Compte rendu du projet make

Equipe 10

Boudouin Philippe
Danet Nicolas
Ferrand Pierre
Gouttefarde Léo

Vendredi 25 Novembre 2016

Table des matières

1 Présentation du projet **2**

1.1 Langage / Framework 2

1.2 Architecture 2

1.3 Algorithme principal 2

1.4 Extensions 2

1.5 Installation 2

1.6 Déploiement sur Grid'5000 3

2 Tests de performance **6**

2.1 Tests mono-coeurs 6

2.2 Tests multi-coeurs 10

2.3 Analyse des graphes 11

1 Présentation du projet

1.1 Langage / Framework

Le projet a été réalisé à l'aide du framework Charm++, qui s'utilise en C++. Il s'agit d'un framework très complet du même type que Qt mais pour permettre de créer des applications distribuées et multi-coeurs.

Pour cela le framework est basé sur une gestion de l'application par objets distribués et méthodes d'objet distantes, dont la spécification doit être faite dans un format de fichier d'interface dédié (fichiers .ci).

Le framework utilise ainsi un compilateur spécifique, et fonctionne sans aucune option spécifique sur les machines de l'Ensimag comme sur Grid'5000.

1.2 Architecture

Le projet s'organise autour de différentes classes :

- `Parser` : Effectue le parsing du Makefile
- `Master` : Classe de l'objet distribué principal, le maître
  Il gère des esclaves, leur transmet des tâches, et récupère les résultats
- `Slave` : Classe des objets distribués d'esclaves
  Les esclaves reçoivent des tâches du maître qu'ils doivent exécuter, puis en revoyer le résultat
- `Node` : Classe des noeuds du graphe de dépendances tiré d'un Makefile
  Cette classe permet de gérer les différentes opérations faites sur les noeuds, elle permet aussi de créer le graphe des dépendances à partir des éléments parsés
- `Job` : Cette classe décrit les tâches envoyées à distance aux esclaves
- `File` : Cette classe décrit les fichier à envoyer entre objets distribués, elle définit leur lecture / écriture / transmission

1.3 Algorithme principal

Le maître envoie des tâches aux esclaves, les esclaves une fois terminé renvoient leur résultat au maître qui regarde alors si une nouvelle tâche à exécuter est disponibles (donc dont toutes les dépendances sont créées). Nous n'utilisons pas NFS afin de supporter les machines qui n'ont pas NFS.

1.4 Extensions

En plus des fonctionnalités de base, notre programme de make permet d'utiliser plusieurs CPUs par machine et minimise les envois de dépendances aux esclaves (on ne renvoie jamais une dépendance précédemment envoyée).

De plus la dernière tâche est effectuée sur le maître pour en accélérer l'exécution (plus besoin de distribuer de tâches à ce moment là).

1.5 Installation

Pour installer Charm++, il faut en récupérer l'archive, la décompresser puis compiler :

```
cd ~
```

```
wget http://charm.cs.illinois.edu/distrib/charm-6.7.1.tar.gz
tar -xvzf charm-6.7.1.tar.gz
cd ~/charm-6.7.1
./build charm++ netlrts-linux-x86_64 --with-production -j8
```

Sur Grid'5000, nous stockons ce dossier dans répertoire home et spécifions ce répertoire d'installation dans le Makefile.

1.6 Déploiement sur Grid'5000

Le code de déploiement est écrit en bash avec les différentes étapes décrites dans le fichier `scripts/deploy.sh`.

Pour l'exécution distribuée, Charm++ comporte un équivalent à mpiexec : charmrun.

