Rapport de Projet

Branch and bound Parallel

MENORET Clément, RULLIER Noémie
15 avril 2013

UNIVERSITÉ DE NANTES

Table des matières

1 Introduction

L'objectif de ce projet fut de paralléliser l'algorithme *branch and bound* permettant de calculer le minimum d'une fonction réelle par le découpage en boites de domaine.

Nous allons donc ici présenter les différentes décisions prises afin de paralléliser l'algorithme à l'aide des technologies *MPI* et *OpenMP*, ainsi qu'une comparaison des résultats obtenus entre la version parallèle et séquentielle.

2 Les étapes de parallélisation

Afin de paralléliser ce programme, nous avons utilisé MPI et OpenMP. Ils permettent respectivement d'exploiter des ordinateurs distants, afin d'exécuter des portions de programme et de traiter les données, sur une architecture à mémoire partagée.

2.1 MPI

Nous avons ici décidé d'utiliser quatre ordinateurs grâce à MPI. Nous avons donc, dans un premier temps, regroupé toutes les machines dans le même *Communicator* et initialisé le tout. A ce stade, les quatre ordinateurs vont exécuter la suite de la fonction main. Nous avons donc précisé que seul l'ordinateur de rang 0 exécute le premier appel à *minimize_first*.

Cette première fonction permet de faire un premier découpage en quatre intervalles. Chaque machine recevra ensuite l'ensemble des informations permettant d'exécuter à leur tour la fonction *minimize* ; chaque machine l'exécutera pour un des quatre intervalles. Ces informations seront envoyées via la fonction *MPI_Send()*. Elle permet d'envoyer des données à un ordinateur en lui précisant son rang. Afin de lui envoyer toutes les données nécessaires, nous avons créé de nouvelles structures :

- **interv** : cette structure permet de stocker l'intervalle qui va être envoyé. Elle est composée des attributs x et y de type *interval* ;
- **consts** : cette structure permet de stocker l'ensemble des paramètres nécessaires à la fonction *minimize* ;
- **package** : cette structure permet d'envoyer à la fois l'intervalle et les autres paramètres. Elle est composée d'un attribut *inter* de type *interv* et d'un attribut *constantes* de type *consts*.

Nous avons décidé de créer deux structures distinctes car pour chaque machine, l'ensemble des paramètres, autre que l'intervalle, sont identiques. Afin que notre structure soit bien envoyée via la fonction *MPI_Send()*, nous avons défini le type de données à envoyer comme étant des *MPI_BYTE*.

Nous avons donc, dans la suite de la fonction main, fait appel à la fonction *MPI_Recv()*. Celle-ci permet aux différentes machines de recevoir les messages envoyés par la machine de rang i (ici $i = 0$). Ces machines vont ensuite exécuter la fonction *minimize*.

De plus, dans le but de récupérer le minimum final, nous avons ajouté la fonction *MPI_Reduce()*. Celle-ci nous permettra de trouver le minimum de tous les minimums calculés par chaque machine et de la stocker dans *min_global*. Nous devons aussi récupérer la liste des intervalles dans lesquels un minimum a été trouvé. Pour cela, nous avons pour toutes les machines (autres que celle de rang 0) envoyé à la machine 0, leur tableau contenant les minimums. La machine de rang 0, va recevoir ce tableau et ajouter les éléments de celui-ci à sa propre liste de minimums. Elle doit cependant créer un tableau de la taille du tableau qu'elle va recevoir ; or elle ne la connait pas. Pour cela, nous utilisons *MPI_Probe()* qui permet d'initialiser un *MPI_Status* et de récupérer sa taille via un *MPI_Get_Count()*.

