Systèmes distribués et Cloud avancé

Makefile parallèle - Go RPC

Novembre 2017

Table des matières

Ludovic Carré, Maxime Deloche,
Vincent Lefoulon & Omar Sanhaji

1 Projet

1.1 Langage et bibliothèque utilisés

1.1.1 Langage Go

Le langage Go a été créé en 2009 par Google. Compilé et avec un typage fort et statique, il a à l'origine été pensé comme un langage "facile à comprendre et facile à adopter" par de jeunes développeurs. Il intègre nativement le multithreading (via un mot-clé `go` permettant de lancer une fonction dans une *goroutine*), et est utilisé dans plusieurs projets d'envergure (par exemple Docker, Kubernetes ou Dropbox).

1.1.2 Bibliothèque RPC

RPC (*Remote Procedure Call*) est un protocole réseau. Il est utilisé pour créer une architecture client-serveur, en permettant à des clients d'effectuer des appels de procédures distantes sur un serveur.

Go RPC est un package Go implémentant ce protocole, et permettant l'appel distant à des méthodes exportées.

1.2 Structure

1.2.1 Architecture de l'application

L'architecture de notre système distribué est un master-slave : une machine (la première dans la liste de réservation) est désignée comme master, et se charge de parser le Makefile et de l'ordonnancement des tâches.

Toutes les autres machines sont des slaves, et font des requêtes au master pour récupérer des tâches. A partir de là, il y a deux possibilités :

- Si il y a des tâches qui sont disponibles : le master répond au slave en lui envoyant la tâche (les instructions et les fichiers nécessaires à sa réalisation). Le slave effectue la tâche, et renvoit le résultat au master (via une seconde fonction distante), avant de refaire une requête pour obtenir une nouvelle tâche.
- Si il n'y a aucune tâche à effectuer pour le moment : le slave se met en 'attente' : il démarre un serveur RPC, et c'est au master de le recontacter lorsqu'une tâche est disponible. Cela permet d'éviter d'effectuer des requêtes inutiles, et de gagner du temps (pas besoin d'attendre une requête, la tâche est affectée dès qu'elle est disponible). Le master, quant à lui, tient une liste des machines en attente, et les réveille lorsque de nouvelles tâches sont disponibles. Si toutes les règles ont été effectuées, le master réveille alors ces slaves en attente à l'aide d'une autre fonction, qui leur demande de s'éteindre.

1.2.2 Organisation du code

La racine du projet contient les dossiers et fichiers suivants :

- `bin` : Contient les exécutables pour les programmes master, slave et sequential.
- `makefiles` : Contient principalement les makesfiles *parallelizable* et *tree* ainsi que d'autres makefiles utilisés pour tester le projet au fur et à mesure de son avancement mais dont les performances n'ont pas été prises en compte.
- `measures` : Contient les graphes de mesure de performances sur les différents makefiles.
- `outputfiles` : Contient les dépendances nécessaires aux slaves pour exécuter les règles données par le master.
- `scripts` : Contient tous les scripts de configuration et mesure utilisés pour le projet.
  - `common.sh` : Déclare les variables globales utilisées par tous les scripts.

- `compile.sh` : Compile les programmes sur le NFS avec taktuk.
- `convert_to_gnuplot.py` : Convertit le JSON brut des mesures en un fichier compatible avec Gnuplot pour générer les graphes.
- `create_non_para_makefile.py et create_para_makefile.py` : Permettent de générer un makefile non parallélisable ou entièrement parallélisable.
- `deploy.sh` : Permet de déployer l'image *jessie-x64-nfs* sur tous les noeuds réservés par Grid5000 et installe Go sur les machines.
- `master.sh, slave.sh et sequential.sh` : Permettent de lancer les différents programmes.
- `plot.sh et plot_one.sh` : Permettent de créer les graphes de performance du système.
- `test.py` : Permet de gérer toute la procédure de test.
- `check_nodes.sh` : Permet de vérifier quels noeuds donnés par Grid5000 sont fonctionnels.

.