1.6.1 Etape 1

Connexion, réservation et installation de 10 noeuds pour 1h par exemple (on installe notamment blender et ffmpeg) :

```
ssh <login>@access.grid5000.fr
ssh <site>

oarsub -I -l nodes=10,walltime=1 -t deploy
kadeploy3 -f $OAR_NODE_FILE -e jessie-x64-std -k
```

1.6.2 Etape 2

Préparation des noeuds et installation de Charm++ :

```
# runs a script remotely
remote_run()
{
  if [[ $# -ge 2 && -f "$2" ]]; then
    ssh -oStrictHostKeyChecking=no root@$1 'bash -s' < "$2"
  fi
}

# Charm++ nodelist file (regenerate it after each oarsub node allocation)
NODELIST=~/nodelist

sort -u $OAR_NODEFILE > nodes

# Generates Charm++ nodelist file
# each line from $OAR_NODEFILE = 1 cpu entry, usually 8 / node
awk '$0="\thost "$0" ++cpus 8"' nodes > tmp
echo "group main" > $NODELIST
cat tmp >> $NODELIST
rm tmp
```

```
echo -e "apt-get update\napt-get -y install blender ffmpeg ImageMagick ncftp" > task.sh
echo -e "nohup sh ~/task.sh &> out.txt &" > runTask.sh

SERVS=$(cat nodes)
SSH=$(tail -n 1 nodes)

# for each node
for SERV in $SERVS; do

  scp ~/.ssh/id_rsa root@$SERV:~/.ssh/id_rsa
  scp ~/.ssh/id_rsa.pub root@$SERV:~/.ssh/id_rsa.pub

  scp $NODELIST root@$SERV:~
  scp nodes root@$SERV:~
  scp task.sh root@$SERV:~

  remote_run $SERV runTask.sh

done

ssh root@$SSH

cd ~
wget http://charm.cs.illinois.edu/distrib/charm-6.7.1.tar.gz
tar -xvzf charm-6.7.1.tar.gz
cd ~/charm-6.7.1
./build charm++ netlrts-linux-x86_64 --with-production -j8
```

1.6.3 Etape 3

Içi il faut télécharger l'archive de notre application dans un dossier et la décompresser dans le dossier home, puis changer les identifiants FTP à la fin du fichier test/bench.sh si l'on souhaite que les résultats soient zippés sur un FTP à la fin.

1.6.4 Etape 4

Il s'agit maintenant de compiler et de finaliser les noeuds avec les instructions suivantes :

```
cd ~/make/src

# Fix Charm++ path
tail -n +4 Makefile >> Makefile_
echo "CPATH=~/charm-6.7.1" > Makefile
cat Makefile_ >> Makefile

make -j8


cd ~/make/sujet/makefiles/premier
```

```
gcc premier.c -lm -o premier
cd ~

# for each node
for SERV in $(cat ~/nodes); do
  scp -o StrictHostKeyChecking=no -rp ~/make root@$SERV:~
done
```

1.6.5 Etape 5

Finalement pour lancer le benchmark de test il suffit d'utiliser l'instruction suivante :

```
cd ~/make
nohup ./test/bench.sh &> bench.log &
```

1.6.6 Etape 5 (mode manuel)

Pour exécuter le programme sans le script de benchmarking, il faut se mettre dans le dossier du Makefile à tester puis appeler le programme comme ceci :

```
~/make/charmrun ++nodelist ~/nodelist ++ppn <Nombre de processus par noeud> \
++p <Nombre de processus total> ~/make/src/Make <FichierMakefile>
```

Le fichier `~/nodelist` contient la liste des noeuds disponibles selon la syntaxe Charm++ (créé en étape 2).

2 Tests de performance

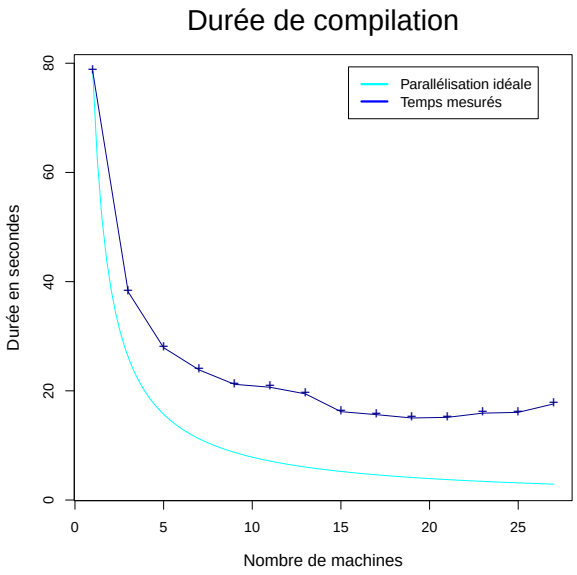
Les tests suivants ont été effectués sur les machines du cluster Edel de Grenoble, avec un maximum de 42 machines. Les caractéristiques de ces machines sont les suivantes :

