Projet Multicore Programming

Léo Cassiau Ugo Mahey

Avril 2016 - M1 ALMA

Table des matières

1 Introduction

Ce document résume le travail effectué au cours d'un projet réalisé à la faculté des sciences et techniques de Nantes, par des étudiants de 1re année du master Architectures Logicielles (ALMA). Ce projet a pour but de mettre en œuvre les connaissances acquises durant le cours de Multicore Programming enseigné par M. Frédéric Goualard. Le sujet du projet consiste à modifier un programme existant afin d'optimiser le temps d'exécution en parallélisant les différentes opérations.

Le programme à optimiser utilise un algorithme de branch and bound [1] afin de trouver le minimum d'une fonction sur un domaine initial. Le programme donné propose de trouver le minimum de quatre fonctions : booth, beale, goldstein price et three hump camel [2]. Enfin, le programme propose de choisir la précision du résultat attendu, en sachant qu'un résultat précis demande un plus grand temps de calcul. L'objectif est donc de trouver la manière la plus efficace de paralléliser le programme, la principale difficulté étant que si un sous-domaine s'avère contenir un nouveau minimum, cela influe alors sur les autres domaines de recherche.

La première partie de ce rapport commence par expliquer comment nous avons parallélisé le programme sur une seule machine qui a un processeur avec plusieurs cœurs, tandis que la deuxième partie explique la parallélisation du programme sur plusieurs machines. Enfin, une conclusion sur l'expérience apportée par le projet termine ce rapport.

2 Parallélisation sur différents processeurs

Nous avons commencé par la parallélisation sur les processeurs multi-coeur afin de mieux cerner les difficultés liées au projet, avant de réaliser la parallélisation sur différentes machines, qui est une partie plus compliquée.

2.1 OpenMP ou Intel TBB ?

Nous avons choisi d'utiliser OpenMP au lieu d'Intel Threading Building Blocks (ITBB) car OpenMP est plus facile à implémenter et nous n'avions pas besoin des possibilités supplémentaires offertes par ITBB. En effet, nous souhaitons simplement paralléliser quatre opérations, et il n'y a qu'une variable partagée entre ces quatre opérations.

2.2 Opération(s) parallélisée(s)

Après une rapide analyse du programme, nous avons déduit que la seule opération intéressante à paralléliser était l'opération *minimize*, qui est récursive. Cette opération vérifie si le domaine donné contient le minimum recherché, et si nécessaire, il divise ce domaine pour faire une nouvelle recherche plus précise. Le domaine est alors divisé en quatre parties égales, et le programme calcule alors le minimum de ces nouveaux domaines.

1. https://fr.wikipedia.org/wiki/S%C3%A9paration_et_%C3%A9valuation
2. https://en.wikipedia.org/wiki/Test_functions_for_optimization (En anglais)

Nous avons donc choisi de paralléliser ces quatre opérations, en les distribuant équitablement entre chaque processeur à chaque itération. Pour cela, nous avons utilisé la notion de *section* d'OpenMp, une opération correspondant à une section. Ainsi, chaque opération est redistribuée à un processeur libre, jusqu'à ce qu'ils soient tous occupés. OpenMP arrête alors de distribuer les opérations, et les processeurs continuent les calculs sur chacun de leurs sous-domaines. Enfin, une variable contenant la liste des minimums est partagée chaque opération, nous avons donc ajouté des notations *critical* d'OpenMP pour éviter que deux processeurs modifient la liste en même temps, ce qui provoque des erreurs.

2.3 Résultats observés et conclusions

Nous observons que par exemple, pour la fonction booth, les performances sont plus que quadruplées sur un ordinateur avec quatre cœurs. Et ce gain en performances important est visible sur toutes les fonctions, dès que l'on souhaite une précision supérieure à la première décimale.

Ces résultats s'expliquent par le fait que le minimum n'est non pas dans le premier sous-domaine, mais dans un des trois autres. Avec la parallélisation, on évite ainsi de parcourir le premier sous-domaine inutilement, en parcourant directement le domaine contenant le minimum. Nous en concluons que distribuer les différentes opérations au plus grand nombre de machines permet un gain de performance important, et qu'il est donc intéressant de paralléliser massivement l'algorithme, sur plusieurs machines par exemple.

3 Parallélisation sur différentes machines

Nous avons utilisé Message Passing Interface (MPI) pour paralléliser les différentes opérations sur les différentes machines. Cette partie étant la plus compliquée à appréhender c'est sur celle ci que nous avons passé le plus de temps.

3.1 Choix dans la parallélisation

Nous avons choisi de paralléliser selon un modèle simple. La première machine (celle de rang 0) va effectuer un découpage des tâches à gérer selon le nombre de machines disponibles. On a un compteur qui va incrémenter le nombre de fois où l'on envoie un intervalle à minimaliser. On découpe d'abord en quatre et l'intervalle, s'il y a plus de quatre machines disponibles, on va redécouper le dernier intervalle et va continuer à envoyer aux machines encore disponibles. Ainsi chaque machine va avoir un "minimize" à faire, on sépare donc le travail à effectuer. S'il y a plus de quatre machines, certaines machines vont avoir à minimiser des intervalles plus petits et donc avoir un peu moins de travail à effectuer. Une fois ce découpage effectué chaque machine va calculer indépendamment un minimal local. Une fois cela fait on effectue un reduce pour calculer le minimum des minimums. Dans notre reduce nous ne prenons pas en compte toute la liste de tous les minimums, le résultat ne va donc afficher que le minimum de tout les minimums. De plus une fois les différentes opérations envoyées aux différentes machines il n'y a plus de communication entre les différentes machines.

Ce manque de communication fait que certaines opérations qui pourraient être évitées sont quand même effectuées.

3.2 Résultat

On peut observer que le temps mit par MPI seul n'est pas forcément plus court. Cela s'explique par un manque de communication entre les ordinateurs. A contrario un trop grand nombre de communications entre les machines entraîne également un ralentissement. Les fonctions de réception de MPI étant bloquantes le coût de communication sur un réseau peuvent devenir importantes. Ce qu'il faut c'est trouvé un juste un équilibre entre les différentes solutions.

4 Conclusion

Pour résumé, OpenMP est un outil très utile, permettant de facilement paralléliser son programme, tout en gardant l'aspect séquentiel du programme. Ainsi, le code est facile à maintenir et à comprendre, et le gain en performances est réel. Tandis que MPI s'avère être plus compliqué à mettre en place, le code demandant d'être adapté à la parallélisation, ce qui rend le code moins compréhensible. De plus, l'envoi des messages entre les différentes machines pose les mêmes problématiques que l'on peut retrouver en réseau : que ce passe-t-il s'y un ordinateur tombe en panne par exemple . Malgré cela, MPI possède un potentiel énorme en gain de temps si l'on possède de nombreuses machines et que l'on a des calculs importants à faire. En définitive, nous serons sûrement amenés à réutiliser OpenMP qui est un outil facile à utiliser et performant. Cependant, il est peu probable que l'on soit amené à réutiliser MPI car il est plus compliqué à mettre en place, et il est plus rare qu'un programme tourné sur de nombreuses machines.

Glossaire

ALMA Architectures Logicielles. 2

ITBB Intel Threading Building Blocks. 2

MPI Message Passing Interface. 3, 4