Mikolaj Pawlikowski
Laurent Alepee
Xavier Martin

Projet Systèmes Distribués
Équipe Hadoop

1. Présentation de Hadoop
    Grands volumes de données
    MapReduce
    Architecture
2. Application à un Makefile
    Parsage des cibles
    Approche du problème
    Analyse de la solution
3. Forcer un éléphant à faire ce qu'il ne veut pas
    Map and don't reduce
    Round Robin
    Échange des fichiers
4. Installation et déploiement
5. Tests de performance
    BLENDER 2.59
    Sleep
    SUBWAY
        Environnement - 320 coeurs sur 79 machines
        Premier essaie - repartionner par défaut
        Deuxième essaie - Round Robin Repartitionner (RRR)
        Troisième essaie - 158 frames sur 79 noeuds
    Efficacité
6. Conclusion

1. Présentation de Hadoop

Grands volumes de données

Hadoop est un environnement conçu pour simplifier le traitement de grands volumes de données sur un système distribué. Hadoop s'occupe entre autres choses de l'échange de données entre noeuds de calcul, mais son intérêt principal est le paradigme de découpage de travail "MapReduce".

MapReduce

Cette approche se résume à découper un grand ensemble de données en paquets **indépendants** de **(clé, valeur)**. On effectue dessus un premier traitement "map" qui a pour but d'extraire une information elle aussi de la forme **(clé, valeur)**. Toutes les paires Clé-Valeur sont envoyées au "reduce", les paires possédant la même clé sont agrégées en **(Clé, (Valeur1, Valeur2)).** Le reduce produit un **(clé, valeur)**. C'est l'enchaînement typique d'un MapReduce.

Architecture

L'architecture utilisée pour Hadoop est composée de deux nodes principaux, le NameNode et le TaskNode (JobTracker), ainsi que de plusieurs DataNodes et JobNodes . Le NameNode est en charge de la gestion des données et de leur emplacement sur le HDFS (Hadoop Distibuted File System). Le JobTracker est en charge de la distribution des tâches (Jobs). Les DataNodes s'occupent du stockage et les JobNodes éxécutent l'algorithme Map-Reduce. Par défaut, tous les slaves peuvent dévenir les noeuds de stockage et/ou de calcul en fonction de besoin.