— 2 CPUs Intel Xeon E5520
— 4 coeurs par CPU
— 24 GB de RAM
— 119 GB d'espace disque SSD

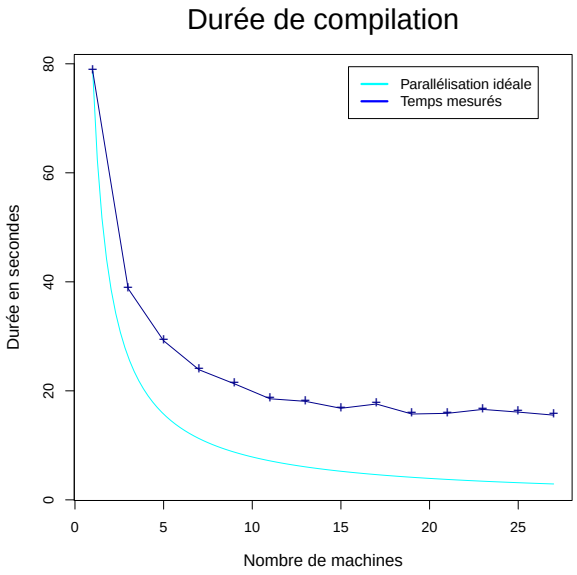
2.1 Tests mono-coeurs

2.1.1 Blender 2.49

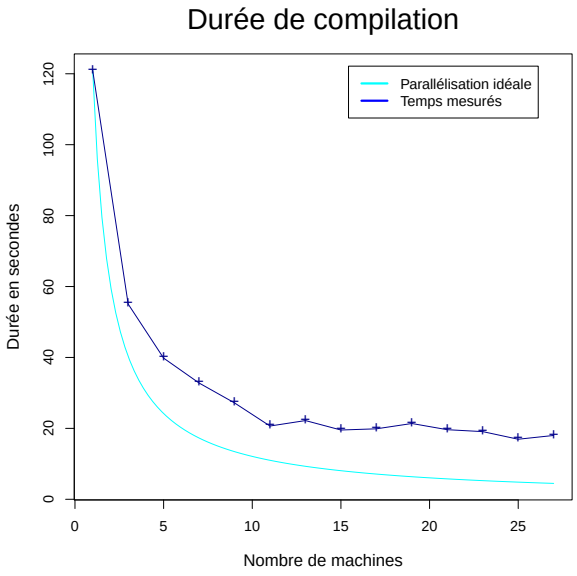
Makefile



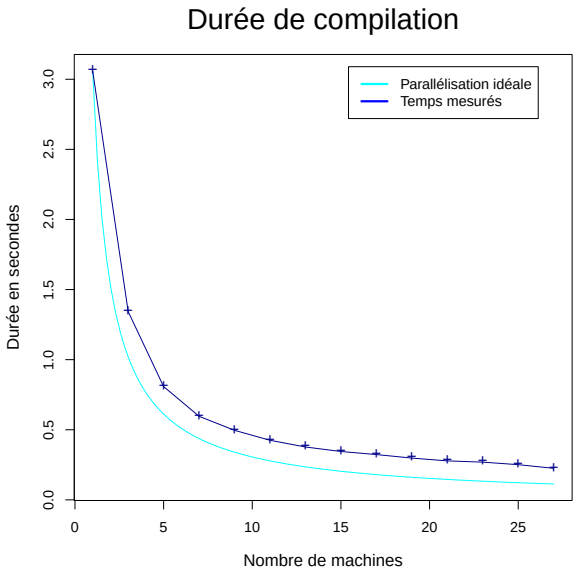
Makefile-recurse



2.1.2 Blender 2.59

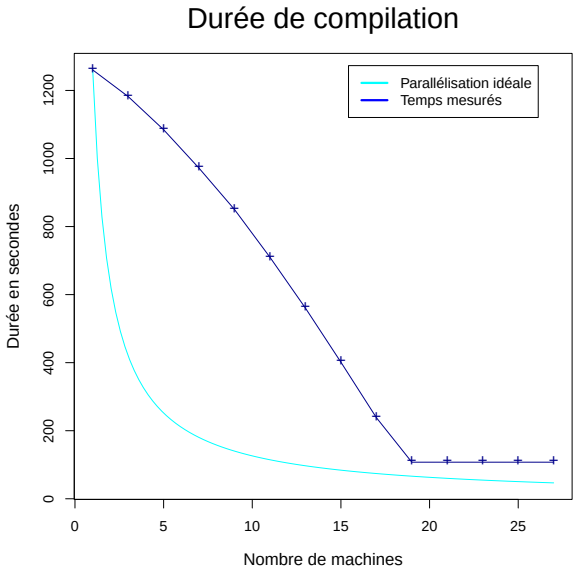


2.1.3 Matrix

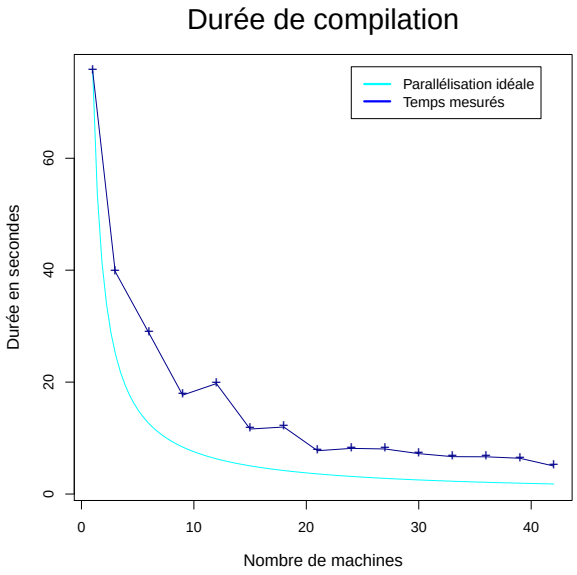


2.1.4 Premier

