Rapport de projet multicore programming

M1 ALMA 206-2017

Étudiants :
JAIN Edwin
LEMETAYER Pierre

Professeur encadrant :
GOUALARD Frédéric

Introduction

Lors de ce projet, nous avons pour but de paralléliser l'exécution d'un programme séquentiel à l'aide de l'interface de programmation OpenMP (Open Multi-Processing) et à la librairie MPI (Message Passing Interface). Comme sujet de ce projet, nous avons à notre disposition une courbe sur 3 dimensions qui contient un maximum et un minimum. Notre objectif est donc de trouver le minimum de cette courbe le plus rapidement possible en utilisant les technologies de parallélisation citées précédemment.

Fonctionnement de l'algorithme séquentiel

Pour ce projet, il nous a été fourni le code source d'un programme séquentiel qui permet de trouver le minimum de la courbe.

Le programme va donc commencer par demander une précision à l'utilisateur. Cette précision est l'intervalle minimal qui ne doit pas être franchie lors du découpage de la fonction. La courbe, qui est réalité un cube va être découpé en 4 parties. Cette opération va être répétée autant de fois jusqu'au moment où l'intervalle des boîtes est sur le point de devenir inférieurs à la précision précédemment renseignée. Avec cette multitude de boîte, le programme va déterminer où se trouve le minimum de la fonction.

C'est donc au moment du découpage de la fonction en différentes parties qu'il est judicieux de paralléliser ce processus pour optimiser le temps d'exécution.

Parallélisation du programme

Pour comparer nos résultats, nous avons commencé par lancer une série d'exécution du programme séquentiel pour obtenir une plage de valeur qui montre le temps d'exécution en tick d'horloge. Nous avons opté pour cette unité de mesure temporelle et non les secondes car l'exécution du programme se faisait trop rapidement pour que nous puissions obtenir des valeurs comparables en seconde. Après plusieurs exécutions, nous obtenons une plage de temps d'exécution variant entre 26000 et 35000 ticks d'horloge.

MPI

Dans un premier temps, nous allons déterminer quelles actions peuvent être partagées avec différentes machines et lesquelles ne le sont pas.

Nous observons donc que la saisie utilisateur ne peut pas être partagée, il est donc nécessaire de l'effectuer par une seule machine puis d'envoyer les valeurs saisies (fonction et précision). Pour cela, nous limitons l'exécution du bloc de code à la seule machine mère, à savoir celle qui est de rang 0 puis nous forçons l'attente de ces valeurs aux autres machines par un broadcast.

La partie que nous pouvons partager est donc le premier appel récursif de la méthode « minimize ». Pour cela, nous choisissons de découper en sous-intervalle l'intervalle initial de x (nous aurions pu choisir y), en autant d'intervalle que nous avons de machines. Nous décidons de laisser cette action à la machine de rang 0, bien qu'il aurait été plus efficace de la laisser à chaque sous machine qui aurait choisi son propre sous intervalle grâce à son rang. Néanmoins, nous expliquerons ce choix plus tard.

Une fois ce découpage effectué, les valeurs sont réparties via un scatter. Chaque machine doit alors découper l'intervalle de y en autant de sous intervalle que X a été découpé. La machine est alors capable de lancer un appel à la méthode minimize avec sa valeur d'intervalle de X et autant de fois qu'il y a d'intervalle de Y dans le but de rester dans un ensemble cubique (et non rectangulaire si y n'est pas découpé).

Une fois que chaque machine a terminée, elle renvoie la valeur, de la borne minimale, qu'elle a trouvée à la machine de rang 0. La machine mère va ensuite effectuer un Reduce de toutes les valeurs qu'elle a obtenues pour déterminer le minimum de la courbe.

Dans notre code, nous avons pu observer un comportement qui n'était initialement pas souhaité. En effet, lors de notre découpage de l'intervalle X, nous effectuons un découpage de la valeur arrondie à l'inférieur de la précédente valeur puissance de 2. Ainsi, si nous avons 17 machines, nous effectuons seulement un découpage en 16. Ce phénomène est identique lorsque nous avons 31 machines.

