Rapport sur le projet *make distribué*

Version du 18 décembre 2023

Par Arvind Candassamy, Monica Huynh, Tien-Dat Phan et Théo Rozier

Table des matières

I/ Travail réalisé

1. Description du travail réalisé

Au cours de ce projet, nous avons construit un programme permettant de parser des fichiers Makefile avec le langage de programmation Scala. Notre programme peut parser une version minimale du langage des Makefiles : les règles contenant une cible, avec ou sans dépendances, et avec une ou plusieurs commandes. Les variables et les fonctions ne sont pas gérées. La gestion parallèle de l'exécution des différentes tâches a nécessité de construire un algorithme d'ordonnancement pour la gestion des dépendances, qui sera détaillé dans la section 4. Ensuite, le programme exécute les cibles du même niveau dans l'arbre de dépendance en parallèle. En cas d'erreur, le programme interrompt l'ensemble des tâches. Nous n'avons pas réalisé le transfert des données entre les nœuds et nous nous reposons sur le système de fichier NFS.

Plusieurs scripts d'automatisation pour réserver des nœuds, déployer l'environnement de Spark sur Grid5000, des tests d'expérimentation et la génération des courbes graphiques des résultats ont été écrits. Une image contenant l'installation de nos technologies (l'intergiciel) a également été faite dans le registry du site *grenoble* de Grid5000. Vous retrouvez tout le code effectué sur notre dépôt GitHub.

2. Intergiciel

Notre intergiciel s'appuie sur l'utilisation d'Apache Spark 3.5.0 avec le langage de programmation Scala dans sa version 2.12.18. Spark est un outil conçu pour le calcul distribué sur la JVM avec une architecture Master-Worker. Nous déployons un cluster Spark sur Grid5000 avec un nœud Master et plusieurs nœuds Worker. Spark assigne des tâches à paralléliser aux Workers et gère la communication entre les nœuds ainsi que leur terminaison. À l'intérieur d'un nœud Worker, des exécuteurs sont créés pour traiter les tâches reçues en utilisant les cœurs qui leur sont assignés.

L'image que nous déployons sur chaque nœud est créée avec Tgz-g5k et est basée sur Debian-10-nfs par défaut. Nous avons installé Spark, le compilateur GCC et le JRE (Java Runtime Environment) sur cette image.

3. Algorithme pour la gestion de dépendances

Chaque cible pouvant dépendre sur d'autres cibles, il est nécessaire d'établir un ordre d'exécution pour les commandes de nos cibles afin d'être certain que toutes les dépendances sont prêtes. Pour cela, on veut savoir quelles cibles peuvent être exécutées en parallèle, et dans quel ordre. On cherche donc à créer une liste contenant elle-même des listes de cibles. Nous devons également éliminer les cibles dont les fichiers sont à jour, et donc n'ont pas besoin d'être exécutées à nouveau, cela implique également qu'il y a des cibles implicites, qui ne sont pas forcément définies, mais dont des cibles peuvent dépendre, par exemple pour des fichiers. Le paramètre pour notre algorithme est une liste de cibles finales qui doivent être satisfaites après exécution ordonnée de toutes les cibles.

Notre algorithme consiste à maintenir une liste de cibles ainsi que leur ordre d'exécution (les cibles d'ordre N s'exécute en parallèle lorsque toutes les cibles d'ordre N-1 ont terminé leur exécution, les cibles d'ordre 0 s'exécutent en premier). À chaque itération, on calcule pour chaque cible l'ordre qu'elle doit avoir relativement à ses dépendances, si une cible n'a aucune dépendance son ordre est forcément 0,

si elle a des dépendances, son ordre vaut l'ordre maximum parmi ses dépendances, plus un. Nous exécutons cette itération tant qu'au moins un ordre a changé dans la boucle, cet algorithme tend vers une stabilité dans laquelle toutes les cibles ont un ordre de `1 + max(ordre(dépendances))`. Si nous ne parvenons pas à être stables, cela signifie qu'il y a une récursion infinie dans la déclaration des cibles, nous considérons cela comme une erreur. Voici un court pseudo-code résumant le concept :

```
calc_scheduling(targets, initial_targets):
  ordered_targets = {}  // map de target -> ordre
  new_targets = initial_targets
  changed = true
  while changed:
    changed = false
    for new_target in new_targets:
      ordered_targets[new_target] = 0  // ordre initialement à zéro
      new_targets.clean()
    for target, order in ordered_targets:
      if not target.dependencies.is_empty():
        dep_order = 0
      else:
        for dep in target.dependencies:
          if dep in ordered_targets:
            dep_order = max(dep_order, ordered_targets[dep])
          else:
            new_targets.push(dep)
        if order != dep_order + 1:
          order = dep_order + 1
          changed = true
  // transposition de target -> order à liste de liste de targets
  return transpose(ordered_targets)
```

