Projet final – Éléments Logiciels pour le traitement des données massives
7 février 2021 - *Florian Schirrer, Caroline Boudier*

Dans ce rapport nous présentons les principaux résultats obtenus dans le notebook joint. Nous nous intéressons à l'algorithme de la décomposition de Cholesky en montrant comment la parallélisation peut permettre d'optimiser plusieurs variantes de cette décomposition.

## I) Principe de l'algorithme de Cholesky

Toute matrice A symétrique définie positive peut être décomposée comme $A = LL^T$ où L est une matrice triangulaire inférieure (dont les éléments diagonaux sont positifs). L'intérêt de cette décomposition est par exemple de résoudre plus facilement des systèmes de la forme $AX = B$ où A est SDP.
Pour résoudre ce système on utilise une méthode récursive avec une forte dépendance entre les différentes étapes. Cette dépendance implique que l'ordre dans lequel on effectuera les opérations sera crucial pour la parallélisation.

## II) Algorithmes étudiés

Deux grandes approches se distinguent pour calculer une décomposition de Cholesky : l'approche par les lignes (Cholesky Banachiewicz) et l'approche par les colonnes (Cholesky Crout). On indique les deux pseudo-codes ci-dessous ainsi qu'une représentation graphique associée permettant de mieux comprendre leurs dépendances et fonctionnement.

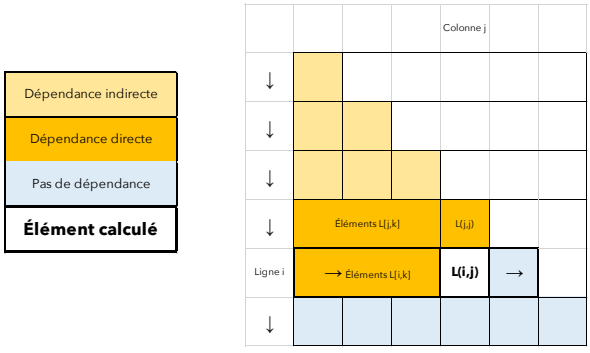
<table>
<tr><th>Cholesky Banachiewicz (lignes)</th><th>Cholesky Crout (colonnes)</th></tr>
<tr><td>

**Pour i de 1 à n :**<br>
 **Pour j de 1 à i :**<br>
 $somme = 0$<br>
 **Pour k de 1 à j − 1 :**<br>
 $somme = somme + L_{i,k} \times L_{j,k}$<br>
 $Si\ i = j :\ L_{i,j} = \sqrt{A_{i,i} - \text{somme}}$<br>
 $Si\ i \neq j : L_{i,j} = \frac{A_{i,j} - somme}{L_{j,j}}$

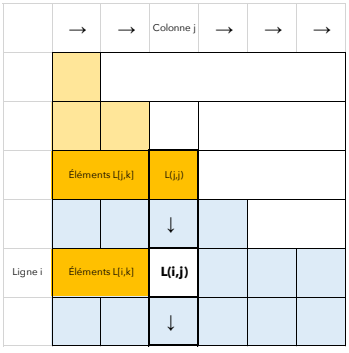
</td><td>

**Pour j de 1 à n :**<br>
 $somme = 0$<br>
 **Pour k de 1 à j − 1 :**<br>
 $somme = somme + L_{j,k} \times L_{j,k}$<br>
 $L_{j,j} = \sqrt{A_{j,j} - \text{somme}}$<br>
<br>
 **Pour i de j + 1 à n :**<br>
 $somme = 0$<br>
 **Pour k de 1 à j :**<br>
 $somme = somme + L_{i,k} \times L_{j,k}$<br>
 $L_{i,j} = \frac{A_{i,j} - somme}{L_{j,j}}$

</td></tr>
<tr><td>




</td><td>




</td></tr>
<tr><td>

On parcourt toutes **les lignes** du triangle inférieur. Sur chaque ligne on parcourt ensuite les colonnes.
Comme représenté sur le schéma, un élément L[i][j] est calculé à partir des éléments L[i][k] et L[j][k] (pour k variant dans 1:j-1) et de l'élément L[j][j]. L'algorithme implique une dépendance croissante et la nécessité de suivre un ordre précis à la fois sur les lignes et les colonnes.