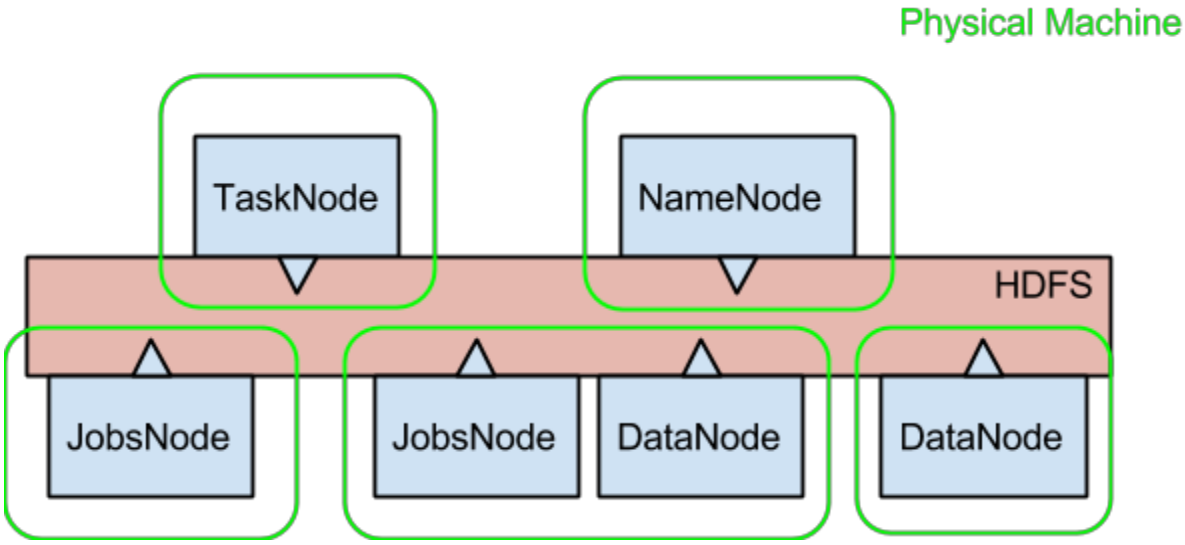


fig. 0. Architecture du système

2. Application à un Makefile

Les tâches MapReduce étant **indépendantes** l'une de l'autres, et les jobs s'éxécutant en séquentiel entre eux, nous devions trouver un moyen de "couper" nos dépéndaces en parties indépendants (qui peuvent donc s'éxécuter en parallèl. Nous avons donc proposé un algorithme suivant.

Parsage des cibles

La première étape du processus est de parser le Makefile afin de récupérer la liste des actions a faire ainsi que les dépendances entre ceux-ci. Pour ce faire l'algorithmes crée un arbre de dépendance, celui-ci a pour tête l'instruction principal et pour fils les dépendance de compilation.

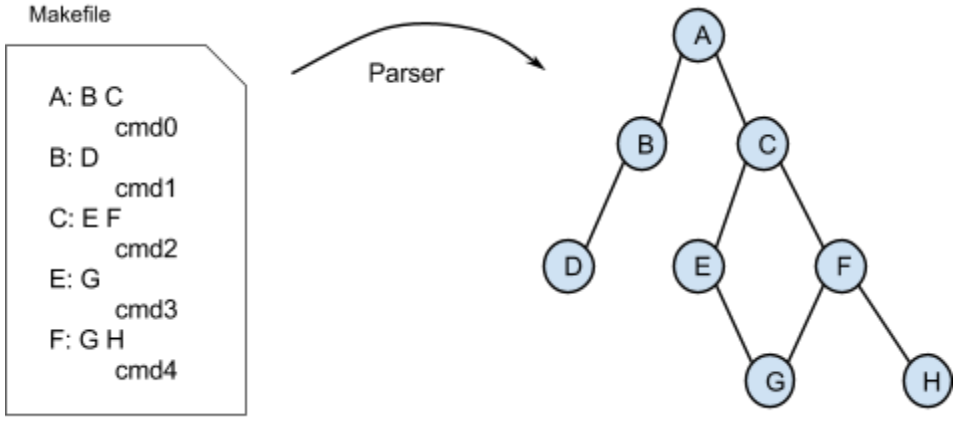


fig 1. Exécution des jobs par paquet de feuille

Approche du problème

Notre solution lance une instance de MapReduce par paquet de feuilles du graphe de dépendance. (voir figure 2)