2.2 OpenMP

Nous avons ajouté un niveau supplémentaire de parallélisation avec l'ajout d'OpenMP. En effet, sur chaque machine traitant une instance de *minimize* nous avons parallélisé les appels récursifs à la fonction *minimize*. Nous avons donc défini quatre sections (*#pragma omp section*), une pour chaque appel à *minimize*. Nous avons choisi des sections plutôt que des tâches car elles permettent d'avoir une barrière à la fin de leur exécution. Afin de ne pas avoir de problèmes de surcharge au niveau du nombre de threads, nous avons ajouté un nombre maximum de threads et défini le nombre de threads lancé à chaque instruction de parallélisation trouvée.

Nous avons ensuite essayé différentes façons d'utiliser *OpenMP*.

La première a été d'utiliser quatre variables permettant de stocker chaque minimum appelé dans les quatre appels de *minimize*. Après que toutes les sections aient terminées, on calcule alors le minimum de tous les minimums retournés.

La deuxième fut d'utiliser des *#pragma critical* permettant d'avoir des sections critiques afin que les variables soit en exclusions mutuelles.

3 Les résultats

Afin d'effectuer nos tests, nous avons choisi d'utiliser tous les algorithmes proposés avec des précisions différentes.

3.1 MPI

Suite à notre étude comparative des deux versions des programmes, séquentiels et parallélisés, on remarque que les résultats obtenus avec la version parallélisée sont identiques aux résultats obtenus avec la version séquentielle. Cependant, le fait d'utiliser MPI de manière virtuelle sur une seule machine physique provoque un ralentissement de l'exécution de l'algorithme du aux nombreux traitements à réaliser pour la simulation du comportement de MPI et les modifications apportées à l'algorithme en terme d'envoi et de réception des données. Entre différentes machines physiques, la charge de travail serait répartie.

Les résultats détaillés de notre benchmark sont donnés dans les tableaux suivants :

Cette comparaison a été faite pour la précision *0,01*.

Sequential version			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	16889	9.153558944677261	5.080
beale	2209	0.001346965117212209	2.976
three_hump_camel	154	0.1145362854871269	4.316
booth	225	0.0008430480957031254	5.348

Parallel version with MPI			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	14989	9.153558944677261	0.512
beale	83	0.001346965117212209	0.044
three_hump_camel	152	0.1145362854871269	0.116
booth	23	0.0008430480957031254	0.040

Cette comparaison a été faite pour la précision *0,005*.

Sequential version			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	29237	6.1188468974418	1.100
beale	76	0.0002946668823264963	0.076
three_hump_camel	152	0.02863407135145293	0.276
booth	21	0.0001564025878906251	0.080

Parallel version with MPI			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	36163	6.1188468974418	11.001
beale	3795	0.0002946668823264963	4.388
three_hump_camel	154	0.02863407135145293	4.672
booth	303	0.0001564025878906251	3.304

Cette comparaison a été faite pour la précision *0,001*.

Sequential version			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	18506	3.788114240740458	2.816
beale	76	4.613139957035884e-06	0.576
three_hump_camel	152	0.0004474073648452814	5.124
booth	23	3.293156623840334e-06	0.568

Parallel version with MPI			
Algorithm	Number of minimizers	Upper bound for minimum	Time (s)
goldstein_price	96767	3.788114240740458	11.321
beale	18525	4.613139957035884e-06	6.112
three_hump_camel	154	0.0004474073648452814	14.157
booth	808	3.293156623840334e-06	4.256

3.2 OpenMP

Nous n'avons malheureusement pas eu le temps de terminer une implémentation de l'algorithme parallèle qui fonctionne bien avec OpenMP, nous ne pouvons donc pas fournir de résultats expérimentaux.

4 Conclusion générale

Ce projet nous a permis de réfléchir à la façon de paralléliser un programme séquentiel tout en s'initiant à la programmation avec MPI et OpenMP. Bien que nous ayons regretté de ne pas pouvoir tester ce programme sur plusieurs machines physiques, l'utilisation du parallélisme sur ce projet reste cruciale à partir du moment où l'on souhaite obtenir un résultat calculé avec une bonne précision en un minimum de temps sur cet algorithme.