**Conclusion :** l'algorithme est difficilement parallélisable.

</td><td>

On parcourt toutes **les colonnes** puis toutes les lignes Un élément L[i][j] est calculé à partir des éléments L[i][k] et L[j][k] (pour k variant dans 1:j-1) et de l'élément L[j][j]. Certes le calcul des colonnes doit respecter un ordre précis. Mais une fois fixé sur une colonne le calcul de chaque élément est indépendant.

**Conclusion :** On pourra donc envisager une parallélisation au niveau de la boucle des i (boucle sur les lignes).

</td></tr>
</table>

Pour aller plus loin en proposant une deuxième version de l'algorithme de Cholesky Crout on va s'intéresser à la représentation de A utilisée. Pour l'instant A est représentée comme une matrice : la référence A[i][j] utilise un tableau

dans un tableau. De manière schématique, pour un élément donné, l'algorithme va **d'abord** chercher la ligne correspondante **puis** la colonne pour y accéder. Cette approche fournit de mauvaises performances en termes de cache mémoire puisque les lignes sont allouées dans des espaces de stockages différents. De plus, si on intègre des pointeurs, cette représentation matricielle des données requiert l'utilisation d'un double pointeur : un sur les lignes et un sur les colonnes ce qui est également assez coûteux.
Or la décomposition de Cholesky à laquelle on s'intéresse est très gourmande en mémoire : le cache joue un rôle crucial dans la performance de nos implémentations. Afin d'optimiser l'utilisation de la mémoire et accélérer l'algorithme, nous proposons donc d'utiliser l'allocation **contigüe**. Une telle approche permet de représenter une matrice à deux dimensions en un tableau/vecteur à une dimension. En utilisant une indexation par les lignes (plus efficace que par les colonnes) l'élément A[i][j] deviendra l'élément A[$i \times n + j$]. Tous les éléments se trouveront ainsi dans le même espace de mémoire ! De plus nous pourrons nous contenter d'un **pointeur unique.**

En résumé on va donc implémenter :

- Une version simple et une version parallélisée pour l'algorithme de Cholesky-Crout
- Une version simple et une version parallélisée pour l'algorithme de Cholesky Crout sous sa forme « vectorisée »

## III) Mise en place de la parallélisation

Une fois mises en place un certain nombre d'optimisation avec Cython (définition des types de variable, retrait des décorateurs wraparound, boundschecking etc), on peut s'intéresser à l'approche la plus pertinente pour la parallélisation avec OpenMP.

Comme décrit sur le schéma de la partie I, l'algorithme de Crout présente un fort potentiel de parallélisation sur le calcul des lignes : on va donc paralléliser sur la boucle des i à l'aide de la fonction prange. Afin que la parallélisation soit efficace on supprime le verrou (Global Interpreter Lock), cela ne pose pas de problème puisque chaque thread met à jour des objets indépendants, il n'y a donc pas de risque de race condition. Plusieurs paramètres peuvent être ajustés :