Makefile



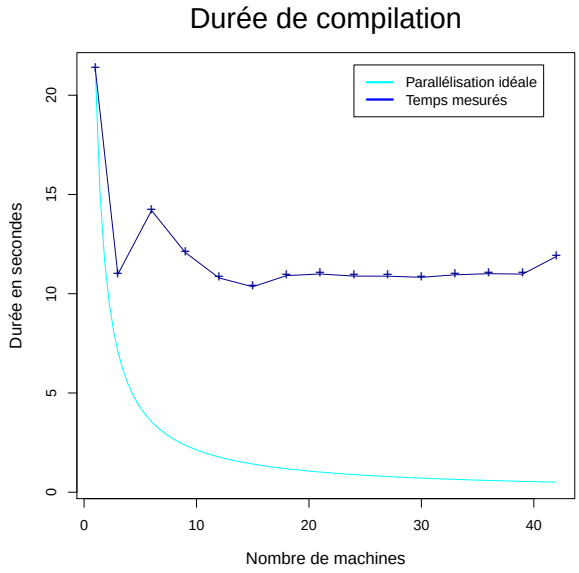
Makefile-small

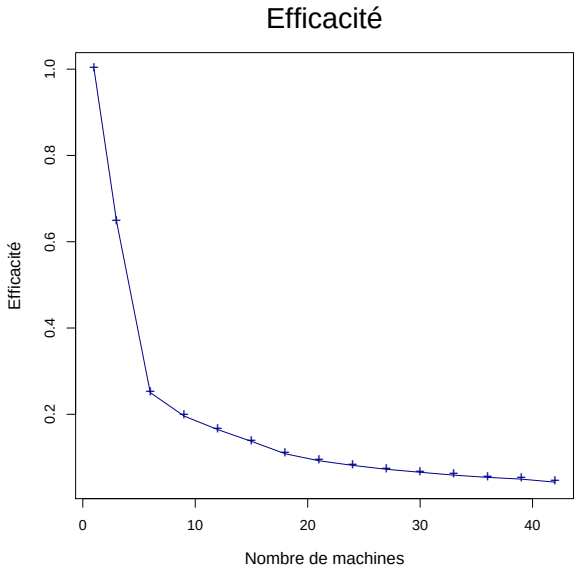


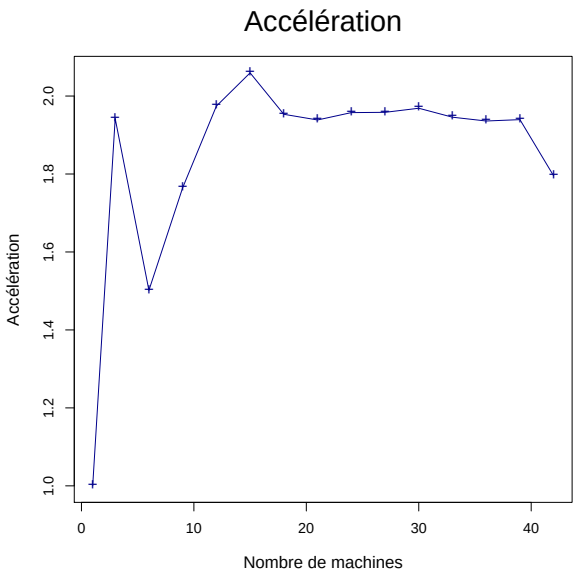
2.2 Tests multi-coeurs

Voici quelques exemples des performances bien meilleures obtenues en exploitant chacun des 8 coeurs (2 x 4 coeurs) disponibles par machine. Nous avons aussi rajouté les courbes d'efficacité et d'accélération pour ces deux exemples.