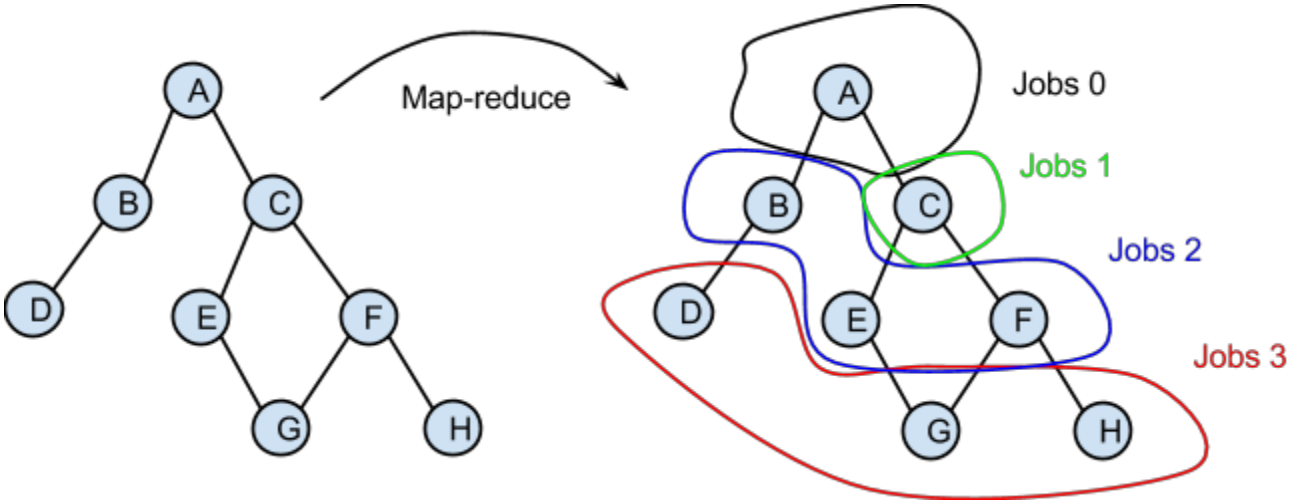


fig. 2. Lecture de l'arbre et création des jobs

Une fois un paquet de feuilles exécuté, on peut les enlever de l'arbre et exécuter les nouvelles feuilles. On s'arrête quand l'arbre est vide.

L'exécution ce fait donc dans l'ordre des dépendances soit: Jobs3, Jobs 2, Jobs 1 puis Jobs 0.

Analyse de la solution

Hadoop n'est pas l'outil approprié: C'est ce qui est ressorti de nos essais. L'approche MapReduce fonctionne pour de grands volumes de données **indépendants**. Ce n'est pas le cas dans un Makefile où l'on ne manipule pas des données mais plutôt des cibles dépendantes les unes des autres.

Non-optimal: Cette solution laisse à désirer, surtout pour les arbres verticaux qui vont engendrer des coûts fixes de mise en place de Job conséquents. De plus, si le volume de travail par unité de compilation est hétérogène, des pertes sont engendrées par l'attente à la "barrière" d'un paquet de feuilles du graphe. Toutefois, notre solution est adaptée aux arbres horizontaux et équilibrés.

Manque de flexibilité: La création de jobs à la volée aurait pu être une solution au problème de barrière mais cela n'est pas une utilisation prévue de Hadoop. Nous le verrons par la suite, la mise en place d'un job est tellement lente que ce n'est pas envisageable.

3. Forcer un éléphant à faire ce qu'il ne veut pas

Map and don't reduce

Premièrement, map et reduce ont été prévu pour traiter rapidement les paires clé-valeur, en mode texte. Les tâches potentiellement longues comme un rendering 3D sont donc complètement contre la philosophie d'Hadoop. Pour qu'il exécute les tâches "feuilles", on stocke dans un fichier temporaire une liste de commandes à exécuter. Durant un job, le map ne fait que passer la main. Le reduce télécharge les dépendances nécessaires pour la commande, puis l'exécute et renvoie le résultat. On triche donc l'éléphant.

Round Robin

Une des parties difficile pour Hadoop, c'est d'ajouter de l'intelligence dans la répartition des tâches qu'il propose. Mais vu qu'il ne sait pas que les lignes à l'entrée sont des commandes à éxécuter, sa stratégie par défaut ne nous permettait pas de mettre au travail tous les noeuds. Nous avons donc écrit donc un Partitioner en mode round robin, pour que tous les noeuds aient du travail.

Échanges des fichiers

Les tâches Hadoop sont indépendantes l'une de l'autre, et le stockage HDFS sert principalement à stocker d'un façon fiable les données d'entrée et de sortie de l'algorithme.
Dans notre utilisation, on copie des fichiers (6MB de sources pour subway, par example) depuis le HDFS dans les noeuds et puis on stocke les résultats à la fin de calcul. HDFS n'a pas été optimisé pour les écritures/lectures en temps réel.

