Université de Nantes
UFR des Sciences et Techniques
Master ALMA 1

Projet Multicore Programming

Guillaume Charon
Jérôme Pages

Nantes, le 9 avril 2013

Sommaire

Introduction

Dans le cadre du module de *Programmation Multicœurs*, le projet de travaux pratiques portait sur l'implémentation en parallèle d'un algorithme séquentiel de *Séparation et évaluation, ou branch and bound* en profitant des fonctionnalités offertes par le réseau et l'architecture multicœurs et multithreads des machines actuelles. En effet, après avoir étudié durant les cours magistraux et les travaux dirigés un certain nombre de moyens à notre disposition pour paralléliser les tâches, nous allons pouvoir les mettre en application dans des cas très concrets.

Dans ce rapport, nous allons présenter dans un premier temps la première version du programme qui utilise uniquement Open MPI pour partager les travaux sur des machines interconnectées localement puis nous présenterons la seconde version qui introduit le concept de parallélisation locale à une machine en utilisant OpenMP.

1 MPI

1.1 Mise en place

1.1.1 Le travail de la machine « Maître »

Dès le début de la fonction *main* et après l'initialisation de MPI, chaque instance de programme récupère son rang et la taille du communicator auquel ils appartiennent. Comme pour la version séquetielle du programme, et puisque seule la machine de rang 0 peut lire les entrées de l'utilisateur, c'est à elle de demander les différentes informations necessaires pour le traitement : le nom de la fonction à traiter et la précision souhaitée. Avant de commencer à découper le travail, la machine principale se met déjà en attente asynchrone des résultats des machines secondaires afin d'être sûre de pouvoir recevoir à temps les informations. Une des premières versions que nous avions réalisé permettait la reception de données trop tardivement, ce qui posait des problèmes d'attente infinie.

Après tout ceci commence le partage du travail par la machine-maître et la transmission aux machines-esclaves des données necessaires. Pour cela, le programme effectue dans un premier temps un premier découpage en quatre boîtes de l'intervalle de travail. Ensuite, un tableau de double est construit dans le but d'envoyer aux autres « processors »les données suivantes :

- L'index de la fonction concernée parmi les 4 fonctions suivantes disponibles : *Booth*, *Beale*, *Goldstein Price* et *Three Hump Camel* ;
- Le minimum et le maximum des deux intervalles nécessaires afin de définir l'espace de travail de la fonction. Puisque les structures sont complexes à transmettre par le réseau, il est plus simple de les découper en types de base, en l'occurence *double* ici ;
- Le seuil, qui correspond à la précision du découpage des intervalles ;
- Et le minimum local, sur lequel repose tout l'algorithme.

Une fois ce tableau effectué, il est envoyé aux trois machines esclaves. Pendant le temps de traitement, la machine principale ne reste pas inactive et s'occupe elle-même du dernier intervalle qui n'a pas encore été traité. Comme dans la version séquentielle, le traitement est récursif, c'est pourquoi l'appel à la fonctions minimize demande un paramètre à initialiser à faux lorsque le partage a déjà été effectué.

Pour stoker les différents minimums locaux trouvés, nous avons choisi d'utiliser la structure de données *Set* en C++, qui a pour caractéristique d'être ordonnée. En effet, à chaque ajout un prédicat de comparaison est utilisé pour insérer au bon endroit l'élément dans l'ensemble. Après avoir calculé le minimum local pour le rang 0, il est ajouté dans cette structure. Pour les autres, on utilise l'instruction

Wait afin d'être sûr de la terminaison de chaque tâches distante et on stocke de manière identique chaque minimum. Il suffit ensuite de récupérer le premier élément de l'ensemble pour obtenir le plus petit minimum trouvé.

1.1.2 Le travail des machines « Esclaves »

De leur côté, les machines dont le rang est supérieur à 0 attendent les données que leur envoie la machine principale puis reconstruisent les structures de données nécessaires au fonctionnement de l'algorithme : la fonction concernée est retrouvée à partir de son indice et les intervalles sont à nouveau créés à partir des valeurs minimales et maximales. Ensuite, le traitement est effectué de manière identique au programme séquentiel.

Une fois le minimum trouvé, il est transmis à la machine maître afin de comparer avec les autres machines le résultat trouvé.

1.2 Problématique : le partage du minimum

Lorsqu'on compare les résultats des programmes séquetiel et parallèles, nous nous rendons assez vite compte que les premiers sont plus précis que les seconds. Cela est dû à l'absence de partage en temps réel des nouveaux minimums locaux trouvés entre les différentes machines car certaines boîtes sont mises de côté alors pourraient améliorer les résultats. Voici un tableau représentant les différences de temps de traitement entre les deux versions de l'algorithme :

Fonction	**séquentielle**	**parallèle**
Booth	3.29316e-06	4.47407e-4
Beale	4.613e-06	3.624e-4
Goldstein Price	3.78811	3.78811
Three Hump Camel	4.47407e-4	4.47407e-4

TABLE 1.1 – Résultats obtenus avec une précision de 0.001 en fonction du type de programme utilisé.

Pour résoudre ce problème, on pourrait imaginer envoyer à chaque nouveau minimum local un message en broadcast annonçant la nouvelle à tous les processors du communicator. Cependant, cette solution naïve n'est pose plus de problème qu'elle semble en résoudre puisque cela provoquerait un engorgement sur le réseau, chaque machine trouvant assez souvent une nouvelle donnée susceptible d'intéresser les autres. De plus, l'envoi sur le réseau est très coûteux en temps, cela ralentirait énormément d'envoyer des données, ainsi que de traiter les nouveaux minimums reçus.

