Projet makefile distibués - GO gRPC

Garrenlus DE SOUZA
Guilherme FACCIN HUTH
Nabil ES-SALIHI
Youssef BENJELLOUN EL KBIBI

Contents

1 CHOIX RÉALISÉS

Pour l'implémentation de la version distribuée de GNU make, nous avons fait le choix d'utiliser le langage Go avec gRPC, ce qui est justifié par plusieurs raisons.

Tout d'abord, le modèle de concurrence robuste de Go, construit autour des goroutines et des canaux, s'aligne parfaitement avec les besoins d'un système distribué, permettant un traitement parallèle et une gestion des tâches efficaces. Ce support natif de la concurrence est crucial pour gérer de manière transparente plusieurs tâches sur différents serveurs.

Ensuite, la simplicité de Go et son système de typage fort contribuent au développement d'un code fiable et maintenable, ce qui est vital dans des environnements distribués complexes, ceci ainsi que le modèle de concurrence, ont fait que Go est devenu très populaire, ce qui nous a plus motivé à l'utiliser pour l'apprendre.

De plus, gRPC, un framework RPC universel open-source et haute performance, complète les capacités de Go en fournissant un moyen puissant de communication serveur-client. Il offre une large interopérabilité entre les langages, assurant l'extensibilité future du système. gRPC fournit également des fonctionnalités avancées comme le streaming bidirectionnel et l'équilibrage de charge intégré, qui sont bénéfiques pour un système de make distribué. Son utilisation des Protobuf pour la définition de l'interface assure une sérialisation des données efficace et fortement typée, réduisant la surcharge du réseau.

Dans l'ensemble, la combinaison des forces de Go en matière de concurrence et des capacités de communication efficaces de gRPC en fait un excellent choix pour développer une version distribuée de GNU Make, à la fois évolutive, efficace et robuste.

2 ARCHITECTURE

Dans ce projet, l'architecture a été utilisée sous la forme d'une distribution master-slave, comme le montre l'image ci-dessous : Cette configuration est conçue pour avoir un nœud maître (qui,

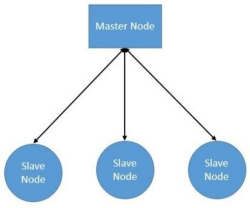

FIGURE 1: MASTER-SLAVES ARCHITECTURE

dans notre cas, est choisi par l'utilisateur de l'application), qui coordonne les tâches à distribuer aux esclaves.

3 LANCER L'APPLICATION

Afin de lancer l'application il suffit de se placer dans la racine du projet, remplacer dans copy_connect_g5k.s votre username et clé ssh, et lancer les commandes suivantes :

```
# On copie les fichiers vers grid5000
./copy_connect_g5k.sh $site # Remplacer par site de votre choix
ssh $site # site spécifié dans copy_connect_g5k.sh
./setup.sh # On installe les dépendences
oarsub -l nodes=$nodes -I # remplacer $nodes par le nombre de
   ↪ workers
cd makefiles
make $your_makefile
```

Par exemple si on veut lancer le makefile premier sur 4 worker nodes à nancy il suffit de faire :

```
./copy_connect_g5k.sh nancy
ssh nancy
./setup.sh
oarsub -l nodes=5 -I # 4 workers + 1 master
cd makefiles
make premier
```

4 NFSV3

Tous les systèmes évalués utilisaient NFS v3. Cela peut être vérifié en exécutant:

`/usr/sbin/nfsstat -m`

Ce qui à son tour présentera les points de montage pour NFS. L'approche présentée utilise le dossier `home`. On peut voir que ce dossier est monté en tant que point de montage NFS. Un exemple est donné ci-dessous:

```
gdesouza@paravance-8:~$ /usr/sbin/nfsstat -m
...
/home/gdesouza from nfs:/export/home/gdesouza
 Flags: rw,nosuid,nodev,relatime,vers=3,rsize=1048576,wsize
    ↪ =1048576,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=
    ↪ sys,mountaddr=172.16.111.24,mountvers=3,mountport=48021,
    ↪ mountproto=udp,local_lock=none,addr=172.16.111.24
```

Dans ce cas (mesures sur le site de *rennes*), la latence vers le serveur était en moyenne de 130 microssecondes, allant de 93 à 176 sur un ensemble de 35 échantillons. On peut effectuer de telles évaluations en utilisant `mtr` avec l'option `--tcp` activé et l'option `-c` avec le nombre souhaité.

