Makefile Distribué avec Apache Thrift et Go

Nathanaël Couret, Clément Taboulot, Vincent Chenal, Maxime Hagenbourger

I. Le travail

1) Apache Thrift

Thrift est un framework RPC présenté par Facebook en 2007 et actuellement hébergé par Apache. Thrift propose son propre langage de définition d'interface (IDL) pour définir des services RPC qui peuvent ensuite être utilisés dans une grande variété de langage. Il rend les interactions entre client et serveur transparente et facilite donc le travail du développeur qui n'a plus qu'à travailler dans son langage préféré.
Dans le cadre de ce projet, le langage sélectionné est le Go. Go est un langage impératif, compilé et concurrent facile à prendre en main du fait d'une syntaxe inspirée du C mais très épurée, d'un support natif du multithreading et de la présence d'un ramasse-miette. La documentation est également de bonne qualité et facilite beaucoup l'apprentissage.

2) Choix techniques

- **Parsing** :
L'étape du parsing et l'une des première tâche effectuée. Le parseur produit un arbre de dépendances. Cette arbre est composé de noeuds représentant une cible du Makefile. Si cette cible possède des dépendances alors le noeud associé dans l'arbre possède des liens avec les noeuds correspondants.

- **Ordonnanceur** :
Il est continuellement sollicité pour retourner une tâche éligible à être calculée. Il s'agit d'un parcours de l'arbre issu du parsing qui retourne les noeud exécutables. Ce choix est dirigé par deux paramètres. Le premier est l'attribut *computing* qui indique si un noeud est en train d'être exécuté par un serveur. Et le deuxième est l'attribut *done* qui indique si un noeud a fini son exécution. La sélection d'un serveur se fait quand à elle avec un algorithme *Round Robin*.

- **Services** :
Nos serveurs n'exposent que deux services. Le premier est la possibilité d'exécuter une ligne de commande. Le deuxième est l'arrêt des serveurs à la fin de l'exécution du client. Le démarrage des serveurs est quand à lui assuré directement par le client qui se connecte en

ssh sur les hosts pour les démarrer. Le côté négatif est l'obligation d'attendre un certains temps que toutes les serveurs soient bien démarré avant de les solliciter (par défaut 2 secondes dans notre implémentation).
Ces services ne retournent qu'un entier qui décrit le statut avec lequel s'est déroulé l'exécution du service.

- **Exclusion mutuelle** :
L'utilisation d'un service par le client se fait dans un thread qui : appelle le service, récupère sa valeur de retour, et modifie l'état du noeud à l'origine de cette appel. Nous avons été amené à utiliser une variable de mutex pour gérer les accès concurrents entre les threads et le programme principal.
Après réflexion notre gestion de ces sections critiques auraient pu être plus fine : un mutex par noeud par exemple. Dans notre rendu il n'existe qu'un seul mutex.

- **Environnement** :
Nous avons fait le choix de confier le transfert de fichier à NFS. Notre home de grid5000 est monté sur chacun des serveurs. De plus chacun d'eux a un utilisateur avec le même UID que notre utilisateur grid5000. Cela présente plusieurs avantages. Le premier est que nous n'avons pas à gérer le transfert du binaire de l'application. Le deuxième est la facilité pour rapatrier les résultats de chaque serveur. En effet, chaque machine "a l'impression" d'écrire dans le même espace mémoire grâce à NFS, ce qui facilite grandement l'implémentation.

3) Description de l'exécutable.

L'exécutable que nous fournissons possède deux modes de démarrage : client ou serveur.
Quand on désire effectuer une compilation il faut le lancer en mode client. Le client s'occupe ensuite de démarrer les serveurs sur toutes les machines.

```
LINE COMMAND
      dmake [OPTIONS]
OPTIONS
      -server=<bool> : démarrage en mode serveur (défaut : false)
      -P=<string> : protocole utilisé
      -framed=<bool>
      -buffered=<bool>
      -addr=<addr:port> : addresse d'écoute pour les serveurs
      -secure=<bool> : communication sécurisée
      -hostfile=<string> : chemin (défaut:./hostfile.txt)
      -makefile=<string> : chemin (default:./Makefile)
```

4) Déploiement

cf README.md dans les sources (https://github.com/Ataww/SDCA-Makefile)

II. Les performances

Procédure de test:

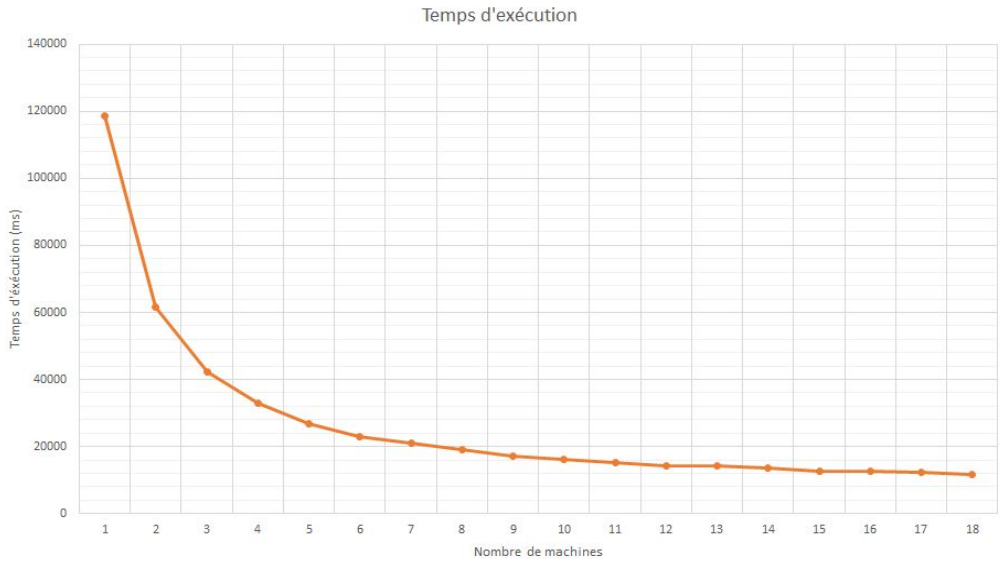
Pour effectuer nos tests nous avons respecté la procédure de déploiement décrite dans le Readme pour mettre en place le cluster de machines. L'ensemble des machines utilisées pour la série de tests sont ainsi réservées dès le début de la procédure.
Pour modifier le nombre de machines présentes pour le test, nous avons créé un fichier hostfile à partir de la liste de noeuds réservés (uniq $OAR_NODE_FILE). Nous avons commencé par commenter toutes les machines sauf la première puis en avons décommenté une pour la mesure suivante jusqu'à arriver à 18 machines.

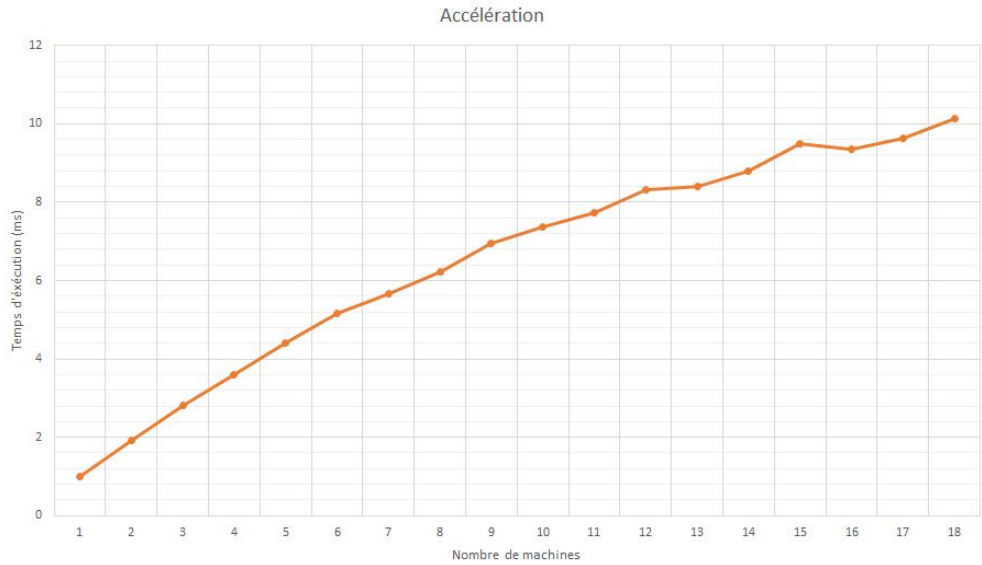
Le timer est démarré au lancement du client et stoppé lorsque l'on a traité toutes les cibles.

