Projet de multicore programming (X8II070)

Alexis BONNIN Jocelin CAILLOUX

Avril 2017

Résumé

Vous trouverez dans ce document un rapport du projet de "Multicore Programming". Il détaille les performances obtenues avec les différentes implémentations de version parallèle de l'algorithme de branch-and-bound permettant de calculer le minimum d'une fonction binaire réelle.

Table des matières

1 Introduction

Ce projet s'inscrit dans le module de "Multicore Programming". Il consiste à paralléliser un algorithme de "branch-and-bound" utilisant l'arithmétique d'intervalle et permettant de trouver le minimum d'une fonction binaire réelle à partir d'un ensemble de domaines. Pour cela, deux versions de l'algorithme ont été implémentées : la première utilisant seulement MPI et la deuxième, utilisant à la fois MPI et OpenMP.

Le projet est disponible sur github à l'adresse : https://github.com/Asteip/GoldsteinBB.git. Pour installer le projet, il suffit d'exécuter la commande *make* à la racine du répertoire (il sera peut-être nécessaire de modifier le fichier makefile pour les dépendances). Le programme peut ensuite être exécuté avec les commandes suivantes :

— Version séquentielle :

```
./optimization-seq
```

— Version avec MPI :

```
mpirun -np 2 --hostfile ./hostfile ./optimization-mpi
```

— Version avec MPI et OpenMP :

```
mpirun -np 2 --hostfile ./hostfile ./optimization-par
```

2 Implémentation avec MPI

2.1 Choix d'implémentation

L'implémentation de l'algorithme avec MPI se trouve dans le fichier *optimization-mpi.cpp*.

Tout d'abord, la saisie de la fonction et de la précision se fait sur la machine de rang 0. Ces deux informations sont ensuite envoyées à toutes les machines en broadcast (*MPI_Bcast()*) depuis la machine de rang 0.

Dans notre solution, nous avons choisi de diviser les intervalles X et Y de la fonction en autant de sous-intervalles qu'il y a de machines disponibles (numprocs). Seuls les sous-intervalles de X sont envoyés sur le réseau via la fonction *MPI_Scatter()* (chaque machine traite donc un sous-intervalle de X). En effet, chaque machine doit connaître tous les intervalles de Y afin de vérifier tous les couples (x,y) possibles lors de l'appel à la fonction *minimize()*. L'objectif de séparer à la fois X et Y et non X seul est de pouvoir traiter des domaines en cube et non en pavé (qui ne sont pas adaptés à l'heuristique). C'est donc la machine de rang 0 qui est chargée de découper l'intervalle X et chaque machine fait ensuite son propre découpage de Y. Ici une autre solution aurait pu être de découper l'intervalle Y dans la machine de rang 0 et d'envoyer le découpage à chaque machine (tout comme pour l'intervalle X). Nous préférons la première solution puisque nous pensons que les temps de transfert sur un réseau des sous-intervalles de Y sont plus élevés qu'un simple découpage sur chaque machine.

Une fois toutes les machines initialisées, celles-ci exécutent l'algorithme de minimisation (*minimize()*) pour leur sous-intervalle de X et pour tous les sous-intervalles de Y. Le résultat final est ensuite obtenu en faisant une réduction sur le minimum calculé par chaque machine (*MPI_ Reduce()*). Pour finir, l'affichage du résultat se fait sur la machine de rang 0.

2.2 Tableaux des performances

Les jeux de tests suivants ont été effectués pour chaque fonction sur des nombres différents de machines. Les temps affichés sont en secondes.

	Nombre de machines				
Précision	1	2	3	4	5
$1e^{-1}$	$1.0e^{-2}$	$2.1e^{-3}$	$2.9e^{-3}$	$1.9e^{-3}$	$1e^{-2}$
$1e^{-2}$	$2.8e^{-2}$	$1.8e^{-2}$	$2.3e^{-2}$	$1.5e^{-2}$	$2.0e{-}2$
$1e^{-3}$	$2.9e^{-1}$	$2.7e^{-1}$	$1.9e^{-1}$	$2.4e^{-1}$	$1.5e^{-1}$
$1e^{-4}$	2.3	2.2	3.0	1.9	2.3
$1e^{-5}$	18	17	24	15	19

FIGURE 1 – Temps d'exécution de l'algorithme (en secondes) pour la fonction booth

	Nombre de machines				
Précision	1	2	3	4	5
$1e^{-1}$	$5.0e^{-2}$	$3.5e^{-3}$	$1.5e^{-3}$	$2.0e^{-3}$	$6.0e^{-2}$
$1e^{-2}$	$2.8e^{-2}$	$2.1e^{-2}$	$2.3e^{-2}$	$1.7e^{-2}$	$5.0e{-}2$
$1e^{-3}$	$2.3e^{-1}$	$3.4e^{-1}$	$1.6e^{-1}$	$3.4e^{-1}$	$2.5e^{-1}$
$1e^{-4}$	2.8	3.9	1.3	2.9	3.0
$1e^{-5}$	23	28	21	21	25

FIGURE 2 – Temps d'exécution de l'algorithme (en secondes) pour la fonction beale

	Nombre de machines				
Précision	1	2	3	4	5
$1e^{-1}$	$1e^{-2}$	$5.0e^{-3}$	$3.0e^{-3}$	$3.5e^{-3}$	$5.0e^{-3}$
$1e^{-2}$	$1.1e^{-1}$	$6.0e^{-2}$	$7.5e^{-2}$	$3.8e^{-2}$	$8.0e^{-2}$
$1e^{-3}$	1.4	1.1	$1.6e^{-1}$	$7.5e^{-1}$	1.1
$1e^{-4}$	56	57	31	43	20

FIGURE 3 – Temps d'exécution de l'algorithme (en secondes) pour la fonction goldstein_price