4.1 WRITING

Afin d'évaluer la vitesse des écritures, un programme C a été développé. L'idée est de forcer chaque `block_size` kilo-octets écrit en mémoire à être vidé sur le périphérique de stockage, dans ce cas, à être validé sur le serveur NFS. Ceci peut être réalisé via l'appel de `fsync` dans le cas de NFS (`nfs-test.c`).

Ce programme a ensuite été exécuté à partir d'un script python (`run-file-evaluation.py`). La taille du fichier va jusqu'à 64 mégaoctets et la taille du bloc va de 1 Ko jusqu'à la taille du fichier lui-même. Dans ce dernier cas, il n'y a qu'un seul appel à `fsync`.

Les résultats sont les suivants pour 1, 2, 4 et 8 Mo en microssecondes lorsqu'il s'agit d'un seul appel à `fsync` effectué. Sans doute le moyen le plus rapide d'écrire le fichier en termes de surcharge.

TABLE 1: FILE SIZE AND WRITING LATENCY

alpha = 0.05	1 MB	2 MB	4 MB	8 MB
average_time	4462.00	7538.78	24877.58	7538.78
lower_bound	4462.62	7540.81	24877.58	13765.93
upper_bound	4461.38	7536.76	-24877.58	13762.13

4.2 RPC

À l'aide d'outils appropriés, il est possible d'évaluer le temps moyen nécessaire au serveur pour répondre aux requêtes RPC. L'outil en question est `nfsiostat` et il donne ces informations pour tous les points de montage utilisant nfs. Dans les expériences mentionnées ci-dessus, les valeurs (pour le temps pendant lequel le client NFS attend le kernel jusqu'à ce que la demande soit satisfaite) ont été mesurées à plusieurs reprises toutes les 10s et ont tourné autour de $0,310$ms pour les lectures et de $8,708$ms pour les écritures. En ce qui concerne la latence de la réponse au RPC elle s'élève à $0,241$ms pour les lectures et à $1,295$ ms pour les écritures (RTT).

5 PING-PONG

5.1 GÉNÉRATION DES MÉTRIQUES

Commençons d'abord par tester l'infrastructure sur la laquelle l'application tourne en utilisant Go et gRPC. Pour celà on peut considérer un setup simple : Un serveur qui ouvre un port RPC, et un client qui se connecte sur ce port et envoie des messages. Ceci est implémenté dans les dossiers pingpong/server et pingpong/client.

Lorsque le client se connecte au serveur il envoie 32 fois un message de taille 1 byte, et mesure les temps d'aller-retour à chaque fois. Ces temps d'aller-retour sont stockés dans un tableau sur lequel on calcul la moyenne : le résultat divisé par 2 correspond à la latence. Maintenant qu'on a la latence on peut l'utiliser ainsi que le tableau précédent pour calculer l'interval de confiance.

Après celà on doit calculer le débit ainsi que la bande passante, on va donc envoyer plusieurs fois des messages de différentes tailles et connaissant la latence on pourra donc calculer le débit, et connaissant l'interval de confiance on pourra calculer la bande passante.

5.2 GÉNÉRATION DE COURBES

Le client précédent écrit tout ce qu'il mesure dans la sortie standard : les temps d'aller-retour, la latence, le débit, la taille des messages... On a donc écrit un script python qui va parser cette sortie afin de tirer les différentes métriques, notamment les différents temps d'aller-retour et les débits pour chaque taille de message. Ensuite en utilsant la bibliothèque matplotlib on génère les différentes courbes.

Pour lancer le pingpong ainsi que la génération des courbes il suffit donc d'ouvrir 2 terminaux (ou d'être sur 2 machines), se placer dans le dossier pingpong et lancer sur l'un des terminaux (ou machine) la commande `go run server/main.go`, et puis sur l'autre terminal (ou machine) `go run client/main.go 2>&1 | python metrics.py`

5.3 RÉSULTATS OBTENUS

Les résultats dépendent beaucoup des sites et des machines sur lesquels le pingpong a été lancé, ceci dit les résultats et courbes présentés ici peuvent être différents.

6 MODÉLISATION DU TEMPS D'EXÉCUTION

Le travail en question a été développé pour organiser les tâches sous la forme d'un arbre, de manière à résoudre le problème en traitant les tâches de la feuille à la racine. Ainsi, pour modéliser le temps d'exécution du programme, il est nécessaire de prendre en compte le niveau de profondeur de l'arbre des tâches.

Toute l'opération commence par le nœud maître, chargé de traiter les informations à envoyer aux nœuds esclaves. On appellera ce temps le temps t_{init}.

