TP parallélisme

Compte-Rendu 16/12/2008

Factorisation LU et descente-remontée

Encadrant :

Mathieu FAVERGE

Binome { Seifeddine HABASSI
Mohamed Amine EL AFRIT

Table des matières

1 Préliminaires

On souhaite résoudre en parallèle un système linéaire carré dense de la forme $Ax = b$.

La factorisation LU consiste à transformer la matrice A en un produit de matrices L U, où L est une matrice trianglaire inférieure et U est une matrice triangulaire supérieure. Pour cela, on s'appuie sur le fait qu'à chaque étape de la mthode de Gauss, il existe une matrice $L^{(k)}$ telle que : $A^{(k+1)} = L^{(k)} \times A^{(k)}$.

La matrice $U = A^{(n)}$, triangulaire supérieure, est donc telle que :

$$U = L^{(n)}L^{(n-1)}...L^{(2)}L^{(1)}A$$

et donc

$$A = L^{(1)-1}L^{(2)-1}...L^{(n-1)-1}L^{(n)-1}U = LU$$

L'algorithme de factorisation LU a l'avantage que les termes de L et U peuvent remplacer en mémoire les termes de A, la diagonale unité de L n'étant pas stockée.

2 Factorisation LU

2.1 Parallélisation de l'algorithme de factorisation

Comme indiqué dans l'énoncé, la matrice à factoriser est répartie par colonne suivant la distribution en serpentin. Dans l'algorithme séquentiel de factorisation LU, on peut distinguer 4 étapes :

1. La recherche du pivot dans la première colonne du bloc de matrice pas encore factorisé.
2. L'échange de la ligne du pivot pour la placer en haut du bloc de matrice pas encore factorisé.

3. La division par le pivot de chaque élément de cette colonne, ce qui nous donne une nouvelle colonne factorisée de L.
4. La mise à jour du nouveau bloc non factorisé : À chaque ligne du nouveau bloc non factorisé, on retranche la ligne nouvellement factorisée coeffcientée par l'élément situé à l'intersection de la ligne que l'on traite et de la colonne factorisé.

Les étapes 1 et 3 sont réalisées entièrement par le processeur qui détient la colonne que l'on factorise, à l'aide de données contenues dans la colonne. Aucune communication n'est donc nécessaire. En revanche, les étapes 2 et 4 concernent tous les processeurs. Pour exéctuer l'étape 2, ils doivent connaître l'indice du pivot, qui n'est connu que par le processeur détenant la colonne factorisée. Ce processeur va donc exéctuer un broadcast de cet indice. De la même manière, à l'étape 4 les éléments de la première colonne du bloc non factorisé doivent être connus de tous. Le processeur détenant cette colonne la diffuse donc.

2.2 Optimisation des performances à l'aide des BLAS

Chacune des étapes décrites précédemment peut être réalisée simplement à l'aide des BLAS afin d'optimiser l'utilisation du cache.

- La recherche du pivot dans une colonne est implantée à l'aide de la fonction *cblas_isamax*. Cette routine *BLAS* calcule en effet l'indice du plus grand élément dans un vecteur.
- La routine *cblas_ ?swap* permet d'échanger en mémoire deux vecteurs. Cela nous permet d'implémenter la seconde étape. Comme nos matrices sont stockées par colonnes, on utilise un stride égal à la hauteur des matrices afin de d'échanger des lignes de matrice.
- Pour diviser chaque élément de la colonne factorisée par le pivot on utilise la routine *BLAS cblas_ ?scal* qui permet de multiplier un vecteur par un coeffcient.
- Enfin, la mise à jour du bloc peut être exéctuée en une seule étape par la foncction *cblas_ ?ger*. En effet, il suffit de voir que l'opération exéctuée revient à retrancher à ce bloc le bloc obtenu en exéctuant le produit de la colonne factorisée par la ligne factorisée. La fonction *cblas_ ?ger* permet justement d'ajouter à une matrice le résultat obtenu en coeffcientant le produit d'un vecteur et de la transposée d'un autre.

2.3 Implantation

L'algorithme de factorisation LU distribué est implanté dans le module *facto.c*. Chaque processus charge le morceau de matrice qui lui est affecté à l'aide des fonctions de gestion de matrices. (Voir section 4) Chacun itère ensuite sur l'ensemble des colonnes de la matrice globale. Le processeur qui possède la colonne courante exéctue les étapes 1 à 4 tandis que les autres n'exéctuent que les étapes 2 et 4. L'échange du pivot ainsi que des colonnes s'exéctuent à l'aide de la fonction de communication *broadcastColumn* définie dans le module *matrix.c*.