4. Installation et déploiement

L'environnement Hadoop que nous avons testé est en **version 1.2.1**.
Le parc de machines est relativement facile à mettre ne place - il suffit de préparer un accès ssh sans mot de passe de master vers les noeuds de calcul. Ensuite, dans la configuration de NameNode, il suffit de mettre les hosts dans le fichier do configuration conf/slaves.

Une fois l'environnement prêt, il faut compiler le jar qui va exécuter notre Make.
Dans le dossier "make" de l'archive rendue, il y a un projet maven/eclipse contenant les sources. Pour le compiler on utilise la commande maven suivante:

```
mvn clean compile jar:jar
```

Cela produit un jar prêt à utiliser avec la commande hadoop:

```
hadoop jar {nom de jar} {nom de la classe à éxécuter} {dossier avec Makefile dans HDFS} {goal make}
```

Exemple d'utilisation se trouve dans le fichier ./make/launch_echos.sh :

```
# ajout de fichiers dans HDFS
hadoop fs -copyFromLocal ../Makefiles/echos/ make-echos

# Utilisation d'un jar
hadoop jar make-0.0.1-SNAPSHOT.jar hadoop_playground.make.Make make-echos all.txt

# vérification de l'output
hadoop fs -cat make-echos/all.txt
```

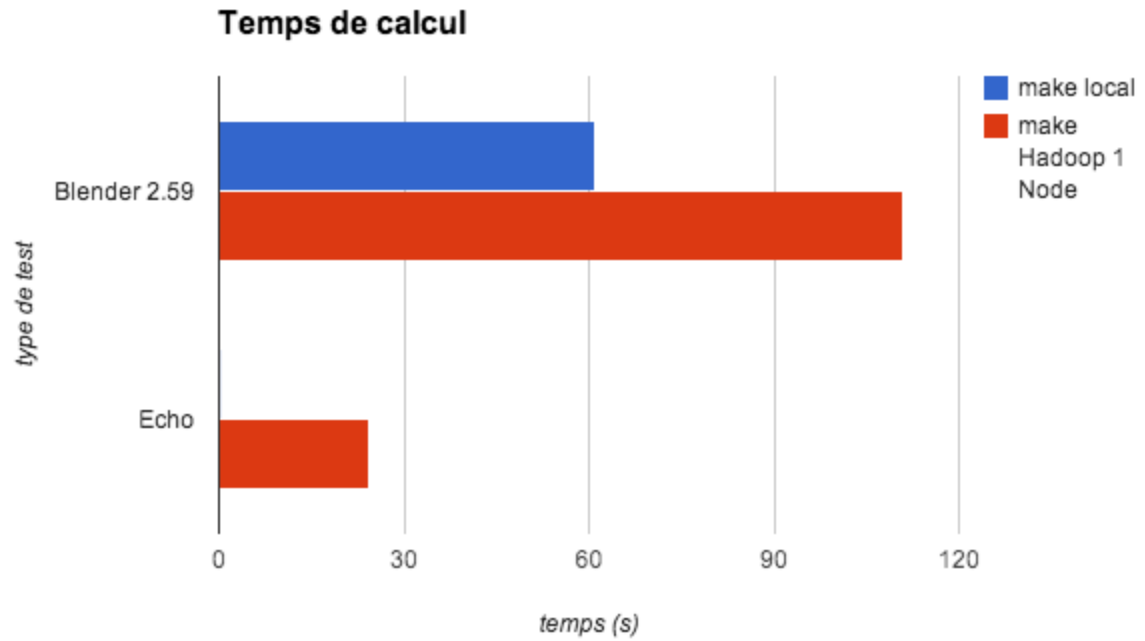
5. Tests de performance

Dans nos tests, nous avons essayé d'adresser différents cas de figure. Nous avons donc fait les tests sur différents types de jobs (courte durée, longue durée), en faisant varier le nombre de noeuds (1-79) et la structure de notre Makefile.