Après avoir traité le fichier makefile initial et décomposé le problème en tâches, le nœud maître envoie à ses nœuds esclaves des messages indiquant leur tâche respective. Comme les messages sont de taille très petite, quelques simplifications seront effectuées afin de ne prendre en compte que la latence pour le transfert des messages ($\mathcal{L}$).

Si les tâches reçues par les nœuds esclaves sont des tâches feuilles, c'est-à-dire s'il n'y a qu'un seul niveau de profondeur dans l'arbre, alors les tâches sont exécutées. Cela donne alors: $max(t_{exec} + t_{nfssave})$.

Où t_{exec} est le temps d'exécution de l'esclave, $t_{nfssave}$ est le temps nécessaire pour sauvegarder le résultat de l'opération et $t^m_{nfsread}$ est le temps nécessaire pour la lecture par le maître.

Pour simplifier les calculs, supposons que les tâches sont réparties de manière homogène entre les nœuds esclaves, de sorte que le temps total soit comptabilisé comme suit :

$$\frac{1}{n_{slaves}} \sum_{j=1}^{m} t^i_{exec} + t^i_{nfssave}$$

Et à la fin de l'exécution du programme, nous avons le temps nécessaire pour regrouper toutes les informations et calculer le résultat du programme : t_{mfin}.

Ansi:

$$\underbrace{t^m_{init} + \mathcal{L} + t^m_{nfsread} + t^m_{mfin}}_{\text{Les tâches sérializables}} + \overbrace{\frac{1}{n_{slaves}} \sum_{j=1}^{m} t^i_{exec} + t^i_{nfssave}}^{\text{Les tâches parallélizable}}$$

Où :

- t^i représente le temps utilisé par les nœuds esclaves numéro i ;
- t^m est le temps utilisé par le nœud maître ;
- n_{slaves} est le nombre de nœuds esclaves ;
- m est le nombre de sous-tâches.

Ceci est simplement une approximation, en supposant que le temps d'exécution des tâches sur chaque nœud a une complexité approximativement similaire et ne dépend pas du matériel sur lequel il s'exécute, de sorte que le temps d'exécution est approximativement le même. Selon les auteurs, une bonne approximation pour la partie qui contient l'opérateur "max" serait d'attribuer la valeur du temps moyen plus un intervalle de confiance.

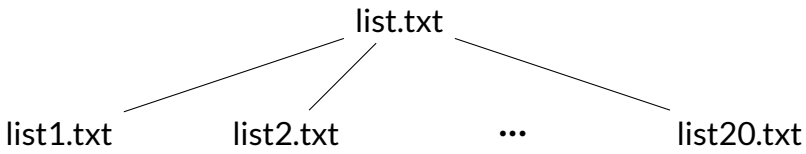

FIGURE 2: DISTRIBUTION DES TÂCHES PREMIER TEST

6.1 PREMIER TEST

Le premier test est caractérisé par un seul niveau dans l'arbre des tâches, où les dépendances peuvent être modélisées comme indiqué dans l'image ci-dessous :

Après avoir lancé le programme sur une seule machine, nous obtenons la courbe qui est dans l'image suivante.

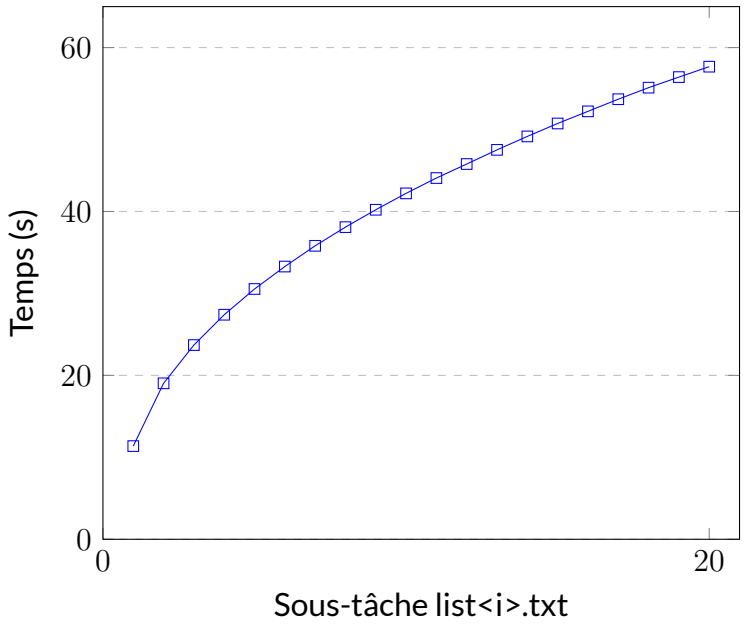

FIGURE 3: TEMPS MOYENNE POUR EXECUTER SOUS-TÂCHE DU TEST PREMIER DANS LA MEMME MACHINE

Nous pouvons observer la complexité croissante à partir du temps nécessaire pour exécuter chacune des 20 sous-tâches du programme.

