds exécutent cet algorithme sur un sous ensemble des données (Map), communiquent entre eux leurs résultats (Shuffle) et les aggrègent avant de les retourner (Reduce). Le résultat est donc le même que si l'algorithme avait été exécuté de manière séquentielle, sur une machine, et sur la totalité des données.

2.2 Environnement

Pour réaliser ce projet, nous avons à notre disposition les machines de l'écoles, accessibles via une connexion ssh. Ces machines représentent donc les noeuds sur lesquelles nous allons exécuter notre algorithme. L'architecture est donc la suivante :

— Des noeuds, représentés par des machines de l'écoles. Ici, une machine est un noeud. Ces noeuds exécutent un word count sur un sous ensemble de données et renvoient le résultat sur ce sous-ensemble.
— Un client, démarrant le processus de map reduce. C'est lui qui envoie les données aux noeuds. Ici, ma machine personnelle fait office de client, mais cela pourrait aussi être une machine de l'école.

2.3 Création du réseau

La première étape du programme est de mettre en place les structures nécessaires pour permettre à nos noeuds de communiquer entre eux. La difficulté est de savoir à qui parler et quand parler, notamment lors de la phase de shuffle, où beaucoup de données doivent être envoyées aux autres noeuds. Cette étape se base donc essentiellement sur la liste des addresses des noeuds. Celle-ci est données par l'utilisateur lorsqu'il configure l'algorithme. De cette liste découle l'identifiant de chaque noeud qui n'est autre que leur position dans cette liste. L'élement de base permettant d'effectuer la connexion avec un autre noeud est une socket. Celle-ci est utilisée afin d'envoyer et recevoir des messages avec un noeud. Il faut donc la créer puis la stocker.
Lors de son initialisation, un noeud attend donc une première connexion. Cette connexion sera forcement celle du client, il peut donc créer la socket correspondant à celui-ci et la stocker. Le client lui envoie ensuite la liste des adrresses des noeuds du réseau ainsi que sa position (ou son identifiant) dans cette liste. Ainsi, chaque noeud connait ses voisins ainsi que sa propre identité. Il ne reste plus qu'a créer les sockets vers les autres noeuds. Afin de les créers proprement, il faut que nos noeuds se coordonnent. Vers qui doivent-ils se connecter et de qui doivent-il attendre une connexion ?

Une règle simple mise en place ici est :

— Je me connecte aux noeuds ayant un identifiant superieur au mien.
— J'attend une connexion des noeuds ayant un identifiant inferieur au mien.

De cette manière, chaque noeud dispose d'une socket pour communiquer avec les autres noeud. Ces socket sont ensuite stockées dans une structure permettant de savoir à quel noeud elles correspondent.

2.3.1 POO

Je me suis très vite rendu compte que cette structure se prétait bien à la programmation orientée objet (POO). C'est un concept que je trouve très élégant et que j'affectionne particulièrement. J'ai donc créer deux classes :

— La classe PrincipalServer (PS) représentant le noeud sur lequel s'exécute le programme. Cette classe peut etre vu comme le noeud en lui-même.
— La classe NeighbourServer (NS), représentant les autres noeuds (les noeuds "voisins" du serveur courant).

De cette façon, il est possible de stocker des attributs pour chaque serveur comme les données à mapper pour le PS, la socket correspondant à un NS ou les données à lui envoyer lors du shuffle.

Cela permet aussi d'avoir des fonctions propres à chaque structure. Ainsi, il est plus cohérent d'associer les fonctions d'envoi et de récéptions aux NS car ces dernières se feront toujours du PS vers un NS.

Chaque noeud est donc représenté par un objet afin d'imiter au maximum le fonctionnement d'un réseau physique au sein du code. Cette configuration m'a fait gagner beaucoup de temps dans la compréhension du fonctionnement de mon code et dans sa conception.

