Calcul Haute Performence

encadré par

Charles Bouillaguet

Rapport de projet

réalisé par

Laurent Dang Vu

Thomas Genin

Mathématiques Appliquées et Informatique Numérique
2e année de cycle ingénieur
Polytech Sorbonne
2019-2020
Paris, France

Table des matières

0.1 Introduction

0.2 Présentation du projet

Ce projet trouve sa genèse dans la résolution d'un système Ax = b, où A est une matrice de taille n, qui est plus ou moins creuse suivant ses valeurs. B quant à lui est un vecteur de traille n et x la solution de ce système linéaire de n inconnues. Pour résoudre ce système nous utiliserons alors la méthode du gradient conjugué avec "prédictionneur". De ce fait, il faudra donc implémenter et calculer un produit matrice vecteur, ou un produit vecteur vecteur. Ce qui peut devenir très chronophage lorsque la taille n du système devient importante.

Ainsi l'objectif de ce projet est de réaliser une implémentation qui nous permettrait de résoudre ce système de façon plus rapide que la manière séquentielle. Et donc de paralléliser notre

programme. Pour ce faire, nous disposons des ressources de la faculté des sciences de la Sorbonne, des machines de ses bibliothèques ainsi que de méthodes de OpenMPI (ou MPI) et de OpenMP.

De cette manière nous réaliserons d'abord un procédé de mise en oeuvre avec MPI. Puis nous rajouterons OpenMP.

0.3 Algorithmes et choix d'implémentation

Dans cette section nous détaillerons les algorithmes utilisés pour réaliser ce projet et nous justifierons notre choix d'implémentation. De plus nous expliciterons les nouvelles fonctions qui apparaissent en plus du code séquentiel et leur utilité.

0.3.1 Programmation MPI

Dans un premier temps nous avons décidé de nous consacrer à la programmation MPI uniquement. Cette section sera scindée en deux parties. La première sera l'implémentation de MPI sur différentes machines physiques. Ensuite nous exposerons l'implémentation en local de MPI sur une seule et unique machine à quatre coeurs.

0.3.2 Les nouvelles fonctions

`tcreate_checkpoint`, cette fonction crée un checkpoint toutes les minutes et sauvegarde les valeurs de x, z, r, p, q, rz. De façon à ce que si le serveur plante, qu'on ne perde pas tout et qu'on puisse relancer le programme avec les valeurs sauvegardées en mémoire. Cette fonction prend en argument n la taille des vecteurs, les vecteurs x, z, p, r, q et le scalaire rz. Enfin elle crée un fichier txt dans lequel elle écrit ces vecteurs.

`init_from_checkpoint`, cette fonction va prendre les mêmes arguments que la fonction précédente, et va les initialiser en lisant les valeurs des vecteurs écrites dans le fichier texte checkpoint.

(Nota Benne) Pour savoir s'il existe un fichier txt avec les valeurs sauvegardées, l'utilisateur va indiquer dans la commande exécutable `checkpoint` pour indiquer que le programme a planté mais qu'il y a eu une sauvegarde des données qu'il pourra charger.

L'objectif des fonctions suivantes et de scindé le calcul des fonctions qui existaient déjà pour que plusieurs processus les exécutes en même temps, et ce sans qu'un d'entre eux est à en donner l'ordre.

`textract_diag_part`, prend en entrée la matrice, un vecteur d, et deux entier n_part et i_ini. Elle effectue la même chose que `extrac_diag` mais seulement sur une portion de vecteur de taille n_part, à partir de l'indice i_ini. Cette fonction à pour but de scindée la tâche de l'extraction

de la diagonale de A entre plusieurs processus en parallèle.

sp_gemv_part, celle-ci suit le même principe que la fonction précédente mais sur le modèle de la fonction `sp_gemv`. Elle sert de même à répartir le produit matrice vecteur entre plusieurs processus (machines).

dot_part, encore une fois sur le même principe que les fonctions précédentes. celle-ci calcule un partie du produit scalaire.

0.3.3 Le cas sur différentes machines physiques

Dans un premier temps nous avons commencé par paralléliser notre algorithme sur plusieurs machines physiques.

Maître esclave

Tout d'abord nous avons pensé à structurer notre code avec la méthode du maître esclave. Ainsi le processus zéro devient le maître et il charge alors la matrice au format csv pour la broadcaster à tous les esclaves (les autres processus). Ainsi tout le monde a la matrice en mémoire. Alors le maître découpe la liste des tâches à accomplir entre les différents esclaves.

Le principe est le suivant : pour le calcul d'un produit matrice vecteur, le maître découpe le vecteur à calculer en sous-vecteurs et en envoie un à chaque processus via `MPI_send()`. Les processus recoivent le sous-vecteur via `MPI_recv()` et exécutent chacun le produit matrice vecteur uniquement sur le sous-vecteur avec la fonction `sp_gemv_part` et renvoient le résultat (une partie du produit) au maître avec un `MPI_send()`.