Toute la difficulté de l'optimisation de cet algorithme réside donc dans la capacité de communication des processors : s'ils communiquent trop, cela diminue les performances mais au contraire, s'ils ne communiquent pas il y a une perte de précision.

1.3 Généralisation à n machines

Outre l'échange des minimums locaux entre les machines, une des voies possibles dans l'amélioration de la parallélisation est la gestion d'un nombre de machine indéfini. En effet, lorsque le communicator possède 5 machines ou plus, il devient important de les mettre aussi à contribution dans l'effort de calcul général pour mettre à profit leurs capacités de traitement. Nous n'avons malheureusement pas eu le temps de mettre cela en place mais nous avons tout de même réfléchi à un moyen d'implémenter facilement la gestion des machines inoccupées.

Tout d'abord, nous avons pensé à créer un vecteur de booléen qui contiendrait l'état d'activité de chaque machine du communicator et initialisé au début à false. Au fur et à mesure du découpage en boîte par la machine de rang 0 et de l'attribution du travail à 3 machines inactives, leurs états serait mis à jour. Dans le cas où une machine finirait son traitement avant que la répartition soit terminée, elle envoie à la machine 0 le minimum local obtenu et cette dernière pourra la considérer comme une machine inactive à nouveau à qui elle peut confier du travail.

Il serait aussi possible de penser à un système plus pyramidal où chaque machine pourrait diviser le travail à effectuer et l'attribuer à 4 autres machines jusqu'à ce que la totalité des processors soient occupés. Pour cela, on peut s'inspirer du proverbe latin « Divide ut imperes », *Diviser pour régner*, en mettant en place un découpage du pool de machines disponibles à chaque étage : le rang 0 charge les rangs 1, 2, 3 et 4 de redécouper le domaine, celles-ci n'auront qu'à diviser le travail aux 3 suivantes et ainsi de suite jusqu'à épuisement des machines inactives. Pour économiser du temps sur les échange sur le réseau, il est possible de demander à chaque machine d'un rang inférieur de rediriger son résultat avec le rang 0 qui n'aurait qu'à sélectionner le plus faible.

2 Open MP

2.1 Pourquoi OpenMP ?

Si nous avons choisi, c'est pour les raisons suivantes :

- Il s'agit de la manière la plus simple de paralléliser un programme puisque cela nécessite juste d'ajouter les bonnes instructions « #pragma omp »aux endroits où le programme peut fonctionner sous formes de tâches concurrentes.
- Cela ne demande pas de réécrire l'algorithme de base pour l'adapter comme avec *Intel Threading Building Blocks*.
- La spécification d'OpenMP a été rédigée par de grands industriels du secteur donc cela en fait un des outils les plus fiables, autant sur le plan de la qualité que du suivi sur le long terme.

2.2 Utilisation d'OpenMP

Toute la difficulté de cette exercice est de choisir les parties du programmes qui gagneraient à être parallélisée. Dans notre cas, nous avons repéré plusieurs fonctions qui mériteraient à faire l'objet de traitements séparés dans plusieurs threads. Dans la suite, nous allons présenter les modifications que nous avons opérés en fonction de l'attribution des rangs.

2.2.1 Pour le rang 0

La séparations des travaux en threads va se faire à deux endroits pour la machine principale.

Premièrement, au niveau de la distribution du travail. En effet, après le découpage des quatre intervalles, on utilise une instruction « #pragma omp parallel for »afin de transmettre le travail plus efficacement. Ainsi, trois threads s'occupent de l'envoi aux machines distantes grâce à des fonctions MPI_Send, et non la fonction MPI_Scatter qui aurait empêché la parallelisation. Le quatrième commence à travailler localement sur le dernier intervalle.

Pour l'intervalle qui est traité localement, un deuxième découpage est effectué et distribué à nouveau à quatre threads qui vont effectuer un quart des traitements. Cela permet en théorie de diviser par quatre le temps de calcul, même si en réalité avec le temps de création et traitement des threads ce taux doit être revu à la baisse.

2.2.2 Pour les autres rangs

Les machines de rang 1 ou supérieur utilisent le même procédé que pour le traitement en local du rang 0 : le découpage est effectué pour donner 4 intervalles qui vont être spérarés dans 4 threads différents afin de paralléliseer les calculs.

Conclusion

Dans un monde où l'amélioration des performances des programmes passera par la parallélisation des tâches pour profiter des architectures des machines, ce projet correspond tout à fait à une problématique d'actualité puisque peu de programmes sont réellement adaptés et conçus pour cela. Nous avons donc pu mettre en pratique les connaissances acquises durant les cours magistraux et les travaux pratiques à propos de la répartition de tâches entre des machines sur un réseau local ainsi que la division via OpenMP en threads des calculs parallélisables.

Parmi les évolutions envisageables possibles, il aurait été possible de transmettre avec les fonctionnalités de MPI, et sous certaines conditions, les minimums locaux de chaque machine pour obtenir un résultat plus précis. Au niveau d'une machine, il est sans doute possible d'optimiser plus efficacement le code pour augmenter la part parallélisable du programme et ainsi réduire au maximum la partie séquentielle qui ralentit le traitement.