La résolution est exéctuée "sur-place", on écrit directement les éléments de L et de U à la place des éléments de la matrice de départ. Les éléments diagonaux de L, tous égaux à 1 par convention, ne sont pas stockés. On peut alors écrire le résultat dans le fichier de sortie. Chaque processus écrit sa portion de matrice au sein du fichier

3 Résolution des équations de type Ax=b

Nous avons maintenant une matrice A factorisée sous la forme d'une matrice LU où L est triangulaire inférieure et U triangulaire supérieure. Nous devons donc résoudre LU x =b. Puisqu'on sait résoudre rapidement des équations de type Ax = b ou A est un matrice triangulaire, on décompose la résolution en deux étapes. On commence par résoudre l'équation Ly = b et on a ensuite, par identification, U x = y .

3.1 Parallélisation de l'algorithme de descente-remontée

Prenons pour exemple l'algorithme de descente qui consiste à résoudre l'équation Ly =b. L'algorithme de remontée étant tout à fait similaire.

Les colonnes de la matrice L sont réparties comme précédemment. Chaque processus détient par ailleurs une partie des vecteurs y et b qui sont stockés le même tableau. En effet les valeurs de b sont écrasées au fur et à mesure que les valeurs de y sont calculées. Les indices des éléments du vecteur b stockés par chaque processus coincident avec ceux des colonnes qu'ils détiennent. Par exemple, si un processus stocke les colonnes 3, 4 et 7 d'une matrice, les indices 3, 4 et 7 du vecteur seront stockés par le même processus. On travaille ligne par ligne, et pour chaque ligne on distingue deux étapes :

1. Sommer le produit de chaque élément précédent la diagonale de la matrice par l'élément de y déjà calculé correspondant. (Ayant le même indice que l'indice de colonne de l'élément de la matrice)
2. Retrancher cette somme à b et diviser le résultat par l'élément diagonal (égal à 1 dans le cas de L). On obtient ainsi un nouvel élément de y .

La première étape peut être réalisée de manière distribuée puisque les processus détiennent les éléments de y correspondant à leurs colonnes. Chaque processus peut donc calculer une "sous-somme" locale. La seconde étape est exéctuée par le processus détenant l'élément de y calculé. Il a besoin de connaître la somme totale, c'est à dire la somme de toutes les "sous-sommes" exéctuées par chaque processus. Cette information lui est communiquée à l'aide de la primitive MPI reduce qui permet à tous les processus d'envoyer une valeur à un processus qui en recevra la somme.

3.2 Optimisation des performances à l'aide des BLAS

L'étape 1 de l'algorithme décrit correspond à un produit scalaire entre les éléments de y déjà calculés et la ligne de la matrice que l'on traite. Cette opération peut être effectuée à l'aide de la routine *BLAS ?dot*.

3.3 Implantation

3.3.1 Descente

L'algorithme de descente est constitué de ces différentes étapes :
- Récupération de la matrice locale et du vecteur local au processus
- Pour toute les lignes k de la matrice globale
  - On exéctue un produit scalaire entre le début (jusqu'à la diagonale non incluse) de la ligne locale et le début du vecteur y local. Ce produit est exéctué au moyen de la routine blas cblas_?dot.
  - On rassemble la somme de globale (sommes des produits scalaires de chacun des processus) chez le processus détenant l'élément k du vecteur avec la primitive MPI_Reduce.

- Le processus détenant l'élément k du vecteur met alors à jour le vecteur y avec le résultat de l'opération $y_k = b_k - sommedesproduitsscalaires$

A la fin de la descente, le vecteur local qui contenait b contient y.

3.3.2 Remontée

L'algorithme de remontée consiste à résoudre l'équation U x = y . U est une matrice triangulaire supérieure contenant aussi les éléments de la diagonale (contrairement à L). L'algorithme de remontée se déroule suivant ces différentes étapes :

- Pour toute les lignes k de la matrice globale en partant de la fin
  - On exéctue un produit scalaire entre la fin (à partir de la diagonale non incluse) de la ligne locale et la fin du vecteur x local. Ce produit est exéctué au moyen de la routine blas cblas_?dot .
  - On rassemble la somme de globale (sommes des produits scalaires de chacun des processus) chez le processus détenant l'élément k du vecteur avec la primitive MPI_Reduce.
  - Le processus détenant l'élément k du vecteur met alors à jour le vecteur x avec le résultat de l'opération $x_k = y_k - \frac{sommedesproduitsscalaires}{U_{k,k}}$.
- A la fin des calculs on regroupe tout les résultats locaux à chaque processus dans un seul fichier de vecteur.