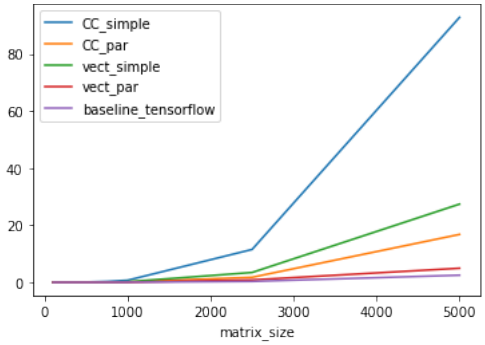
- **Nombre de threads**. Notre machine est équipée d'un processeur Intel Core i5 quatre cœurs nous permettant de lancer simultanément jusqu'à huit threads.
- L'**ordonnancement** des threads (schedule). On distingue principalement deux approches : statique et dynamique. Avec l'approche statique les itérations sont divisées en parts égales et assignées aux threads dans l'ordre : aucune synchronisation n'est nécessaire. Avec l'approche dynamique, on assigne une nouvelle itération à un thread dynamiquement une fois qu'il a fini de traiter l'itération précédente. Cette méthode peut être pertinente dans le cas où les itérations ont des tailles très variables ou imprévisibles mais elle a aussi un coût d'exécution (puisqu'elle nécessite une synchronisation). Dans notre cas l'ordonnancement statique est le plus pertinent car à j fixé les itérations sur les i nécessitent exactement le même travail de la part du thread (cf schéma).
- Le **cut-off**. Intégrer un cut-off consiste à donner à l'algorithme une condition pour laquelle la parallélisation s'arrête et l'algorithme continue de manière séquentielle. Elle est pertinente dans le cas de tâches trop granulaires (c'est-à-dire que chaque itération est très rapide et finalement les exécuter de manière séquentielle est plus rapide que de paralléliser en raison du coût de synchronisation). Ici on fixera le cut-off de manière à ce que les dernières colonnes (proportion p de colonnes) soient de nouveau calculées en séquentiel.

## IV) Résultats

On peut désormais analyser les résultats. On va pour ce faire comparer les temps d'exécution, le speed-up et l'efficacité de nos algorithmes[1].

***Figure 1*** : Temps d'exécution (s) et speed-up pour l'algorithme Cholesky Crout (CC) simple et parallélisé, l'algorithme vectorisé simple et parallélisé et la fonction tensorflow en fonction de la taille de la matrice. ***Figure 2*** : temps d'exécution (s).

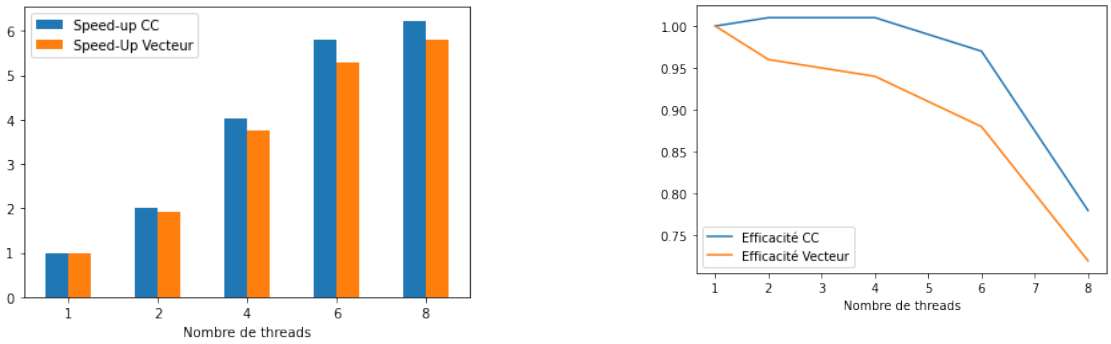
| matrix_size | CC_simple | CC_par | speed_up_CC | vect_simple | vect_par | vect_speedup | baseline_tensorflow |
|---|---|---|---|---|---|---|---|
| **100** | 0.001 | 0.000 | 1.384 | 0.000 | 0.000 | 0.372 | 0.000 |
| **500** | 0.088 | 0.031 | 2.848 | 0.022 | 0.007 | 2.887 | 0.003 |
| **750** | 0.295 | 0.092 | 3.195 | 0.078 | 0.031 | 2.516 | 0.010 |
| **1000** | 0.721 | 0.120 | 6.003 | 0.202 | 0.048 | 4.228 | 0.026 |
| **2500** | 11.517 | 1.761 | 6.541 | 3.450 | 0.843 | 4.095 | 0.318 |
| **5000** | 92.813 | 16.783 | 5.530 | 27.396 | 4.935 | 5.552 | 2.495 |