La deuxième étape consiste à éliminer toutes les cibles qu'il n'est pas nécessaire d'exécuter, cela s'applique uniquement aux cibles pour lesquelles un fichier existe avec le même chemin/nom. Lorsqu'un fichier de cible a été modifié antérieurement au fichier Makefile lui-même, alors on garde la cible, car les modifications apportées au Makefile ont potentiellement changé les commandes pour le créer. Sinon, on vérifie les fichiers des dépendances de la cible, si l'une de ces dépendances au moins a été modifiée ultérieurement au fichier de la cible, alors on garde la cible. Dans tous les cas contraires, les cibles sont éliminées. Au final, si un ordre ne contient plus aucune cible, on l'élimine, ce qui permet d'exécuter toutes les cibles suivantes un ordre plus tôt.

II/ Performances

Pour l'ensemble des tests de performance réalisés, nous avons travaillé sur le cluster dahu du site de *grenoble* sur Grid5000. Une machine a 64 coeurs.

1. Network File System

NFS est le système de fichiers en réseau monté par défaut sur les nœuds de Grid5000. C'est un système permettant de partager les fichiers stockés sur plusieurs serveurs distants comme s'ils étaient des fichiers locaux. Cela permet donc une simplicité vis-à-vis de la gestion des données entre les différents nœuds d'un même cluster sur Grid5000. Cependant, ce système de fichiers se trouve sur le réseau standard lent de la machine Grid5000. Puisque nos expérimentations reposent sur ce système de fichiers, nous décidons de monter un plan d'expérimentation pour mesurer les performances brutes de l'écriture et de lecture sur NFS.

Pour ce faire, un premier script *latency.sh* a été écrit au sein du dossier */grid5000/measures/nfs/*. Il consiste à mesurer le temps que prend l'écriture dans un fichier se trouvant sur le NFS. Après avoir comparé plusieurs outils de benchmark comme fio, IOzone ou Bonnie++ sur leur installation, leur utilisation et leur précision, on a décidé d'utiliser la commande *dd* qui est déjà disponible sur les machines et permet de réaliser des tests simples de lecture / écriture. Le scénario est tel qu'on va mesurer la latence d'écriture et de lecture pour différentes tailles de bloc (4k à 64k par pas de puissance de 2) :

1) Pour une taille de bloc t1, on va d'abord mesurer la latence de l'écriture de 100 000 blocs de taille t1 dans un fichier se trouvant sur le NFS

2) On fait l'expérience 10 fois avant de calculer la moyenne

3) On passe ensuite à la mesure de la latence en lecture de 100 000 blocs de taille t1 sur le fichier qui vient d'être généré dans l'étape 1.

4) Tout comme l'étape 2, on refait la mesure 10 fois pour obtenir une moyenne.

A chaque fin de mesure de calcul, on écrit le temps mis (en secondes) pour exécuter la commande dans un fichier CSV. On se retrouve avec un nouveau dossier */results/* comprenant le fichier généré *latency.csv*. On s'assure de ne pas utiliser le cache en assignant la valeur *direct* aux attributs *iflag* et *oflag*.

Ensuite, un programme en R a été écrit pour générer les courbes de performance depuis le fichier latency.csv grâce à la bibliothèque *ggplot2*. Notre fichier *latency_perf.R* se trouve au même endroit que le script *latency.sh*. Ce programme va lire le fichier CSV, créer deux courbes (une pour la latence d'écriture et une autre pour la lecture) par rapport à la taille des blocs utilisés et nous enregistrons ce graphique dans le fichier *nfs_latency_plot.png* dans le dossier */results/*.

Finalement, un script a été créé pour automatiser le lancement des mesures et la génération des courbes : le fichier *generate_latency_nfs_plots.sh*. Pour que tout fonctionne correctement, il est nécessaire d'avoir les permissions de le lancer et également d'installer le package ggplot2 dans R sur le nœud réservé de Grid5000. Les indications sont précisées dans le README.md du dépôt.