4 Gestion des matrices

Afin que les différents processus puissent effectuer diverses opérations sur les matrices et les vecteurs, nous avons implémenté un module de gestion de matrices (matrix.c). Ce module gère l'allocation et la libération mémoire, l'initialisation et les opérations sur les matrices. Les vecteurs sont représentés par des matrices de largeur 1.

Nous avons défini une structure de matrice que nous avons nommée matrix_t. Cette structure contient les champs :

- width : le nombre de colonnes de la matrice
- height : le nombre de lignes de la matrice
- array : un pointeur sur les données la matrice

Les données de la matrice sont stockées sous la forme d'un bloc de width*height éléments, les éléments étant rangés par colonnes. Les éléments de la matrice sont type MAT_DATA_TYPE que l'on peut définir à l'aide d'un #define (float, ou double par exemple) dans le module *matrix.h*.

4.1 Allocation mémoire et initialisation

La fonction *newMatrix* permet d'allouer de la mémoire à une matrice. Elle prend en paramètre la largeur et la hauteur de la matrice voulue, alloue de la place pour la structure matrix_t, puis fait de même pour le tableau de données. La quantité de mémoire allouée dépend de la taille de la matrice. Toutes les valeurs de la matrice sont initialisées à 0. Une fonction de désallocation est aussi proposée.

4.2 Format de stockage d'une matrice dans un fichier

Tous les processus doivent accéder aux matrices à traiter afin de récupérer les morceaux qui leur sont attribués. Pour ce faire, nous stockons les matrices dans des fichiers auxquels ont accès tous les processeurs. Le stockage est réalisé de manière binaire, c'est à dire que nous générons directement les fichiers avec des *fwrite*. Les

premiers éléments d'un fichier de matrice dont sa taille (hauteur et largeur).

Les données de la matrice sont stockées colonne par colonne en file indienne dans la matrice. On prend en compte la taille de chaque élément (float, double ...) afin de pouvoir précisément récupérer les données du fichier. Les dimension de la matrices sont en effet stockées sous la forme d'entiers tandis que chaque élément est stockée sous la forme de MAT_DATA_TYPE. Le module *init.c* permet de créer des fichiers de test.

On entre en ligne de commande la taille et le type de matrice souhaité (random int, random float, exemple, identity) et une matrice est générée puis stockée sous forme d'un fichier dont le nom est aussi entré en paramètres.

4.3 Répartition sur plusieurs processus

4.3.1 Le serpentin

Les colonnes de la matrice sont réparties sur les différents processus suivant la distribution du serpentin

Chaque processus P_i stocke dans une matrice locale, de manière contigue, les colonnes qui lui sont allouées. Différents opérateurs permettent d'associer le numéro de colonne local à un processus à son numéro dans la matrice complète.

- La fonction *getNbColumns* permet de déterminer le nombre total de colonnes qui vont être affectées à un processus donné.
- La fonction *whoHasColumn* permet de déterminer le rang du processus détenant une colonne donnée de la matrice complète.
- La fonction *getLocalColumnIndex* renvoie l'indice de colonne local au processus correspondant à un indice de colonne de la matrice globale. Le résultat n'est valide que si le processus détient réellement la colonne.
- La fonction *getGlobalColumnIndex* est la fonction inverse. Elle renvoie l'indice de la colonne dans la matrice globale correspondant à un indice dans la matrice locale.

4.3.2 Chargement et stockage dans un fichier

Un ensemble de fonctions permettent de passer directement du format de fichier avec une matrice globale à un format en mémoire, réparti pour chaque processus selon la distribution en serpentin.

- La fonction *getMatrixColumnsFromFile* charge les colonnes de la matrice globale affectées à un processus donné et les place côte à côte dans la matrice locale au processus. Les paramètres en entrée sont le chemin du fichier stockant la matrice globale, le rang du processus et le nombre total de processus.
- La fonction *writeMatrixColumnsToFile* effectue l'opération inverse c'est à dire qu'elle écrit au bon endroit, dans le fichier contenant la matrice globale, l'ensemble des colonnes locales d'un processus.
- Les fonctions *getMatrixRowsFromFile* et *writeMatrixRowsToFile* permettent d'effectuer la même opération pour une distribution serpentin par ligne. On l'utilise ici pour charger et écrire nos vecteurs pendant l'étape de résolution.

4.4 Affichage

L'affichage se fait à l'aide du module *showMatrix.c*. On lui passe en paramètre le fichier contenant la matrice ou le vecteur à afficher.