La somme des temps de chaque sous-tâche était égale à : 814,03 secondes.

Lors de la mesure du temps total d'exécution du programme, on a observé : 814,23 secondes.

De cette manière, nous pouvons dire que la part des tâches sérialisables est négligeable.

Ce qui résulte en temps d'exécution sera le temps exécuté par le programme sur une seule machine divisé par le nombre de nœuds. (Si les machine sont identiques)

6.2 MATRIX TEST

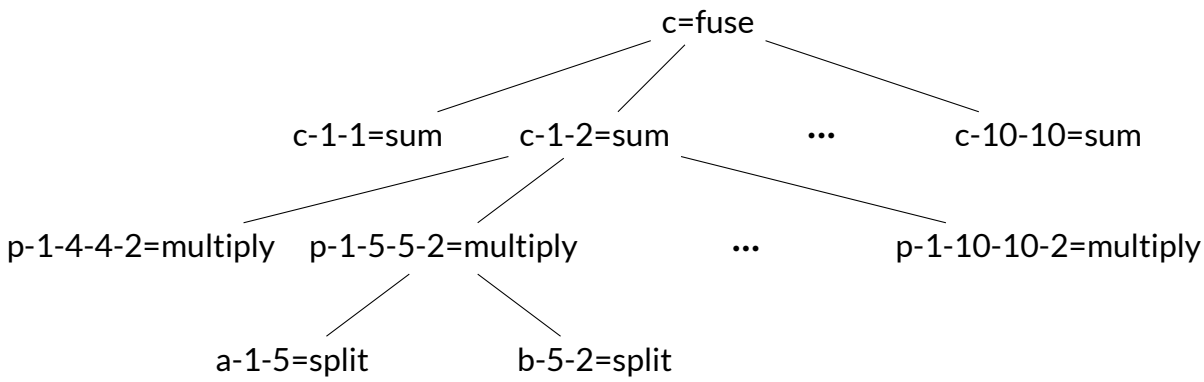

FIGURE 4: DISTRIBUTION DES TÂCHES MATRIX TEST

6.3 SECOND TEST

Ici on va lancer le make distribué pour premier et matrix (fournis) 3 fois chacun, en augmentant le nombre de worker nodes de 1 à 20. A chaque exécution on mesure le temps, et on le stocke dans un fichier avec le nombre de workers utilisés. Pour un nombre de noeuds donné on calcule les moyenne des temps mesurés, et on génère les courbes d'évolution du temps d'exécution en fonction du nombre de workers. On obtient les courbes quivantes :

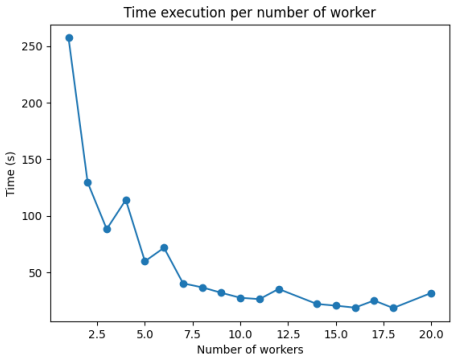

FIGURE 5: PREMIER

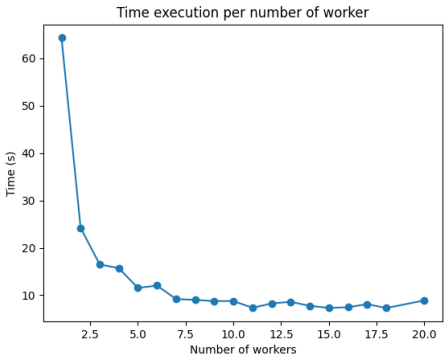

FIGURE 6: MATRIX

Ces résultats sont logiques du fait que plus on a de workers, plus on a de puissance de calcul et donc on est plus rapide. La décroissance est moins rapide après à cause du surcoût payé par nfs quand on augmente beaucoup le nombre de workers.