[1] Speed-up = $\frac{séquentiel}{parallélisé}$ Efficacité = $\frac{Speed\_up}{Nombre\ de\ threads}$

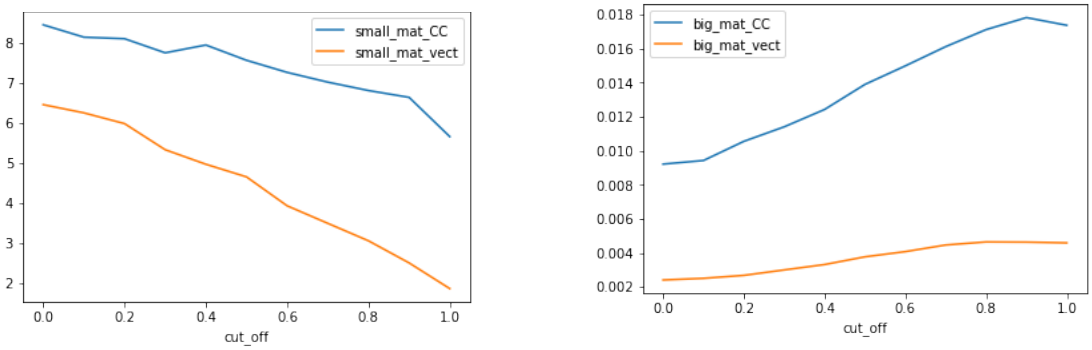
Comme prévu les versions parallélisées de chacun des algorithmes permettent une forte accélération. On remarque également que la version vectorisée est bien plus rapide que la version classique conformément à nos attentes : elle se révèle presque aussi rapide que Tensorflow (entre 2 et 3 fois plus lente).

***Figure 3 et 4*** : Speed-up et efficacité en fonction du nombre de threads (matrice de taille 3000).




Le speed-up est toujours inférieur au nombre de threads ce qui est attendu puisque la partie sérielle de l'algorithme constitue un goulet d'étranglement pour l'accélération (loi d'Amdahl). La version vectorisée de l'algorithme requiert de modifier la forme de la matrice (fonction np.reshape). La proportion de code exécuté en séquentiel est donc plus importante dans cette version ce qui explique que le speed-up soit légèrement plus faible pour l'algorithme vectorisé. On remarque également que dans les deux cas l'efficacité diminue avec le nombre de threads. On peut penser que plus le nombre de threads augmente plus le coût de synchronisation et d'accès à la mémoire augmente.

***Figure 5 et 6*** : Temps d'exécution en fonction du cut-off pour deux matrices (n=50 à gauche et n=300 à droite).




Pour rappel un cut-off de 0 signifie que la parallélisation a lieu pour l'intégralité des colonnes et un cut-off de 1 implique qu'il n'y a plus aucune parallélisation. On constate que pour une petite matrice (graph de gauche) le cut-off améliore la performance : les opérations sont trop granulaires pour être pertinentes pour une parallélisation et on a plutôt intérêt à privilégier une approche séquentielle. L'effet inverse est logiquement obtenu pour une matrice de taille plus importante (300*300) : plus le cut-off est fort moins la performance est bonne.

## V) **Conclusion**

En conclusion nous avons proposé deux versions différentes pour la décomposition de Cholesky, toutes deux parallélisées avec succès. Au-delà de la parallélisation nous avons montré que la prise en compte de l'allocation de la mémoire était cruciale pour améliorer la performance : le fait d'adopter l'approche contiguë nous a permis de limiter l'augmentation du temps d'exécution avec l'augmentation de la taille des matrices en nous rapprochant des performances de Tensorflow. Une prochaine étape pourrait consister à creuser la piste d'approches moins intensives en mémoire, par exemple en évitant le recours à une matrice intermédiaire L et en itérant directement sur la matrice A.