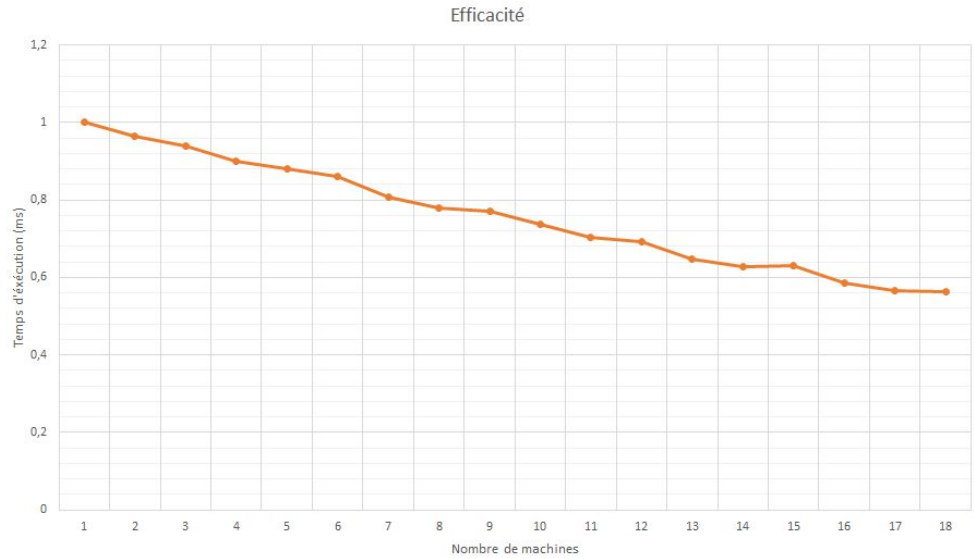
Les tests ont été réalisés sur des machines Genepi sur le site Grenoble de grid5000. Ces machines contiennent 8 cores chacunes mais nous n'avons pas tiré parti de ces coeurs (1 coeur/machine).

Nous n'avons pas recalculé x fois la même expérience de manière à mettre en évidence une zone de confiance autour de la courbe. Il aurait été intéressant de le faire.