Néanmoins, ce comportement pourrait être utilisé pour effectuer un calcul plus sécurisé. C'est à dire envoyer plusieurs fois une valeur à une machine qui n'aurait aucun sous intervalle de X, permettant ainsi de nous prévenir d'une machine qui pourrait planter et nous faire perdre une partie du calcul. Bien sûr, l'état actuel de notre code ne le permet pas, c'est néanmoins une perspective d'évolution qu'il est possible d'envisager.

Après la réalisation de la parallélisation via MPI, nous avons effectué une batterie de test pour mesurer les performances de notre programme avec un nombre de machines croissant par puissance de 2.

Nous avons commencé par lancer notre programme avec une seule machine pour comparer le résultat avec l'exécution séquentielle. Au bout de quelques tests, nous obtenons une plage de temps d'exécution variant entre 33000 et 44000 ticks d'horloge. Les résultats pour le test à une machine sont donc moins bons que pour l'exécution séquentielle.

Nous avons ensuite enchaîné avec des tests avec 2, 4 et 8 machines. Pour ces tests, les temps respectifs sont 20375, 19000 et 17000 ticks d'horloge. Nous observons donc une baisse constante des temps d'exécution mais qui ne sont pas proportionnels au nombre de machines.

Comme dernier test, nous avons lancé notre programme avec 16 machines. Cependant, nous obtenons des temps d'exécutions qui sont moins bons que pour 8 machines puisque la plage varie entre 18000 et 30000 ticks d'horloge. Nous pensons que cette baisse de performance est dû au nombre de threads disponibles à un instant T. En effet, avec 8 machines, il est possible de les simuler sur les threads de la machine hôte en simultané alors que pour 16 machines, le programme est obligé d'attendre que des threads terminent la simulation d'une machine pour lancer les suivantes.

OpenMP

Dans un second temps, nous avons appliqué OpenMP pour paralléliser le programme que nous avons préalablement parallélisé avec MPI. Dans un premier temps, nous avons tenté d'appliquer des « parallel for » sur les quelques boucles for qui étaient présentent dans le code, mais nous avons observé que ce n'était pas probant puisque nous passions de 10 à plus de 700 ticks d'horloge avec la présence d'un « parallel for ». Nous avons alors décidé de nous focaliser sur les parties communes à chaque machine MPI. C'est pourquoi nous avons voulu paralléliser l'appel à la fonction minimize. Avec l'utilisation de la méthode « split_box » qui fournit 2 sous-intervalles pour X et Y, nous obtenons 4 nouvelles combinaisons pour l'appel à minimize.

Ainsi, en fixant le nombre de threads nécessaires à 4, nous pouvons faire appel aux 4 appels de minimize chacun dans un Thread.

Pour cela, nous avons défini une section à paralléliser (omp parallel sections), où nous cherchons la réduction à la valeur minimale pour la borne minimale et où chaque appel à minimize est une sous-section (4 au total).

Dans le code actuel, OpenMp ne fonctionne pas. Nous pouvons néanmoins supposer, dans un environnement où le processeur est capable de gérer 4 threads, que le temps d'exécution global aurait diminué.

Conclusion

Lors de ce projet, nous avons pu mettre en application ce que nous avions vu lors du module de multicore programming. Ce type de mise en œuvre est un bon moyen de montrer les effets de la parallélisation sur des programmes qui peuvent demander beaucoup de calcul. En effet, nous avons pu noter une amélioration, plus ou moins efficace, de nos temps d'exécutions. Cependant, bien que les temps soient meilleurs, ils sont légèrement faussés puisque nous travaillons sur un programme non prioritaire pour la machine. C'est ainsi que nous pouvons obtenir des plages de temps pouvant varier de plus de 10000 ticks d'horloge alors qu'en temps normal, cette variation ne devrait pas être aussi importante.

Nous notons aussi qu'il n'est pas pertinent de tenter de paralléliser tous les blocs de code, comme nous avons pu l'observer avec OpenMp.