Concernant la façon dont les sous-vecteurs sont construits, on découpe le vecteur de manière linéaire suivant le nombre d'esclave. Ainsi on ne tient pas compte des éventuels problèmes dus à la répartition de la matrice. Donc si la matrice est creuse à certain endroit et dense dans d'autre comme c'est le cas pour la matrice `ecology2`, on aura des processus qui effectuent beaucoup de calculs et d'autres moins. Donc certains processus peuvent attendre que les autres aient fini car le maître attend le retour de tous les esclaves pour passer à la suite.

Cependant cette implémentation de sera pas retenue par la suite. Nous abandonnerons donc la centralisation sur le maître, et nous le remplacerons par une hiérarchie plane des processus. Ceci étant dû à des problèmes d'implémentation du programme.

Version statique

Par la suite nous avons donc opté pour un programme sans maître esclave avec un répartition statique des charges de travail, sans équilibrage de charge (c'est-à-dire une répartition équilibrée

de la charge de travail par processus ou machine). Nous utiliserons alors les fonctions de MPI.

En résumé, nous avons transféré le contenu de la fonction cg_solve dans le main pour avoir un programme commun à tous les processus facile à paralléliser.

Nous avons d'abord parallélisé les opérations les plus coûteuses : le produit scalaire et le produit matriciel. Nous avons fait cela en réadaptant les fonctions déjà existantes. Ainsi, chaque processus calcule seulement n_part des n (n_part étant une partie de n) produits du produit scalaire et n_part composantes.

Après le calcul, le processus 0 rassemble les résultats du calcul. Dans le cas du produit scalaire, il faut un Allreduce pour faire la somme des résultats obtenus chez les autres processus puis pour leur envoyer cette somme. Dans le cas du produit matriciel, il fait un Allgatherv pour reconstruire le vecteur en entier en recevant les composantes calculées propres à chaque autre processus. Puis il le renvoie à tout le monde.

On a pris Allgatherv au lieu de se contenter de Allgather car il y a un processeur qui effectue en plus un nombre de calculs égal au reste de la division de n par le nombre de processus (et non pas égal au quotient comme dans le cas des autres processus). Autrement dit, ce processeur a moins de calculs à réaliser que les autres : le Allgatherv permet de prendre en compte ce cas particulier.

Pour finir, à la fin, seul le processus 0 vérifie que la solution x trouvée est correcte (en la comparant au bon résultat déjà connu) et l'écrit dans un fichier.

Nous aurions aussi pu faire en sorte que chaque processus ne possède qu'une partie du vecteur total. Ainsi les opérations d'affectation (=) seraient moins lourdes.

0.3.4 Le cas de la programmation en local

Cependant, bien que nous ayons parallélisé le programme, nous avons obtenu des résultats avec un temps d'exécution plus important qu'avec le programme séquentiel. Et ce peu importe la taille de la matrice. Ce phénomène est du au coût des communications de la PPTI[I].

Nous avons donc opté pour une implémentation MPI non plus entre machines physiques mais entre les coeurs d'une machine. Nous avons donc exécuté notre programme sur les quatre coeurs d'une machine de la PPTI. Ce qui nous a permis de réduire considérablement le coût des communication de MPI.

I. Plateforme Pédagogique et Technique Informatique

0.3.5 Programme OpenMP

Par la suite nous avons aussi utilisé la bibliothèque OpenMP pour paralléliser nos programme, et notamment la vectorisation sous OpenMP 4.0. En effet, ce choix s'est imposé de lui-même car la plupart des boucles que nous avons pu paralléliser faisaient intervenir des vecteurs. De plus, la plupart des opérations vectorielles imbriquées dans les boucles se font sur des indices de vecteurs contigus en mémoire (très souvent de 0 à n-1). Ce qui rend l'utilisation de SIMD [II] optimale. Nous avons aussi beaucoup utilisé la clause `reduction` pour gérer les sommes sur les scalaires et les vecteurs de scalaires.

Cependant dans certaines boucles, la succession des vecteurs indicés n'est pas contigues en mémoire. Notamment sur les boucles indicées à u où u va de Ap[i] à Ap[i+1]. Il aurait donc fallu rendre cette distribution continue en mémoire pour optimiser l'accès en mémoire et au cache, pour SIMD, avec SSE et AVX2. Néanmoins nous avons parallélisé ces boucles avec la méthode `parallel for` de OpenMP, par manque de temps pour utiliser SSE ou AVX2.