1.3 Déploiement

Le déploiement du code sur Grid5000 est fait en plusieurs étapes. Il s'agit tout d'abord de réserver le nombre de noeuds souhaité pour l'exécution puis de cloner le projet dans le NFS de Grid5000. Ensuite le script *deploy.sh* va utiliser `kadeploy3` pour démarrer les machines avec jessie-x64-nfs. Enfin, deux commandes `taktuk` vont faire l'update et l'installation de Go sur les machines. Le script *compile.sh* va ensuite demander à un noeud de faire la compilation de chacun des différents programmes (master, slave et sequential) dans NFS.

2 Performances

2.1 Méthodologie

2.1.1 Mesures effectuées

Nous avons mesuré :

- sur le **master** : le temps total d'exécution
- sur chaque **slave** : le temps total d'exécution, le temps passé à exécuter des tâches, le temps passé à attendre de recevoir des tâches du master

Ces deux derniers temps nous permettent de calculer le pourcentage de temps effectivement utilisé par les slaves pour faire des calculs, et le temps passé à interagir avec le master.

Nous utilisons 2 types de Makefiles différents :

- un Makefile "parallélisable" : toutes les tâches dépendent d'une seule tâche racine, et sont donc complètement parallélisables
- un Makefile en "arbre" : c'est un Makefile relativement complexe et partiellement parallélisable

Ces Makefiles contiennent des `sleep` de temps différents, afin de simuler des traitements complexes.

Pour chacun de ces Makefiles, nous avons tracé le temps d'exécution total (celui du master) en fonction du nombre d'esclaves, et calculé le taux d'efficacité des slaves :

$$\frac{\textit{temps passé à travailler}}{\textit{temps total} - \textit{temps à attendre une tâche}}$$

On exclut le temps passé à attendre une tâche du temps total, puisque ce n'est pas du temps qui aurait pu être utilisé à calculer (car à ce moment-là, il n'y avait pas de tâche disponible).

2.1.2 Précision

Tous les temps sont calculés en milli-secondes. Chaque test est effectué 30 fois, et la moyenne est calculée : cela permet de limiter l'impact de l'incertitude des mesures. Nous comparons chaque Makefile à son temps d'exécution séquentiel avec notre programme.

2.1.3 Intervalle de confiance

Nous avons n mesures, de moyenne $\bar{x}$ et de variance σ (ces deux valeurs sont empiriques). On utilise, pour calculer l'intervalle de confiance, une loi de Student (avec $\alpha = 5\%$) :

$$[\bar{x} \pm \frac{Z_{\alpha/2} * \sigma}{\sqrt{(n)}}]$$

avec $Z_{\alpha/2}$ pour une loi de Student à $n - 1 = 29$ degrés de liberté qui vaut 1.697.

2.2 Conditions expérimentales

2.2.1 Spécifications des machines

Nous avons testé avec les machines du cluster `graphene` de grid5000, à Nancy : elles contiennent des processeurs Intel Xeon 4 coeurs à 2.53GHz, avec 16Go de RAM et une distribution Debian SMP.

2.3 Résultats obtenus

Le temps avec 0 slave représente le temps séquentiel (sans les communications donc). Le temps avec un seul esclave est donc supérieur : c'est cohérent. On constate également que le temps d'exécution diminue fortement et par palier au fur et à mesure que l'on rajoute des machines en parallèle, jusqu'à stagner (lorsqu'il y a assez d'esclaves pour que toutes les tâches possibles soient parallélisées).