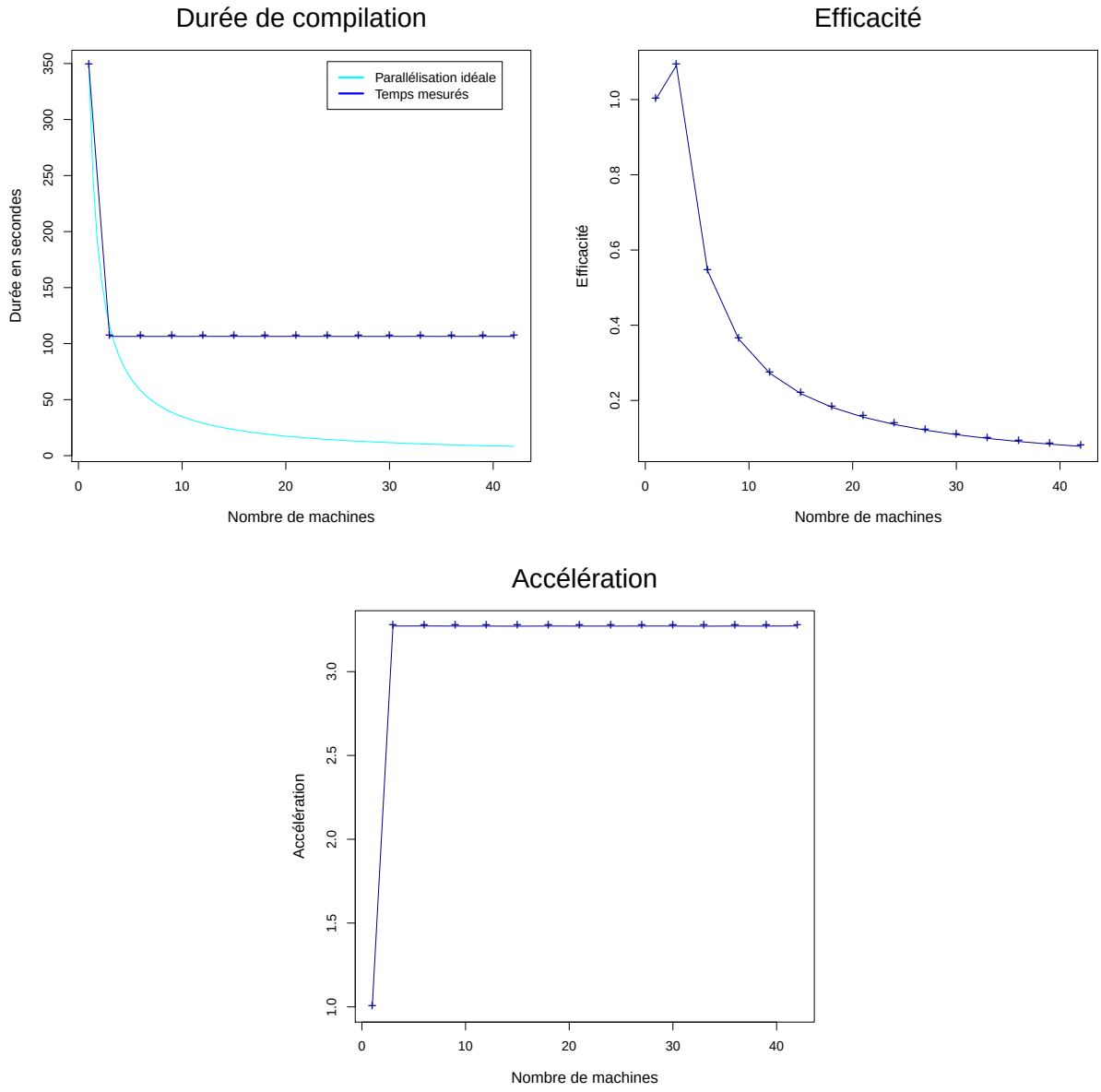
2.2.1 Blender 2.49 : Makefile







2.2.2 Premier : Makefile



2.3 Analyse des graphes

Les courbes expérimentales obtenues semblent être plutôt homogènes. Quelques points semblent néanmoins étranges puisqu'ils induisent une augmentation pour un point lorsque l'on attend une diminution. Cela peut s'expliquer par l'incertitude des mesures expérimentales. Cela peut également s'expliquer par l'une des extensions que nous avons réalisées : lorsque l'on envoie une tache à un esclave, on ne renvoie pas les dépendances qu'on lui a déjà précédemment envoyé. En rajoutant des machines on diminue la probabilité de renvoyer à un esclave une tache pour laquelle il a déjà reçu des dépendances ce qui augmente le temps réseau (cette optimisation est moins efficace dans ces cas-là, mais nous obtenons tout de même de meilleurs résultats qu'en ne l'implémentant pas).

On remarque que les courbes expérimentales des temps d'exécution ont la même allure que les courbes théoriques (obtenues en divisant le temps obtenu avec une machine par le nombre de machines). Cependant, de gros écarts apparaissent et croissent avec le nombre de machines. Cela s'explique par

le besoin d'envoyer les fichiers sur le réseau et donc par le fait de rajouter du temps d'envoi et de traitement à la fois des fichiers sources, mais aussi des réponses. On remarque que les temps finissent par stagner, souvent aux alentours de 20 