0.4 Résultats

0.4.1 Tests sur une seule machine

Nous avons testé notre programme sur une machine à 4 coeurs. Voici les résultats :

Matrice	Temps (MPI + Openmp) en s	Temps (MPI) en s	Temps (en séquentiel) en s
Cfd1	4.3	5.4	8
Cfd2	39.6	40.3	54
parabolic_fem	50.8	60.2	29
tmt_sym	113.9	118.2	125
hood	70.9	71.4	145
ecology2	197.1	194.6	158
G3_circuit	219.1	220.9	172
Thermal2	235.6	231.2	257
Nd24k	183.6	183.9	620
Serena	117	124	171
Flan_1565	1402		2530

Figure 1 – Performances avec MPI 4 coeurs seulement

Nous avons des résultats satisfaisants pour la majorité des matrices.

II. Single Instruction on Multiple Data

0.4.2 Tests sur un cluster de 17 machines

Nous avons aussi testé notre programme sur le cluster de 17 machines de la salle 408. Voici les résultats seulement pour certaines matrices comme le temps de calcul est trop long pour certaines :

Matrice	Temps (MPI + Openmp) en s	Temps (MPI) en s	Temps (en séquentiel) en s
Cfd1	34.4	34	8
Cfd2	257.7	259.3	54

FIGURE 2 – Quelques performances avec MPI en clustering

En effet on observe que les coûts des communications pour le cluster de 17 machines entraînent parfois des performances de calcul désastreuses.

0.4.3 Comparaison cluster et local

Nous obtenons de bien meilleures performances avec MPI sur une seule machine qu'en clustering. Leurs résultats de calcul très lents s'expliquent par les communications entre les machines de la PPTI qui sont mauvaises. Le fait que notre programme fonctionne plus rapidement sur une seule machine que sur ce cluster particulier confirme notre hypothèse.

0.5 Tests avec MPI et OpenMP

Dans cette section nous aborderons les résultats obtenus avec MPI (en local) et OpenMP.

Matrice	Temps (MPI + Openmp) en s	Temps (MPI) en s	Temps (en séquentiel) en s
Cfd1	4.3	5.4	8
Cfd2	39.6	40.3	54
parabolic_fem	50.8	60.2	29
tmt_sym	113.9	118.2	125
hood	70.9	71.4	145
ecology2	197.1	194.6	158
G3_circuit	219.1	220.9	172
Thermal2	235.6	231.2	257
Nd24k	183.6	183.9	620
Serena	117	124	171
Flan_1565	1402		2530

FIGURE 3 – Performances de MPI et OpenMP

On observe que plus la matrice est grande, plus on obtient de bonnes performances. De plus, on observe que plus la densité des matrices est répartie inéquitablement et plus notre programme

peine à obtenir de bonnes performance (i.e. des performances en dessous de la version séquentielle). Ceci est du au fait que nous avons une répartition de charge statique pour MPI. Ainsi, pour améliorer ces performances, il faudrait répartir la matrice de façon dynamique ou autre. Pour ce faire, nous pourrions reprendre notre version du maître esclave, ou alors créer une fonction qui prendrait la matrice A, et suivant les blocs creux ou denses, créer des sous-vecteurs de tailles différentes pour les affecter aux processus dans le main.

De plus, pour comparer la version MPI 4 coeurs avec OpenMP à la version avec seulement 4 coeurs et MPI, on remarque qu'on arrive à obtenir de légères meilleures performances sur de petites matrices, avec la version combinée, qu'avec seulement MPI. En revanche, lorsque la taille devient importante, on paye un temps important pour OpenMP et les performances passent en dessous de MPI. Cela est dû au fait que notre implémentation de OpenMP n'est pas optimisée, car on utilise de la vectorisation là où les accès mémoire ne se font pas à la suite.

0.6 Conclusion

0.7 Ce qui ressort de ce projet

Au vu de ce projet, nous avons pu mettre en application ce qui est vraiment la base du calcul haute performance. C'est-à-dire, non pas paralléliser le plus possible, mais tenir compte des propriétés des machines sur lesquelles on travaille. Comme le coup des communication sur le réseau ou encore la structure des éléments que l'on parallélise. Il faut aussi faire des tests de façon empirique pour déterminer quelle est la manière optimale de réaliser le calcul.

0.8 Pistes d'amélioration

Tout d'abord nous pourrions mettre en oeuvre, grâce à nos cours sur la résolution de systèmes linéaires, d'autres méthodes que celle du gradient conjuguée et voir si celles-ci nous permettent d'obtenir de meilleures performances de calcul. Et aussi voir si ces autres méthodes nous permettent de manipuler des données avec une structure plus parallélisable ou plus vectorisable, de manière à optimiser la vectorisation SIMD.

Nous pourrions aussi implanter SSE et AVX2 de manière à permettre un accès aux données contigues en mémoire pour certaines portions du code où ce n'est pas le cas. Nous pourrions aussi gérer le cache de manière à accéder plus rapidement au données en fonction des calculs passés et des suivants. Et enfin bien sûr, pour certains types de matrices, implanter une répartition non statique des charges. Pour conclure, notre fonction checkpoint nous fait perdre un temps précieux dans l'exécution du code. L'optimiser nous ferait donc aussi gagner en temps.