2. Stratégie de tests

Nous avons mesuré la performance de notre algorithme en suivant plusieurs configurations :

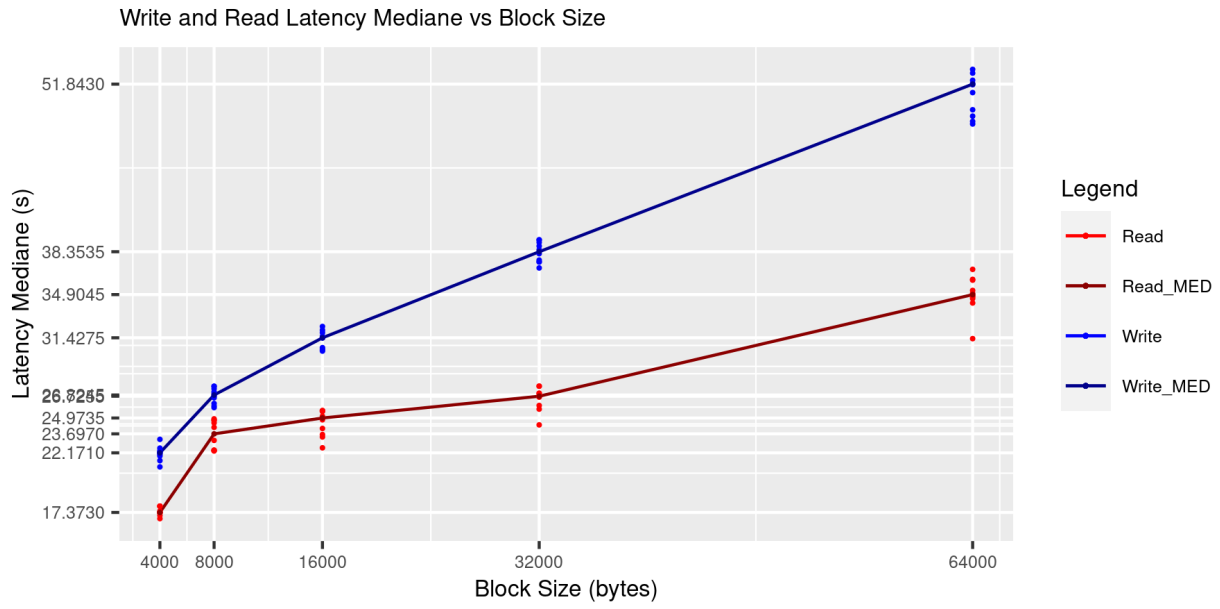
- Avec différents nombre de nœuds / workers réservés sur Grid5000 allant de 1, 2, 4, 8 et 16.
- Avec plusieurs itérations
- Avec différentes caractéristiques sur l'exécuteur de Spark : le nombre d'exécutors total (1, 2 ou 4) sur un nœud, leur nombre de cœurs (16, 32, 64) et une mémoire de 1Go.
- Et pour le makefile matrix, on a fait varier le nombre de découpes (2, 4, 8,16,32)

Pour chaque expérience, nous mesurons à la fois le temps nécessaire à l'ordonnancement des dépendances sur le nœud maître et le temps d'exécution distribué sur les esclaves. Le dernier intervalle est calculé à partir du moment où Spark commence la repartition de travail et exécution

3. Analyse des résultats

a. NFS

La durée totale de l'exécution des scripts jusqu'à la génération des courbes avec les paramètres par défaut (10 itérations) prend presque une heure.



Voici le fichier image générée par le script R. On affiche les 10 points ainsi que la médiane pour chaque taille de bloc pour la latence en écriture et en lecture. De ce graphique, on analyse que l'écriture sur le NFS est plus lente que la lecture d'un facteur moyen de 1.3. Dans les deux cas, l'ordre de grandeur de la mesure est de l'ordre de la *seconde* allant pour la lecture de 17,3s pour une taille de blocs de 4k à 34,9s pour 64k et pour l'écriture de 22.1s pour 4k à 51.8s pour 64k. Ce temps est beaucoup plus lent que ce à quoi on s'attendait puisqu'on est surtout habitué à l'utilisation du cache du système, et à être dans de l'ordre du millisecondes.

b. Test sur le Makefile premier

Ce Makefile est composé de vingt commandes de compilation indépendantes qui pourraient être parallélisées. Avec un node worker sur le cluster dahu à Grenoble, on a réussi à atteindre l'accélération théorique auquel on s'attendait. En effet, nous avons tout d'abord calculé le temps de l'exécution avec la commande *make* sur une machine de Grid5000. La commande make effectue son exécution en séquentiel et a pris un temps d'exécution d'environ 400s.