2.3.1 Makefile "Arbre"

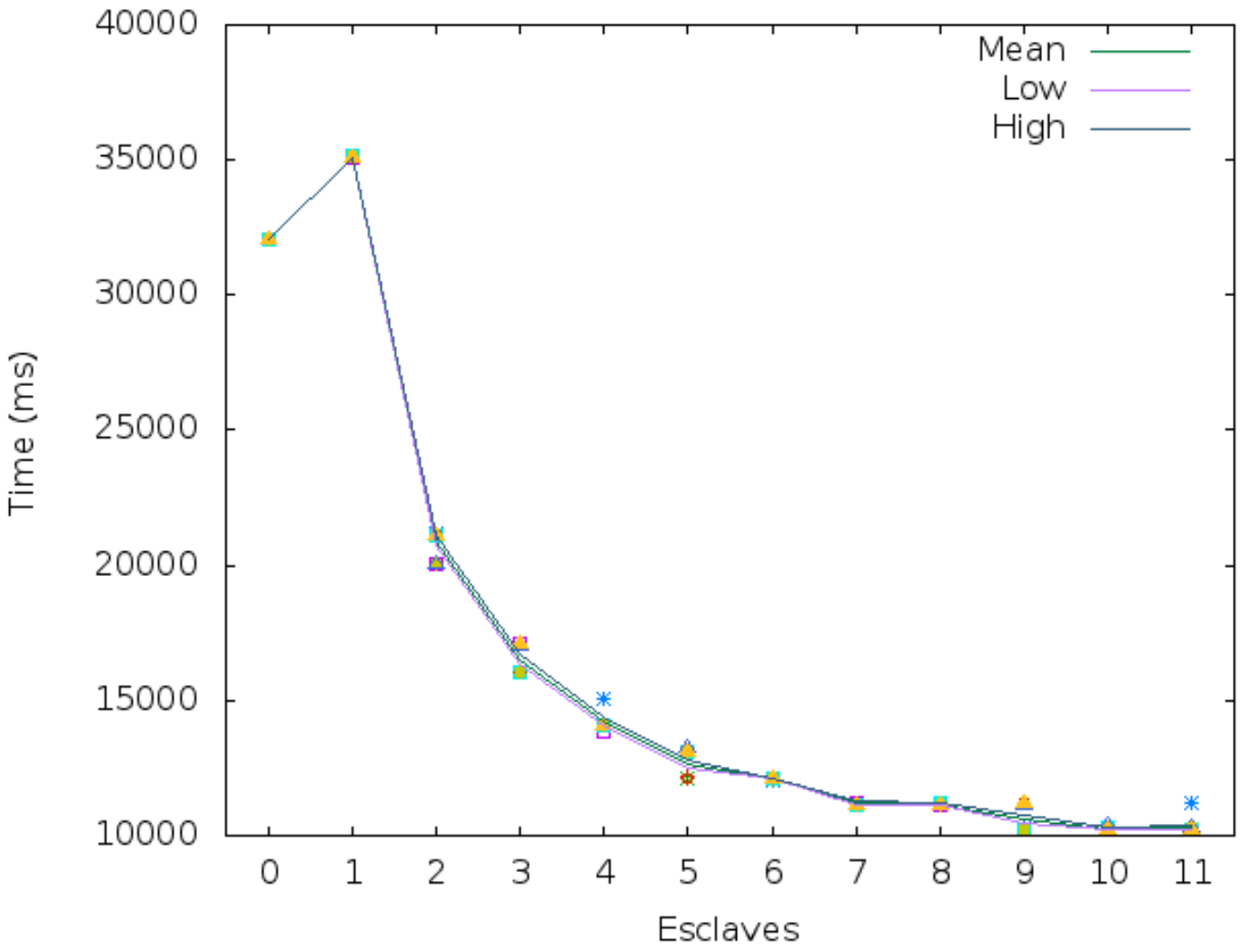

FIGURE 1 – Tree times

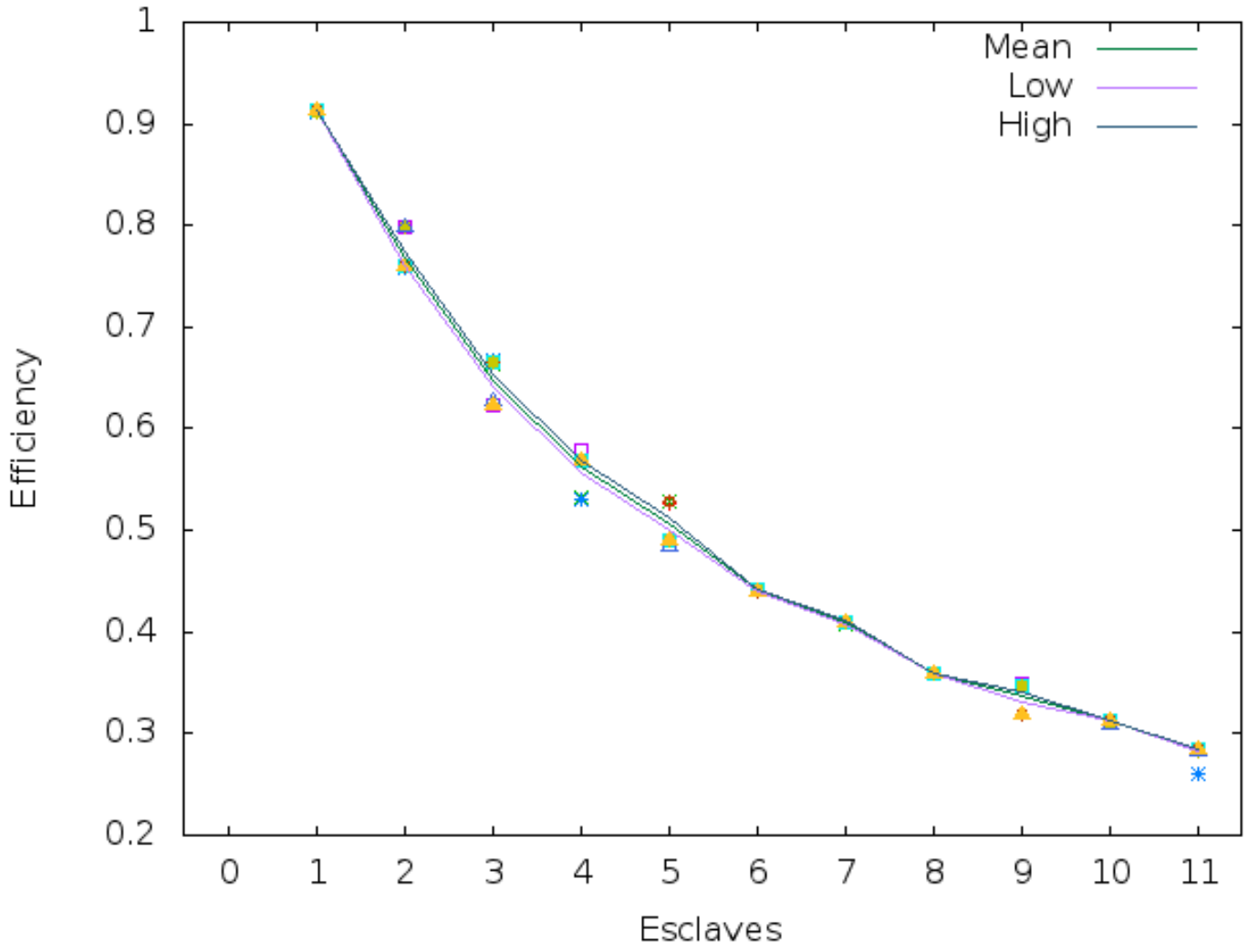

FIGURE 2 – Tree efficiencies

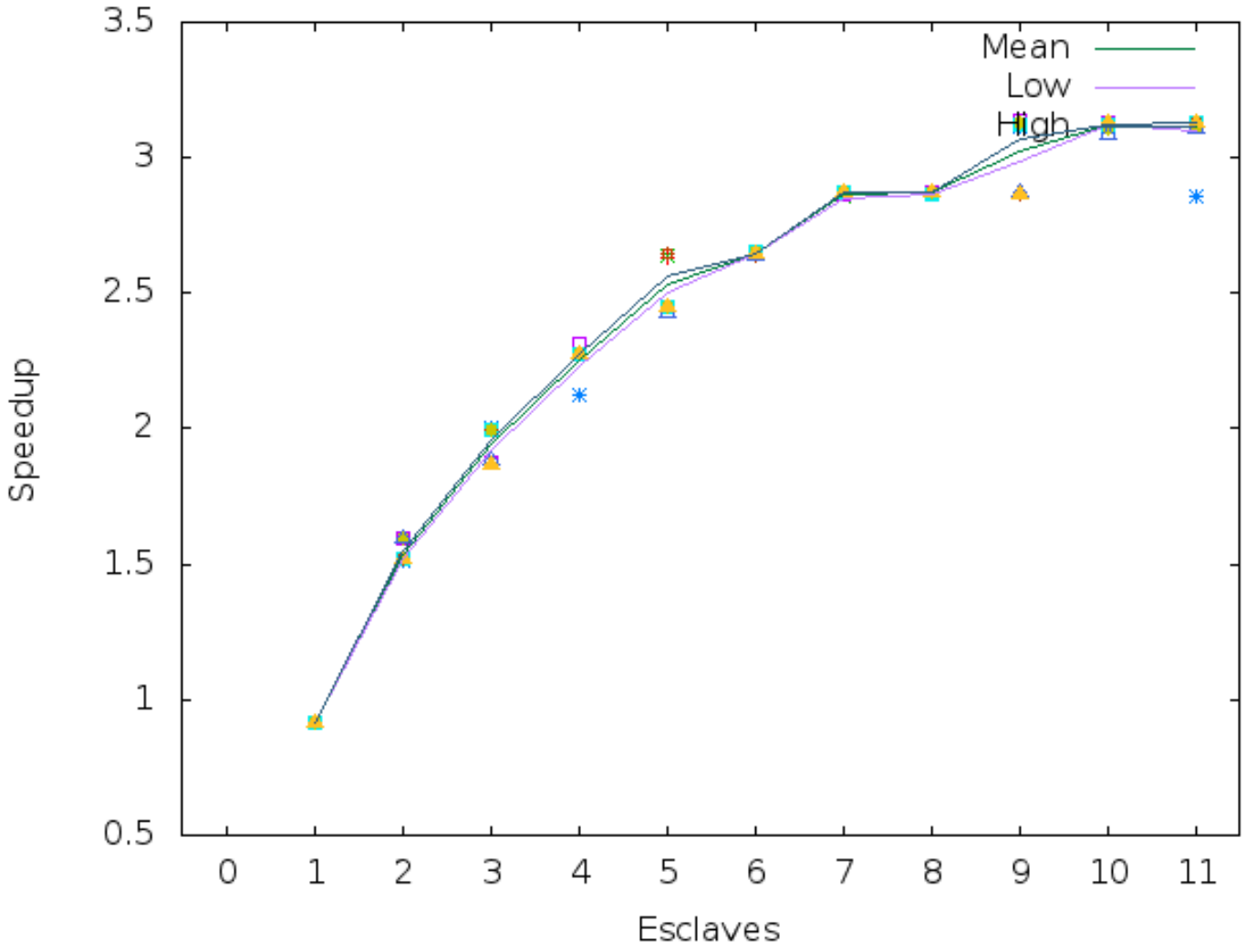

FIGURE 3 – Tree speedups

2.3.2 Makefile "Parallélisable"