2.4 Map

Une fois le réseau créé, il est temps de passer à la phase de map. L'algorithme exécuté lors de cette phase est celui du word count. Celui-ci prend un texte en entrée et renvoie une liste de clé valeur, la clé étant un mot du texte et la valeur son nombre d'occurence dans ce texte. Cet algorithme se prete très bien au Map Reduce car il est assez facilement parallelisable.

Chaque noeud reçoit donc une partie du texte initiale pour y appliquer un word count. Ils disposent donc d'un ensemble clé valeur qu'ils devront mettre en commun pour renvoyer le resultat final. Je tiens à préciser que cet ensemble clé valeur est unique et la valeur peut etre superieur à 1. Il existe des variantes où chaque mot est toujours associé à la valeur 1 et où il y a donc plusieurs fois la même clé dans l'ensemble clé valeur. Ici, nous utilisons la fonction Counter de python qui effectue déja une sorte de réduction locale au noeud et nous permet d'avoir un ensemble de clé valeur unique.

2.5 Shuffle

Une fois le word count effectué, il faut regrouper les mêmes clés sur un même noeud. En effet, chaque noeud ayant un split different du texte initial, ils possèdent probablements des clés identiques. Ces clés représentent le nombre d'occurence du mot dans chaque split du texte, il faut donc les regrouper sur un même serveur pour les additionner.
Problème : Comment sait on à quel noeud envoyer les clés ?
La réponse est d'utiliser le hash de la clé. Une fonction de hash basique va prendre une chaîne de caractère en entrée et renvoyer un entier en sortie. L'interet du hash est qu'une entrée produira une sortie unique et propre à celle-ci. De ce fait, peut importe le noeud sur lequel on se trouve, une même clé produira le même hash. Ce hash étant un entier, il suffit donc d'y appliquer un modulo n (n étant le nombre de noeud) pour obtenir l'id du noeud sur lequel envoyer le couple clé valeur correspondant. Chaque noeud sait donc, sans se concerter, à qui envoyer chaque clé valeur. Cela limite donc grandement les echanges de messages sur le réseau et rend notre implémentation beaucoup plus efficace.

Durant cette phase de shuffle, un noeud va donc construire n liste de clé valeur qu'il va populer selon le résultat du hash de chaque clé. Ces listes correspondent aux couples qui vont être envoyé aux NS, elles sont donc stockées dans leur NS correspondant. Une fois populées, elles sont sérialisées puis envoyées. Les noeuds devant à la fois envoyer et recevoir des données, il est nécessaire d'utiliser des threads pour realiser tout ça. Un thread est un processus qui va permettre d'exécuter du code en arrière plan. Il est donc possible d'en lancer plusieurs en même temps afin d'avoir une exécution parallèle du code. Deux threads sont donc lancer par NS, un pour l'envoi et un pour la réception. Ces threads sont bien sur stockés dans les objets NS correspondant.
Enfin l'envoi de message se fait de façon automatique avec des fonctions utilisant les fonction send et recv de la bibliothèque socket afin d'éviter d'avoir à gérer le nombre d'octet à lire lors de la réception. Ces fonctions utilise un suffixe entier sur 4 octets en début de message pour indiquer la taille du message qui suit. De cette manière, il est possible d'envoyer des messages de taille arbitraire sans perte d'information.

2.6 Reduce

La partie reduce consiste à sommer les valeurs d'une même clé. L'utilisation du defaultdict en python vient nous faciliter la tâche. Cette étape est assez simple et ne diffère que très peu du cas séquentiel. Après ça, vient la derniere étape où chaque noeud envoie au client ses clés réduites. Celui-ci n'a plus qu'a écrire les données finales dans un fichier local. (à noter que le client est géré comme un NS)

3 Mesures

Afin de mettre en avant la loi d'Amdahl, il est impératif de mettre en place des mesures du temps d'exécution. Au fil de mes tests, je me suis rapidement rendu compte que le goulot d'étranglement de mon map reduce se situait au moment de l'envoi initial des données. Celles-ci étant ammenées à peser plusieurs giga octets, l'envoi prenait plusieurs dizaine de secondes et représentait la majorité du temps d'exécution. Il à donc été choisi de mesurer aussi le temps après l'envoi de ces données. Les mesures ont été faites avec la bibliothèque time de python. Chaque étape a été mesuré, coté client et coté noeud. Seul les mesures coté client ont été gardé. Une référence séquentielle à été réalisé afin de mesurer son temps d'exécution. Toutes les mesures ont été réalisé 5 fois puis moyenné afin de limiter la variance des résultats et avoir des mesures plus robustes.