Nous avons déterminé que le temps d'exécution du Makefile pouvait se représenter avec 98% de calculs, tous indépendants et donc parallélisables, et 2% par la dernière opération qui consiste à concaténer l'ensemble des fichiers générés. Le modèle théorique qu'on a construit se base sur l'accélération théorique de la loi d'Amdahl :

$$S_{\text{latence}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

Et les explications des différents signes provenant de Wikipédia :

- S_{latence} est l'accélération théorique en latence de l'exécution de toute la tâche ;
- s est le nombre de fils d'exécutions (*threads*) utilisés pour exécuter la tâche
- p est le pourcentage du temps d'exécution de toute la tâche concernant la partie bénéficiant de l'amélioration des ressources du système *avant l'amélioration*.

Dans notre cas, nous pouvons indiquer que lors de l'utilisation de plusieurs processeurs, 20 calculs pourront être accélérés (98% du temps de compilation) donc que s vaut 20 et que p vaut 0.98. Ainsi :

$$S_{latence}(s) = \frac{1}{1 - 0.98 + \frac{0.98}{20}} \approx 14,49$$

Donc nous nous retrouvons avec un temps théorique de $\frac{400}{14,49}$ environ égal à 27,6 secondes.

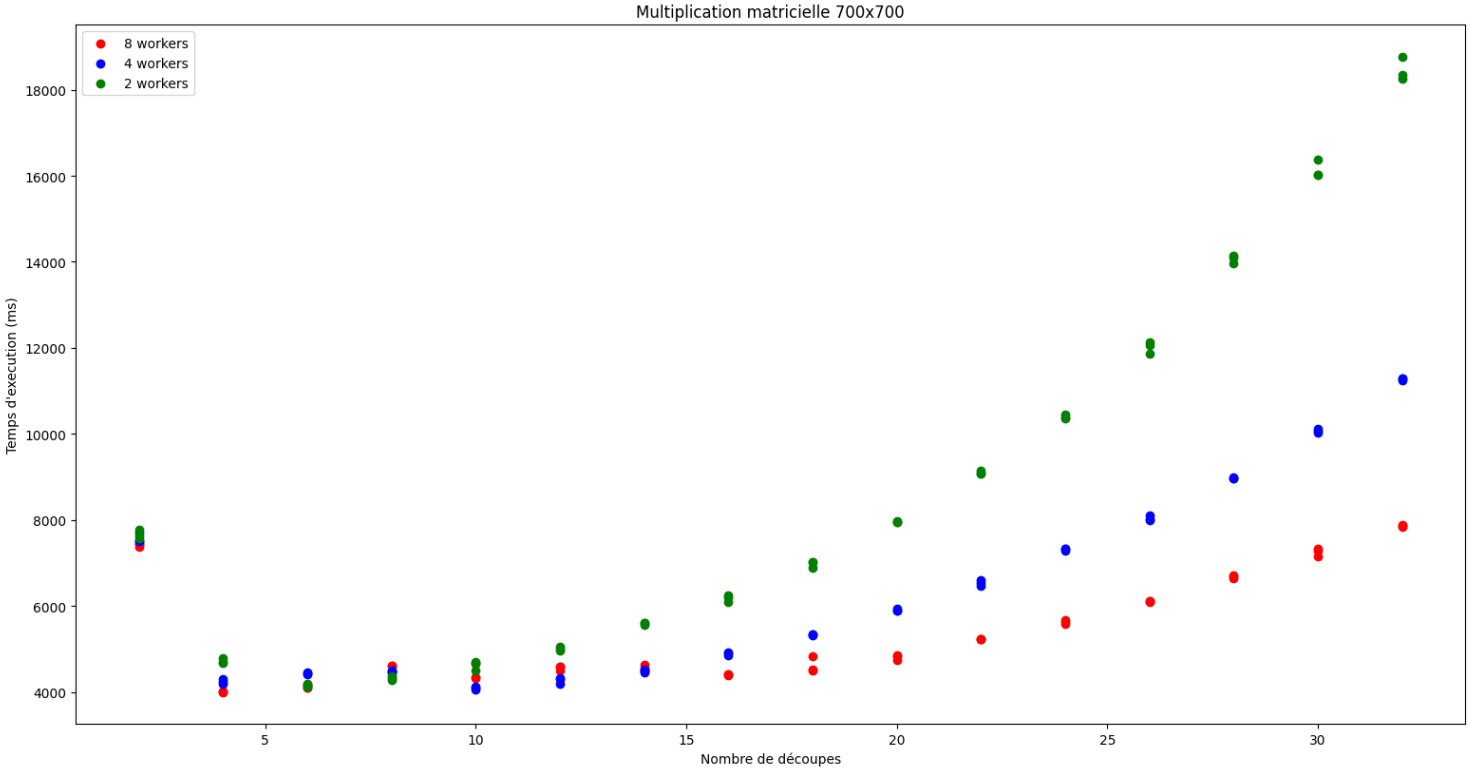
Lorsque nous avons mesuré le temps d'exécution sur ce Makefile, nous avons relevé un temps de 24 secondes : ce qui est cohérent voire meilleur que l'accélération théorique. En plus, le passage à l'échelle dans ce cas-là ne va pas augmenter la performance puisque le programme est simple et peu coûteux.