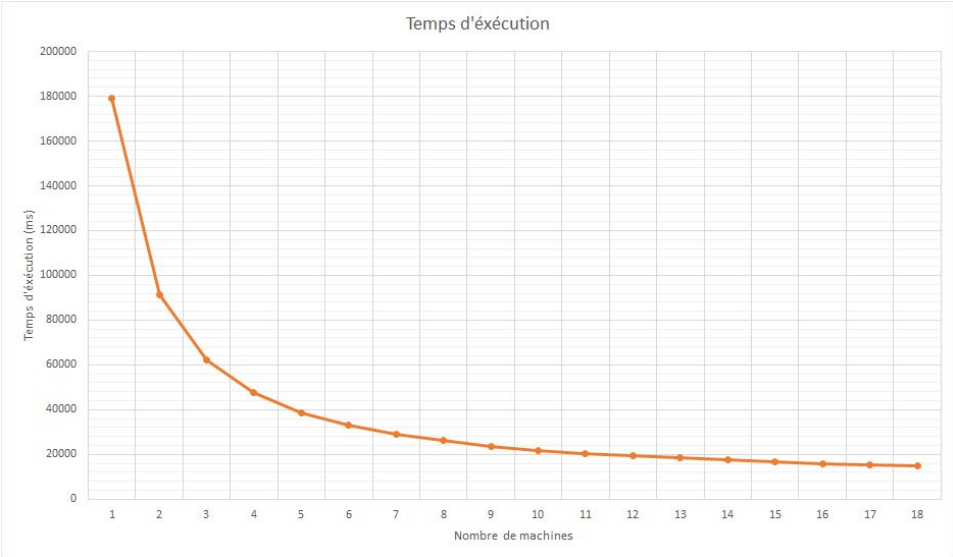
Makefile blender 2.49

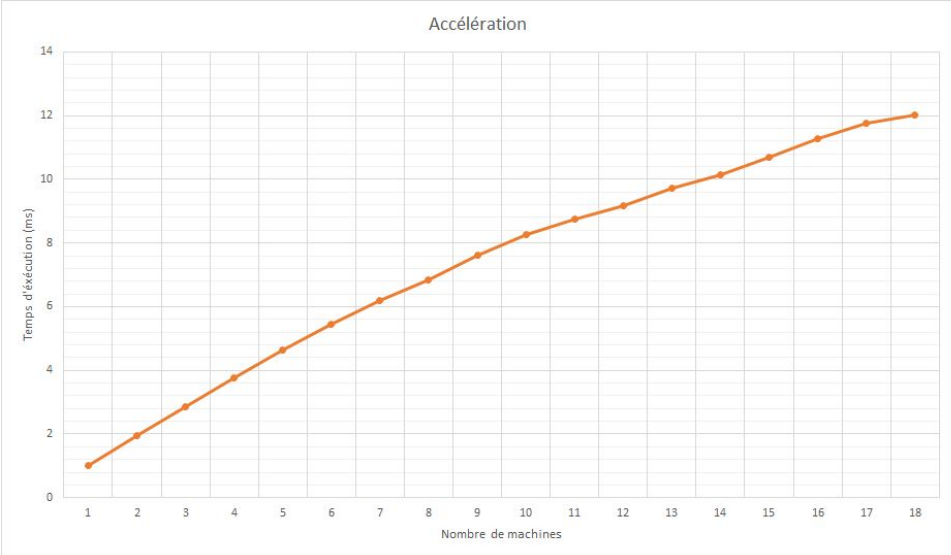


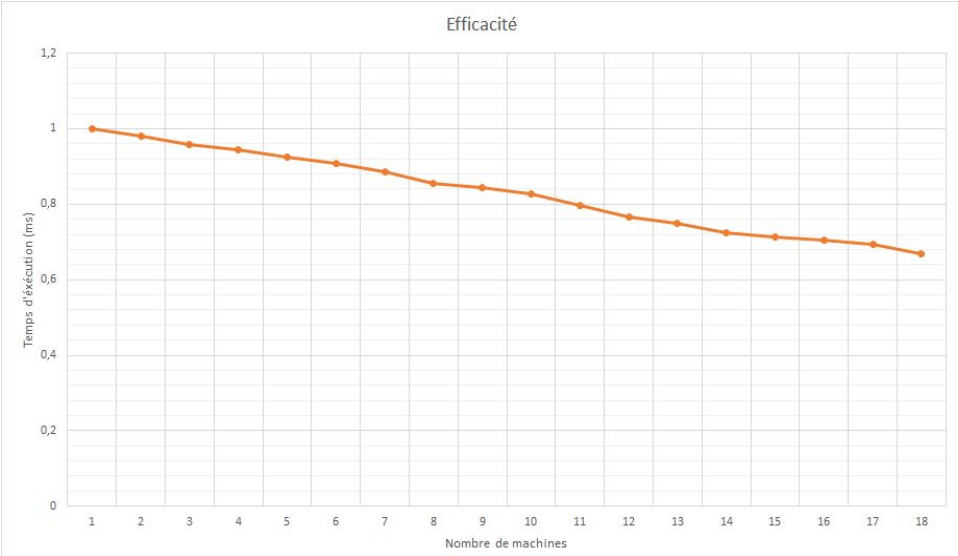




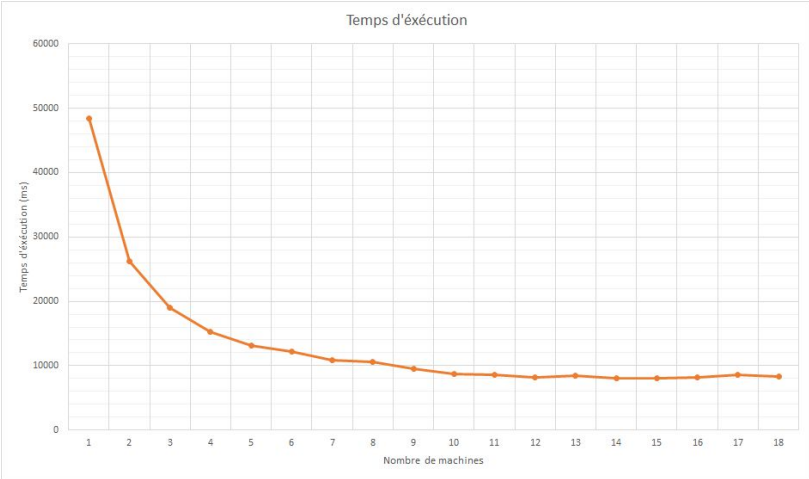
Makefile blender 2.59

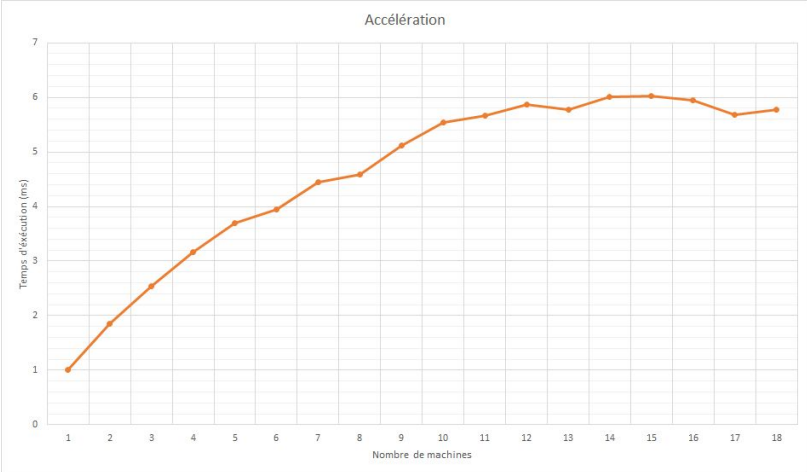


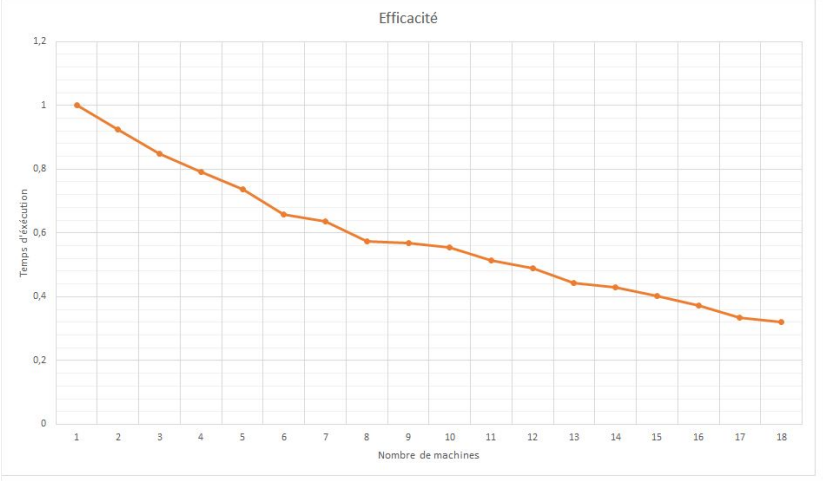




Makefile premier-small







Analyse des résultats

Pour les trois Makefiles testés on obtient de bons résultats et notre implémentation permet bien d'accélérer le calcul. Cependant on peut remarquer certaines différences entre les différents Makefiles.

De manière générale, et pour chaque Makefile, le temps d'exécution diminue significativement au début jusqu'à atteindre une certaine stabilité. Au bout d'un moment l'ajout de nouvelles machines au sein du cluster n'est plus utile.
La courbe d'accélération réagit de manière parfaitement inverse, elle augmente significativement au début puis finit par se stabiliser.
Cette stabilisation se traduit également par une baisse de l'efficacité. Plus on ajoute des machines, plus l'aspect traitement de la distribution des cibles (logique + mutex) et l'aspect réseau joue un rôle limitant.

Cependant on remarque tout de même des différences importantes entre les Makefiles blender et premier.
En calculant les Makefiles blender_2.49 et blender 2.59 ont obtient des valeurs d'accélération de 10 et 12 respectivement avec un cluster de 18 machines.
Alors que le calcul du Makefile premier-small n'est que 6 fois plus rapide pour le même nombre de machines.

La "taille" des commandes distribuées, au sens temps de calcul, est un facteur très important. En effet il faut que cette taille soit significativement plus grande que l'overhead induit par le réseau et la logique de distribution des tâches pour obtenir de bonne performance. Sinon cela veut dire que l'on passerait autant de temps à distribuer les tâches qu'à les calculer. Autrement dit il serait dans ce cas préférable de les calculer soit même plutôt que de déléguer le travail.

Nous pensons que ce critère peut expliquer la différence d'accélération entre les Makefiles blender et premier.

Pistes d'amélioration

Performances:

- Utiliser plus de coeurs sur chaque noeud en distribuant plusieurs cibles aux machines au lieu d'une seule systématiquement.
- Essayer d'autres configurations de mutex (une variable de mutex par noeud au lieu d'une seule pour tout les noeuds)

Déploiement :

- Créer une image debian qui contient directement nos dépendances.
- Déployer uniquement le binaire de l'application dans le cluster de calcul plutôt que d'installer tout l'attirail nécessaire à la compilation.
- Éviter le couplage de l'application à l'environnement de déploiement (lancement de /home/user/Go/bin/dmake ...)