BLENDER 2.59

Tout d'abord nous avons essayé d'évaluer le surcoût de la couche Hadoop-HDFS-replication dans une tâche simple. Nous avons donc utilisé l'example de Blender 2.59 avec un seul Noeud de calcul et nous l'avons comparé à une exécution d'un make local. L'observation d'un surcoût présenté de ~25 seconds par "couche" a été ensuite confirmé par un lancement d'un makefile à un seul target "echo hello > test.txt", qui a pris 25 secondes pour s'exécuter.

Commande à utiliser:
time hadoop jar make-0.0.1-SNAPSHOT.jar hadoop_playground.make.Make blender_2.59 out.avi
real 1m51.556s
user 0m1.895s
sys 0m0.081s



Sleep

Ensuite, pour vérifier le fonctionnement du système pour un Makefile qui contient un arbre équilibré, nous avons simulé des tâches plus longues par un sleep de 10 secondes. Comme prévu, un make local donne un résultat de 1m20.470s car les tâches sont éxécutées en séquence.

Ensuite, sur un cluster de 15 noeuds (seulement 8 qui travaillent), la commande

```
time hadoop jar make-0.0.1-SNAPSHOT.jar hadoop_playground.make.Make sleeps all.txt
```

donne un temps de 52.230s.

Vu que le surcoût n'est à payer qu'une seule fois, on perd en moyenne ~52.2/8 = 2.775s par job, ce qui nous donne **une efficacité de 35%.**

Nous voyons donc clairement que ce n'est pas un cas d'utilisation intéressant.

SUBWAY

Motivations

Cette fois, nous cherchions à mettre en place un cas d'utilisation dans lequel notre solution pourrait se défendre. Pour amortir le coût de Hadoop, nous avions donc besoin d'une tâche plus longue, qui pourrait s'exécuter en parallèle sur beaucoup de machines. Nous avons préparé un fichier blender qui prend environ 4m30 pour générer une frame. Les fichiers de test se trouvent dans Makefiles/subway.

Pour accélérer les choses, nous avons écrit le fichier *generate.py*, qui permet de créer un Makefile complet prêt à éxécuter avec un nombre n de frames dans le film final.

Exécution séquentielle d'une commande make:

real 4m28.387s
user 16m35.853s // tous les 4 coeurs travaillent à ~100%
sys 0m2.367s

Environnement - 320 coeurs sur 79 machines

Nous avons préparé un cluster de 81 nodes (toutes les machines de 2ème étage), avec un NameNode, un JobTracker et 79 noeuds de calcul. Nous lançons un Makefile qui produit 79 frames.

Premier essai - 'Repartionner' par défaut

Le premièr lancement nous donne les temps suivants: **19m17.704s**

Dans les logs nous voyons que uniquement 39 noeds ont travaillé, et que certains ont fait juste une frame, tandis que les autres en ont produit 3.

Deuxième essai - Round Robin 'Repartitionner' (RRR)

Suite à la création d'un partitionneur custom pour réaliser un round-robin, les performances ce dernier sont largement améliorées .