c. Test sur le Makefile matrix

Ce Makefile consiste à réaliser un produit matriciel. Il est possible de définir le grain de découpage. On a décidé de mesurer le temps d'exécution sur une matrice de dimension 20 et pour différentes tailles de découpes (précisées dans la stratégie de tests). Le script associé à ce test est situé dans */grid5000/experiments/* et se nomme *run_experiment_matrix.sh*. A la fin de l'exécution du script, un fichier *matrix_result.txt* est généré à l'endroit où le script a été lancé.

Cependant, nous avons constaté que trop tard que la mesure des performances sur une matrice de dimension 20 ne nous permettait pas d'observer le passage à l'échelle, car un nœud worker avec 64 cœurs permet déjà d'atteindre le temps optimal pour le cluster Spark. Ensuite, nous avons réalisé des expérimentations sur des dimensions plus grandes : 700 et 1000, et nous avons remarqué des différences:

Un nombre élevé de découpages entraîne une diminution rapide du temps d'exécution, mais au-delà d'un certain seuil, le temps d'exécution commence à augmenter. Ainsi, le point de découpage optimal ainsi que la vitesse peuvent varier en fonction du nombre de cœurs dans le cluster (ou du nombre de nœuds workers), ainsi que de la dimension de la matrice. Un nombre trop élevé de découpages engendre une grande quantité de travail à paralléliser, dépassant éventuellement la capacité du cluster. Voici le graphique qui montre les performances différentes pour les trois configurations.



d. Test sur Spark configuration

En pratique, Spark peut être optimisé en ajustant le nombre d'executors par nœud worker et la stratégie de partition en fonction de la nature des données pour obtenir de meilleures performances. Dans notre application, nous avons expérimenté différentes valeurs pour ces variables.

- Le nombre de partitions est un facteur crucial qui détermine la distribution des commandes entre les exécuteurs. Chaque partition est traitée exclusivement par un exécuteur, bien qu'un exécuteur puisse prendre en charge plusieurs partitions. Ce paramètre être l'un des éléments clés ayant

considérablement amélioré les performances de la parallélisation. Par défaut, le nombre de partitions était fixé à 2, ce qui limitait la concurrence.

En pratique, le nombre de partitions est défini comme un multiple du nombre total de cœurs dans le cluster. Dans notre cas, nous avons réglé ce nombre sur le nombre de tâches à paralléliser. Cela peut avoir un impact sur la performance, notamment lorsque le nombre de tâches est élevé (comme dans le programme matrix, par exemple), et peut même dépasser le nombre total de cœurs. Dans de telles situations, certains exécuteurs peuvent se retrouver à traiter un nombre plus élevé de partitions que d'autres.

- L’’augmentation du nombre d'exécutors sur un nœud Worker n'a pas amélioré, voire a empiré, les performances. Cette situation découle du fait qu'un exécuteur représente une instance de la machine virtuelle Java (JVM), entraînant une surcharge potentielle sur le TaskScheduler de Spark, notamment en ce qui concerne le lancement et l'isolation des ressources.