TP de Programmation Parallèle

Florian Gasc, Silouane Gerin

Introduction

Dans le TP que nous avons effectué, nous avons implémenté les algorithmes de Fox et de Cannon dont le but est de paralléliser la multiplication de matrices. La méthode linéaire pour multiplier des matrices a une complexité en $O(n^3)$. Le but de ce TP est donc de distribuer les calculs sur l'ensemble des processus afin de répartir la charge de travail.

Nous allons travailler avec trois matrices carrées, A, B, C, dont l'ordre est égal aux nombres de processus lancés. On admet que l'utilisateur passe en entrée un nombre carré de processus ainsi que les données correctement formatées afin de remplir les matrices A et B. Le formatage des données est simple : autant de nombres que de cases, séparés par des espaces.

Algorithme de Fox

L'algorithme de Fox fonctionne de la manière suivante : à partir de nos matrices A et B, nous allons construire la matrice C telle que : $C_{i,j} = A_{i,0} \times B_{0,j} + A_{i,1} \times B_{1,j} + \ldots + A_{i,n-1} \times B_{n-1,j}$ avec $0 < i,j < n$ et n l'ordre de la matrice.

Pour ce faire nous allons « séparer » notre matrice en autant de blocs qu'il existe de processus. Chaque processus est donc responsable des calculs inhérents à son bloc. Ainsi il va falloir, transmettre à chaque processus, les données nécessaires pour réaliser ses calculs.

Voici l'algorithme de Fox : Chaque processus effectue k fois les étapes suivantes, k étant l'ordre de la sous-matrice représentant les blocs.

- On détecte la diagonale de notre matrice formée par les blocs.
- On diffuse les blocs formant cette diagonale sur la ligne à laquelle ils appartiennent.

- On calcule le produit du bloc recu depuis la diffusion avec le bloc B du processus. On stocke le resultat dans le bloc C.
- On décale verticalement les blocs B sur l'ensemble de notre matrice de processus.
- On trouve la prochaine diagonale.

Algorithme de Cannon

De même que pour l'algorithme de Fox, on veut récuperer nos données correctement afin que le processus réalise les calculs locaux.

Voici le déroulé de l'algorithme :

- Preskewing de A et B
- Chaque processus effectue k fois les étapes suivantes, k étant l'ordre de la sous-matrice représentant les blocs.
- On calcule le produit du bloc A depuis la diffusion avec le bloc B du processus. On stocke le resultat dans le bloc C.
- On décale verticalement les blocs B sur l'ensemble de notre matrice de processus.
- On décale horizontalement les blocs A sur l'ensemble de notre matrice de processus.
- Fin de la boucle for
- Postskewing de A et B

Optimisations

En travaillant avec la librairie MPI, nous avons eu accès à des fonctionnalités qui ont facilité la manipulation des données. Dans les améliorations et optimisations que nous avons implémentés :

- La lecture des matrices et leur allocation n'est fait qu'une seule fois. Si l'on imagine travailler avec des matrices de très grandes tailles, il est impératif de limiter les accès disque et l'utilisation de la mémoire. En limitant ces opérations notre programme gagne en efficacité.

- L'utilisation des fonctions MPI_Type_create_subarray et MPI_Type_create_resized nous permet de représenter dans notre programme la disposition de données de notre bloc. Cela permet des manipulations bien plus simples et bien plus claires des blocs à extraire de nos matrices globales.

- De la même manière nous avons utilisé les fonctions MPI_Scatterv et MPI_Gatherv afin de distribuer et rassembler les blocs en amont et en aval du calcul. Encore une fois, le code s'en trouve simplifié et optimisé.

Nous n'avons pas eu le temps en revanche, d'implémenter toutes les optimisations que nous avions à l'esprit.

- En particulier, il n'est pas possible de charger des matrices d'ordre différents que le nombre de processus. Ce qui implique que l'on ne puisse pas charger une matrice de 10 000 par 10 000 : cela voudrait dire que l'on a lancé 10 000 processus sur la machine. Evidemment ce n'est pas possible. Cette optimisation est relativement simple à mettre en œuvre.

- Nous voulions charger directement à la lecture du fichier, les différents blocs de chaque processus, dans un souci d'optimisation mais cela n'a pas été réalisé à temps.

- Nous voulions utiliser une librairie de type BLAS afin de réduire la complexité des produits de matrice.

- Impossible d'utilier gprof pour définir avec précision le temps passé dans chaque fonction.

Complexité

Soit n la taille de la matrice, q*q taille de la grille contenant tout les processus, b le temps pour communiquer un élément de la matrice, w le coût d'une operation arithmetique, L le coût de demarrage d'une communication. On représente m, l'ordre des blocs, avec dans notre cas m = q.

On calcule, pour l'algorithme de **Fox**, la complexité mise en œuvre :

- Une boucle for qui varie en fonction de la taille du bloc (**m**). Pour chaque boucle on a :
- Une diffusion sur la matrice
- On multiplie la matrice : trois boucles for imbriquée qui dependent de la taille du bloc
- Un envoi et une reception de message
- Une recopie de tableau : une boucle for qui dépend de la taille de la matrice.

Complexité en fonction de m : (Bcast * m) + (multiply_matrix) + Sendrecv + copy (m*(L+mb))+(m^3w) * m+(L+m^2b) * m+m * w m * ((L+mb) +(m^3)+ (L+m^2b)+w)

Pour l'algorithme de **Cannon**

- Pre et post-skewing
- Une boucle for dépendante de la taille de bloc. Par itération on a :
- Une multiplication de matrice
- Deux Sendrecv_replace.

Soit : 2 * preskew + (tcompute+(L+m^2b)$2) * m d'ou 2 * partiEntierInferieur(q/2)(L+m^2b)+qmax(m^3 * w, L+m^2b)

La taille d'un bloc est la racine carrée du nombre de processus. De plus l'ordre de la matrice est égal au nombre de processus. Donc on a nombre_processus * nb_processus éléments.

Mesures

Nous avons mesuré les temps d'execution de différentes portions de notre programme. Nous avons condensé ces résultats en des moyennes qui nous ont

semblées représentatives. Voici les différents algorithmes implémentés :

Linéaire

Real	User	Système
0m3.420s	0m2.115s	0m1.852s

Fox

Real	User	Système
0m2.683s	0m1.680s	0m1.210s

Initialisation	Scatterv	Algorithme	Gatherv	Temps total
2.421600s	0.000797s	0.121387s	0.000820s	2.544604s

Cannon

Real	User	Système
0m2.746s	m1.770s	0m1.190s

Initialisation	Scatterv	Algorithme	Gatherv	Temps total
2.493193s	0.000790s	0.109243s	0.003925s	2.607151s

Conclusion

A la vue de ces résultats, il nous a été impossible de déterminer un algorithme plus rapide qu'un autre. Le temps passé a éxécuter l'algorithme est négligeable par rapport à la phase d'initialisation de MPI et la lecture des matrices depuis le fichier.