Toujours avec 79 noeds et 79 frames: **5m36.339s**

L'efficacité atteint 79%

Troisième essai - 158 frames sur 79 noeuds

Cette fois, nous avons atteint le temps de **10m38.872s**

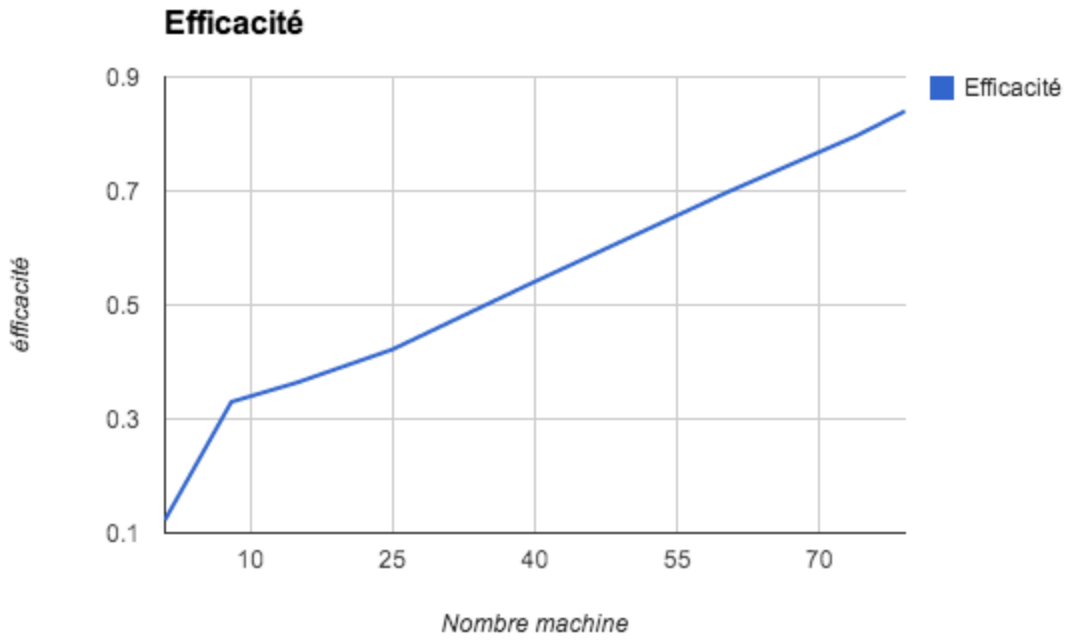
Ceci est possible, puisque tous les noeuds ont travaillé à 100% pendant ce temps, et il n'y avait aucun problème (toutes les machines stables).

L'efficacité atteint 84%

Il est important de noter que pour ces 158 frames, les noeuds ont stocké 1.5 GB de fichiers temporaires dans HDFS. Dans notre configuration nous avons configuré la duplication sur 3 noeuds, ce qui ralentit les I/O.

Efficacité

Voici la courbe d'efficacité pour les mesures d'exemple Subway pour les différents nombres de noeuds (1, 8, 16, 32, 39, 60, 74, 79 machines), où chaque noeud calcule une frame. On voit clairement une amortissement du surcoût introduit par Hadoop qui monte avec le nombre des noeuds. On voit aussi que le coût lui même ne change pas beaucoup, ce qui représente un des gros avantages d'Hadoop - lancer un cluster de 80 machines n'est pas beaucoup plus difficile que de 2 machines (pas de configuration des slaves).



Graphique de l'éfficacité par machine

6. Conclusion

Hadoop est très bien, mais il n'est clairement pas conçu pour ce qu'on attendait de lui.
Le sur-coût entraîné par la création des jobs, la répartition et le stockage redondant pénalise les tâches rapides. Notre algorithme de gestion de Makefile est efficace pour les graphes de dépendances horizontaux qui exécutent des commandes avec un temps d'exécution dans le même ordre de grandeur.

Nous avons néanmoins réussi à implémenter un outil qui:
- est efficace dans les cas cités plus haut,
- est relativement facile à déployer sur des clusters de grande taille une fois la configuration d'Hadoop bien comprise,

- permet le stockage fiable avec un nombre de duplicats configurable,
- gère les failles de noeuds de calcul et la répartition des tâches,
- fourni les mécanismes de reprise après panne,
- permet l'ajout de noeuds à la volée, passe à l'échelle rapidement et facilement,
- peut être déployé sur n'importe quelle machine capable de lancer un programme java via ssh.

Les résultats obtenus sont donc conformes à ce qu'on attendait et malgré les limitations décrites, nous avons trouvé des cas d'utilisation intéressants pour notre solution.