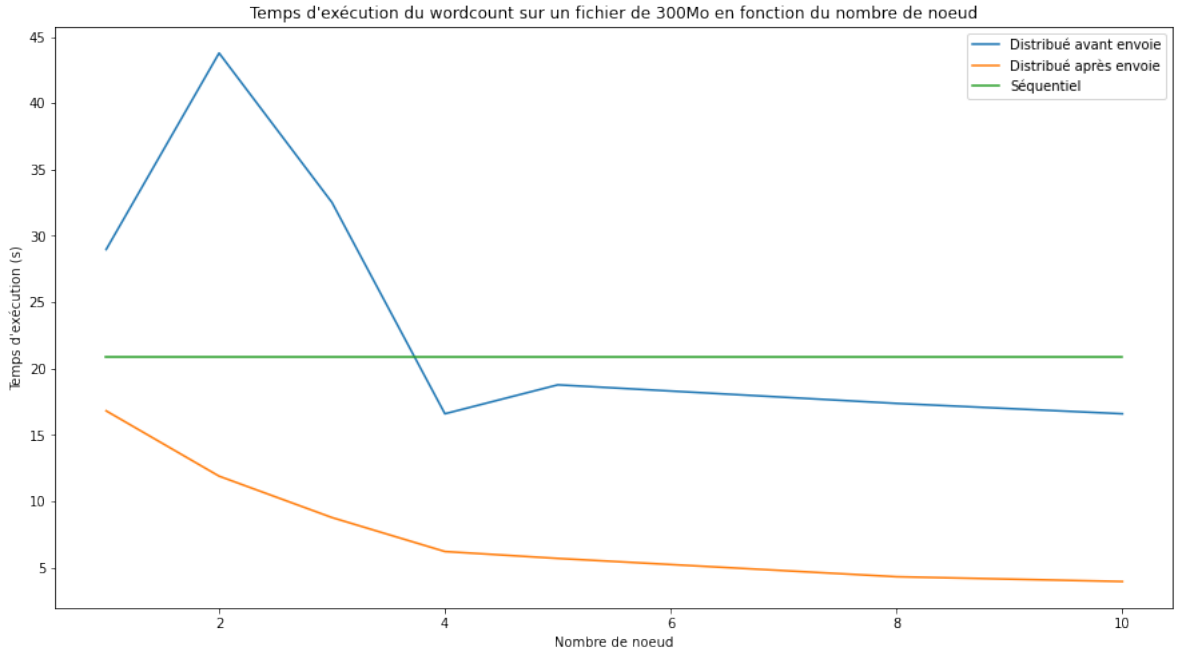

FIGURE 2 – Mesures de l'accélération

Ces mesures ont été effectué sur le premier fichier split mis a disposition par Mr Sharrock. Ce choix à été fait car il n'était pas possible d'en charger deux en mémoire localement pour la mesure séquentielle. Ce fichier pèse environ 300Mo, ce qui est suffisant pour observer une accélération.
On remarque que le temps moyen avant envoi est beaucoup plus élevé avec 2 coeurs. Cela est dû au réseau au moment de la prise de mesures. Globalement, la loi d'Amdahl semble etre validée, notamment sur les mesures après envoi. On obtient une accélération x5 pour 10 noeuds par rapport à la version séquentielle. Si on inclut le temps d'envoi des données, il faudra utiliser au moins 4 noeuds pour avoir un gain en performance par rapport a la version séquentielle. L'accélération sera plafonnée par ce temps d'envoi.