machines, car passé ce nombre, il n'y a plus assez de tâches à distribuer et rajouter des esclaves ne fait plus gagner de temps puisque certains esclaves se retrouvent sans tâche à effectuer. En revanche, à partir d'une quarantaine de machines, la tendance est que les temps recommencent à augmenter, car il y a de nombreux esclaves qui demandent des temps de traitement et des temps d'envois réseaux importants (car l'optimisation qui consiste à ne pas renvoyer une dépendance déjà envoyée à un esclave ne marche plus).

Les courbes d'efficacité nous semblent cohérentes, puisqu'elles sont toujours inférieurs à 1 et qu'elles sont décroissantes. Les courbes d'accélérations semblent également globalement cohérentes, puisque rajouter des machines permet d'augmenter l'accélération (représentant le temps gagné en distribuant le makefile plutôt qu'en l'exécutant en séquentiel). Les quelques irrégularités étant dues aux irrégularités des mesures.

On remarque que l'utilisation de plusieurs coeurs par machine permet de gagner énormément de temps : on gagne en moyenne un facteur 2. Mais cela ne nous étonne pas car c'est ce que nous attendions.

Nous remarquons que notre makefile permet de gagner énormément de temps en distribuant les tâches. En effet, entre la distribution des tâches, et l'utilisation de plusieurs coeurs, on a besoin de jusqu'à 6 fois moins de temps comparé à une exécution séquentielle par exemple pour l'exemple Premier.