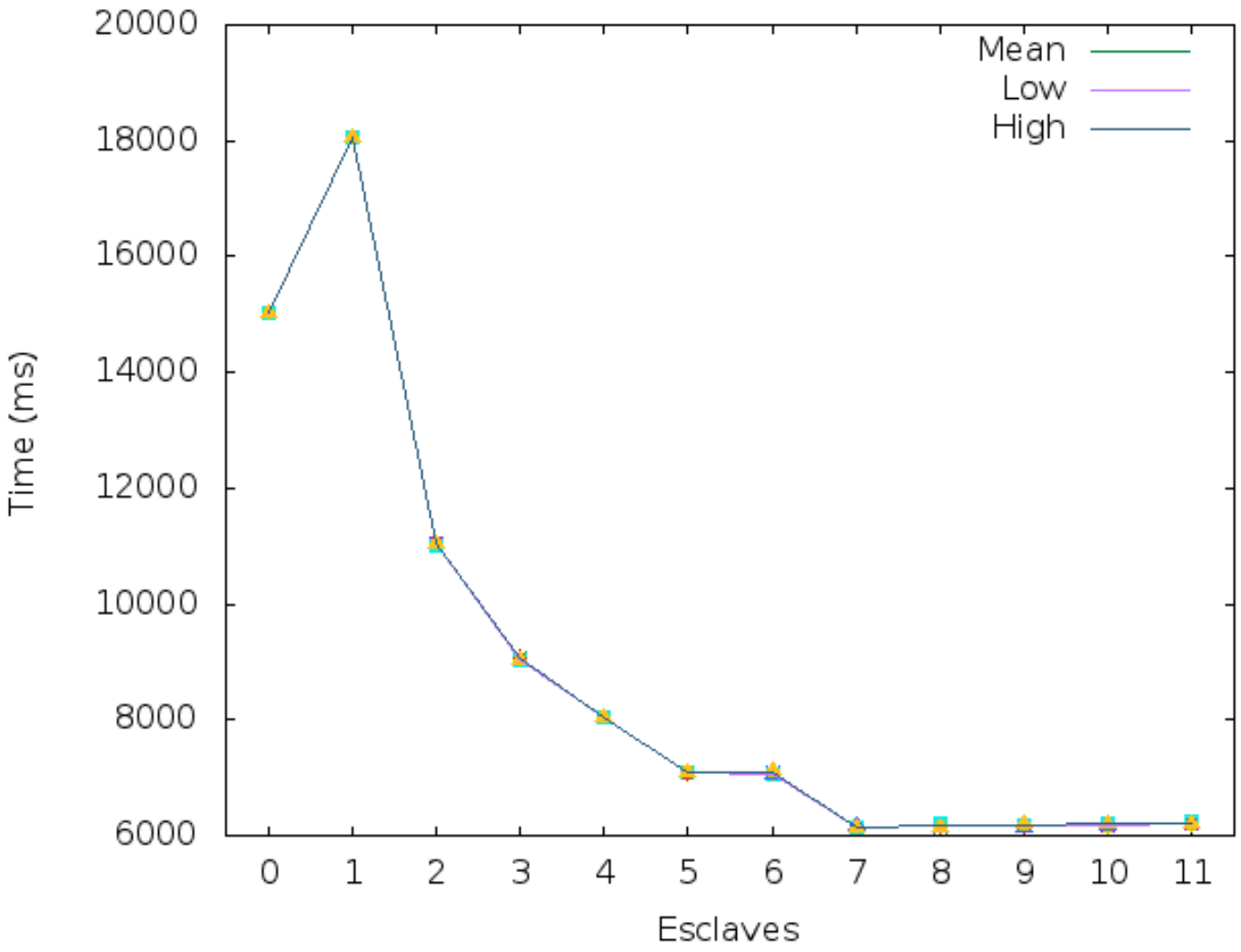

FIGURE 4 – Parallélisable temps d'exécution

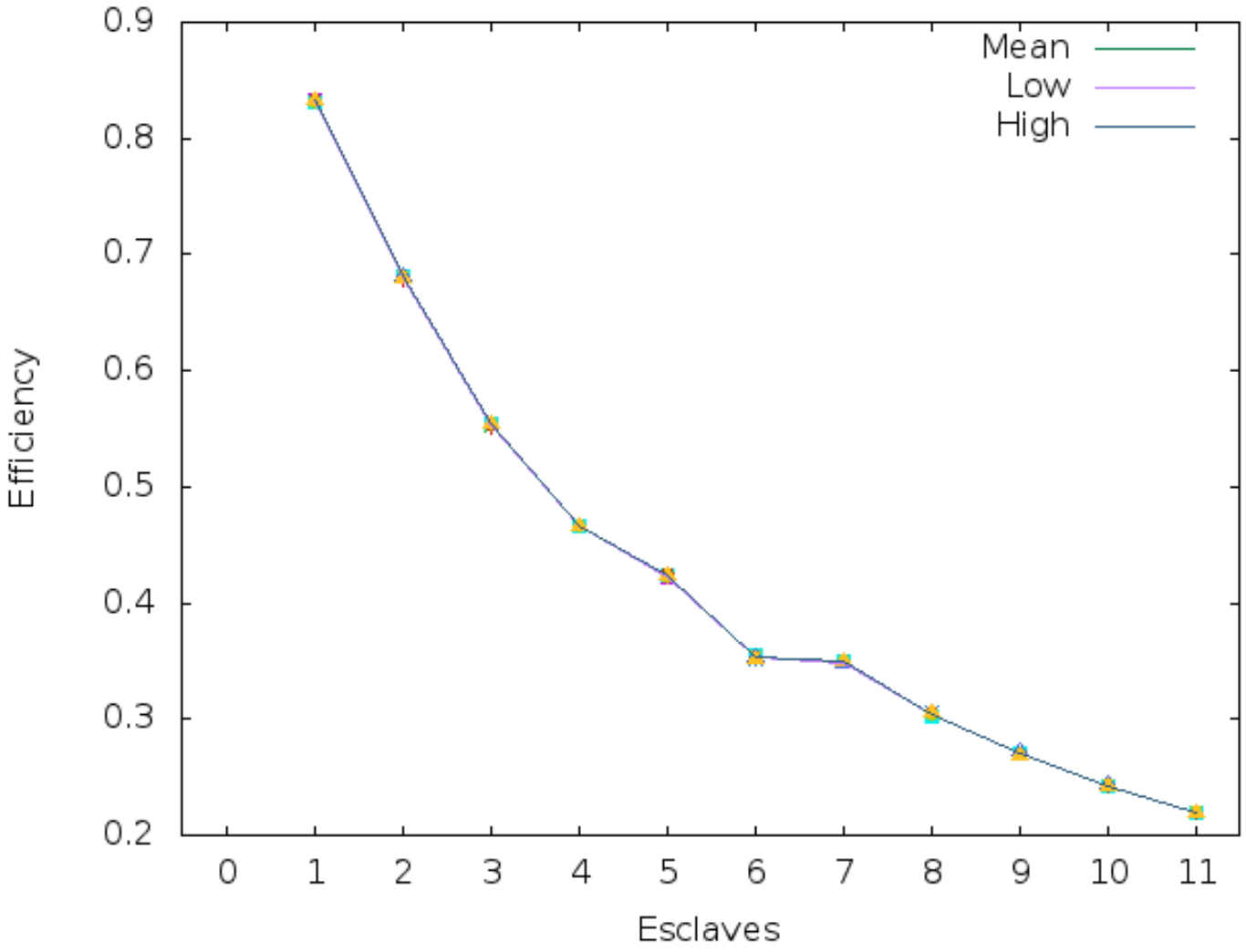

FIGURE 5 – Parallélisable Efficacité

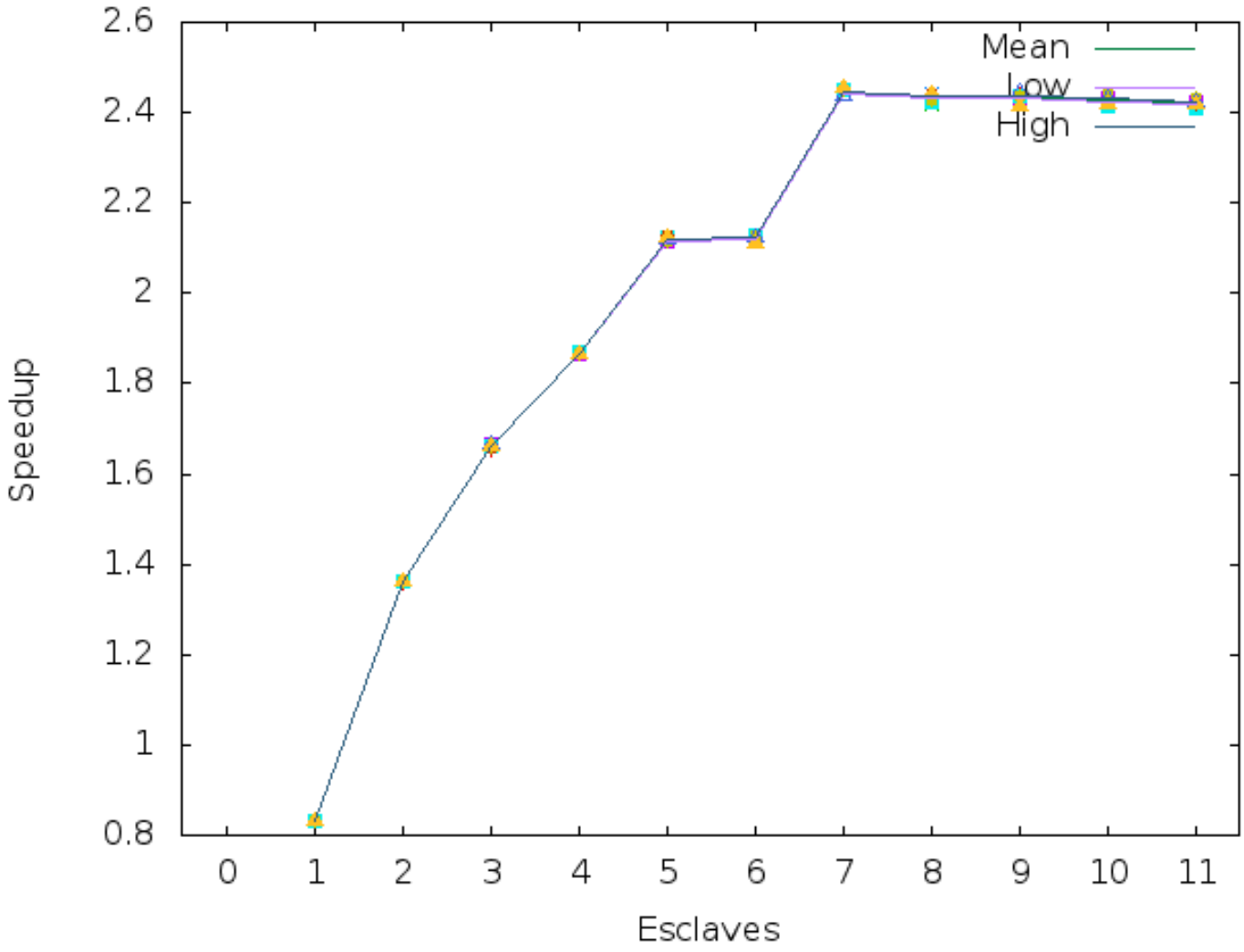

FIGURE 6 – Parallélisable accélération

2.3.3 Taux d'efficacité des slaves

Taux d'efficacité des slaves sur le Makefile "Parallélisable" (moyenne sur les 30 répétitions et sur les 11 slaves) : 99.4%

Taux d'efficacité des slaves sur le Makefile "Arbre" (moyenne sur les 30 répétitions et sur les 11 slaves) : 99.2%

Ces taux montrent qu'en moyenne, plus de 99% du temps des slaves est utilisé à directement travailler sur des tâches, et que les temps de communication ne représentent qu'une très légère part du temps total.

2.4 Analyse critique

L'utilisation de Go facilite grandement la gestion de la concurrence (création de threads, exclusion mutuelle...). De plus, le langage est bien documenté. En revanche, débuguer la concurrence est délicat (mais ceci n'est pas spécifique à Go RPC). Quant à RPC, le package rend la communication bi-directionnelle compliquée (car il nécessite de démarrer un serveur de chaque côté) : il n'est donc pas adapté à toutes les architecture (une alternative serait de l'associer à un système *Publish/Subscribe*).