	Nombre de machines				
Précision	1	2	3	4	5
$1e^{-1}$	$9.0e^{-3}$	$5.5e^{-3}$	$2.8e^{-3}$	$3.0e^{-3}$	6.0^{-3}
$1e^{-2}$	$1.1e^{-1}$	$6.2e^{-2}$	$7.6e^{-2}$	$3.5e^{-2}$	$8.5e^{-2}$
$1e^{-3}$	1.4	1.1	1.4	$7.6e^{-1}$	1.0
$1e^{-4}$	57	57	31	41	21

FIGURE 4 – Temps d'exécution de l'algorithme (en secondes) pour la fonction three_hump_camel

On observe que les résultats peuvent changer en fonction du nombre de machines spécifié par l'utilisateur. Cette différence est explicable si la fonction *minimize()* renvoie une valeur différente en fonction de l'intervalle donné, même si les minimums se situent au même point. En effet, nous divisons le premier intervalle en parts égales selon le nombre de machines. L'algorithme utilisant une dichotomie, il est normal que si nous utilisons un nombre de machine n, nous obtenons le même résultat sur des instances où le nombre de machine est de la forme $n*2^k$. En effet, $n*2^k$ est aussi le nombre de divisions égales de l'intervalle de départ à l'itération k pour l'instance avec n machines.

Fonction	**Version séquentielle**	**Version parallèle**
booth	3.8	1.9
beale	4.4	2.9
goldstein_price	93	43
three_hump_camel	110	41

FIGURE 5 – Comparaisons des temps (en secondes) entre la version séquentielle et la version parallèle pour une précision de 0.0001 et pour 4 machines

On remarque ici que les temps d'exécution du programme en séquentiel sont deux fois plus élevés qu'avec le programme en MPI. Cela montre bien que la parallélisation de l'algorithme est intéressante avec MPI pour ce problème. Pour avoir une meilleure vision de la puissance de calcul, il faudrait lancer l'exécution sur plusieurs machines et non sur une seule.

3 Implémentation avec OpenMP

3.1 Choix d'implémentation

L'implémentation de l'algorithme avec OpenMP se trouve dans le fichier *optimization-par.cpp*. Ce fichier contient également l'implémentation avec MPI présentée dans la section 2.

Tout d'abord, nous avons parallélisé les boucles "for" permettant le remplissage des tableaux contenant les sous-intervalles de X et de Y. Ceci peut paraître négligeable pour un petit nombre de machine mais peut devenir intéressant lorsque l'on doit faire le découpage pour beaucoup de

machine. Ensuite, nous avons choisi de paralléliser la boucle qui appelle la fonction *minimize()* avec un "parallel for" tout en partageant la variable "min_ub" pour chaque thread. Nous avons testé une solution avec un "parallel for reduction" avec des sections critiques dans la boucles for (pour éviter les erreurs dues au partage de "min_ub"). Cette solution entraînait des temps plus élevés à cause des sections critiques, c'est pourquoi nous l'avons abandonnée (mise en commentaire dans le code).

Pour terminer, nous avons mis en place des "parallel section" dans la fonction *minimize()* : nous avons créé une section pour chaque appel récursif à *minimize()*, tout en limitant le nombre de thread à quatre. Il a été nécessaire d'ajouter des sections critiques dans la fonction où l'on accède en écriture aux variables partagées. On obtient, dans cette solution, de meilleures performances qu'avec un simple "parallel for" détaillé dans le paragraphe précédent.

3.2 Tableaux des performances

Les jeux de tests suivants ont été effectués pour un nombre de machines égal à 2. Les temps affichés sont en secondes.

	Préscision			
Fonction	0.1	0.01	0.001	0.0001
booth	0.056	0,14	0,89	6,57
beale	0.042	0.069	0.44	3.05
goldstein_price	0.047	0.2	1.78	48.55
three_hump_camel	0.071	0.39	7.4	26.3

FIGURE 6 – Temps d'exécution de l'algorithme (en secondes) pour les différentes fonctions

Fonction	**Version séquentielle**	**Version parallèle**
booth	3.8	6.57
beale	4.4	3.05
goldstein_price	93.4	48.55
three_hump_camel	109.43	26.3

FIGURE 7 – Comparaisons des temps (en secondes) entre la version séquentielle et la version parallèle pour une précision de 0.0001

On remarque ici que pour une précision de 0.0001, on obtient des ratios de temps différents suivant les fonctions. Cela s'explique par le fait que certaines fonctions ont un résultat local satisfaisant plus rapide à calculer que les autres. Par exemple, dans le cas de la fonction goldstein_price, le temps de calcul avec l'algorithme séquentiel est deux plus long qu'avec l'algorithme parallèle.

4 Conclusion

Notre solution actuelle permet d'avoir des temps raisonnables par rapport à l'algorithme séquentiel jusqu'à une précision de l'ordre de 0.00001. Les temps d'exécution entre la version avec OpenMP et sans OpenMP sont similaires sur les précisions faibles et plus significatifs sur les grandes précisions : on peut noter des variations suivant la fonction testée, dans certains cas la première version sera meilleure (ex : booth) et dans d'autre, la seconde sera à privilégier (ex : beale).

Il est possible d'améliorer les performances en modifiant l'heuristique de façon à adapter l'algorithme à la version parallèle. Nous n'avons pas testé les bibliothèques Intel TBB et C++11 Thread pour ce projet, mais il pourrait être intéressant de comparer les performances avec ces implémentations. Concernant les difficultés rencontrées, nous avons passé du temps à comprendre comment découper les données du problème pour profiter au mieux des performances de MPI. Pour finir, ce projet nous a permis de mettre en pratique les outils de parallélisation vus en cours.