CALCUL HAUTE PERFORMANCE

POLYTECH SORBONNE

Rendu de mi-parcours

BERBAGUI Amine
HACHANI Ghassen

MAIN 4

3 avril 2022

Table des matières

1 Exploration des performances du code séquentiel

On cherche à explorer les performances du fichier *lanczos_modp.c*, c'est-à-dire repérer les endroits du code qui prennent le plus de temps lors de l'exécution. Pour cela, on utilise les outils présent dans le cours, à savoir **GNU Profiler** ainsi que l'outil **Perf Tools**.
On a donc :

- ✔ GNU Profiler :
  - 69% du temps dans **sparse_matrix_vector_product**
  - 20% du temps dans **orthogonalize**
  - 11% du temps dans **block_dot_products**
- ✔ Perf Tool :
  - 81% du temps de **sparse_matrix_vector_product** à la ligne 283
  - 29% du temps de **orthogonalize** à la ligne 299
  - 44% du temps de **block_dot_products** à la ligne 312

➔ Ligne 283 : `y[i * n + l] = (a + v * b) % prime;` ⟶ fonction sparse_matrix_vector_product
➔ Ligne 299 : `C[i * n + j] = (x + y * z) % prime;` ⟶ fonction matmul_CpAB
➔ Ligne 312 : `C[i * n + j] = (x + y * z) % prime;` ⟶ fonction matmul_CpAtB

Ainsi on a identifié les hots-spots du programme et l'on peut maintenant optimiser ces 3 fonctions de sorte à réduire le temps d'exécution avec une matrice donnée. Avec la matrice **TF16** par exemple, on a initialement un temps d'exécution d'environ 1m30s. On va donc essayer par la suite d'optimiser au maximum ce temps. On passera ensuite à des matrices plus grandes pour les tâches de parallélisation.

2 Optimisation du code séquentiel

On cherche maintenant à optimiser le code séquentiel avant de commencer le travail de parallélisation. Comme indiqué sur le sujet, le code peut être optimisé de plusieurs manières. L'une d'entre elle concerne le modulo **prime** qui est répeté dans les 3 lignes de commandes citées plus haut. Après les éclaircissements que vous nous avez gentillement fournies à la fin d'une séance de TD, on a quelque peu modifié les fonctions *matmul_ CpAB* et *matmul_ CpAtB*. En effet, ces deux fonctions effectuent des multiplications de matrices que l'on peut représenter comme suit :

$$C[i,j] = (C[i,j] + A[i,k].B[k,j]) \ \% \ \textbf{prime}$$
$$\text{or on sait que } A[i,k] < \textbf{prime} \text{ et } B[k,j] < \textbf{prime}$$
$$\iff C[i,j] < n.(prime-1)^2 \text{ or } n.(prime-1)^2 < 2^{64} \text{ qui est la taille maximale.}$$

Ainsi, on peut se passer du % **prime** dans notre 3ème boucle for,définir dans la 2ème boucle for la variable x car elle ne dépend pas de k et rajouter $C[i,j] = x \ \%$ **prime** en dehors de la 3ème boucle pour ne plus avoir à faire n opérations % **prime**.

Ainsi, en terme de performances, on a un léger gain de temps non négligeable du code séquentiel qui passe de 1mn 30 à 1mn 10 (matrice TF16).

Dans la même idée, appliquons ce même raisonnement à la fonction **SMVP**, abréviation de sparse_matrix_vector_product qui contient également des opérations modulo prime. Comme on l'as vu dans la 1ère partie, `y[i * n + l] = (a + v * b) % prime` représente une très grosse partie de l'exécution de cette fonction. On cherche donc à optimiser cette ligne. On commence par modifier la fonction *block_ lanczos* qui appelle la fonction **SMVP**. On pose **Mpad** qui est similaire à **Npad** sauf qu'il dépend du nombre de colonnes (ncols) puis on modifie **bloc_size_pad** qui vaut Npad si Npad > Mpad, Mpad sinon fois n. Puis on fait passer bloc_size_pad en paramètre de sparse_matrix_vector_product.

Dans la fonction **SMVP**, on crée un tableau (tab) de taille block_size_pad que l'on initialise à 0. Ainsi dans la boucle où se trouve le modulo, on va remplacer le tableau **y** par **tab** et l'on supprime le % **prime**. On crée une boucle for à la fin et pour chaque itération, on affecte à a la valeur de y[i] puis à y[i] la somme de a et **tab[i]** le tout modulo prime. Le tableau tab sert donc de transition entre y et a et à séparer les multiplications de matrices avec les opérations modulo prime.

Ainsi, en terme de performance toujours avec la matrice TF16, on passe de 1m10 à 28s ! On est passé de 1m30 au départ à 28s (temps/3) juste en optimisant les opérations de modulo qui représentaient le plus gros du temps d'exécution. Et tout cela avant même la parallélisation.

3 Parallélisation OpenMP

Passons maintenant à la parallélisation avec OpenMP seulement. Nous allons décrire brièvement les commandes ajoutées dans les fonctions concernées ainsi que les performances constatées.

1. **block_dot_products** : cette fonction contient initialement 4 boucles for, 2 d'initialisation à 0 des matrices vtAv et vtAAv et 2 autres de calcul. Ainsi, on a créer deux tableaux (tab1,tab2) de taille n.n que l'on initialise à 0 dans une même boucle et qui vont venir remplacer vtAv et vtAAv en argument des appels de la fonction *matmul_ CpAtB* dans les deux boucles qui suivent.Enfin, la dernière boucle permet de copier les valeurs de tab1 (resp tab2) dans vtAv (resp vtAAv) sans oublier modulo prime.
   On réserve une équipe de threads au départ. La 1ère et la dernière boucle ne nécessite qu'un `#pragma omp for` car il n'y a pas de conflit possible. Cependant la 2ème et 3ème boucle nécessitent une clause de réduction sur tab1 et tab2 car on doit s'assurer que tab1 contient la somme des tab1 privés de chaque thread, idem pour tab2.

2. **orthogonalize** : pour cette fonction, on réserve une équipe de threads sur les 5 dernières boucles for qui s'occupent de calculer la prochaine valeur de v et de p et qui peuvent donc être faite en parallèle. Donc on place tout bêtement un `#pragma omp for` avant chacune de ces boucles.

3. **SMVP** : On réserve une équipe de thread pour toute les boucles for. On place un `#pragma omp for` au dessus de la 1ère,2ème et dernière boucle car elles ne provoquent pas de conflits entre les threads. Puis la 3ème boucle qui contient une clause de réduction comme dans la fonction bloc_dot_product.
4. **block_lanczos** : dans cette fonction, on peut effectuer en parallèle la boucle for d'initialisation de Av,v,p et tmp. La boucle qui suit également, et enfin la dernière boucle de la fonction où l'on copie les valeurs de tmp dans v.

Performances

Finalement en rajoutant la parallélisation OpenMP comme indiqué ci-dessus, on a un temps d'exécution (toujours pour la matrice TF16) qui passe de 28s à 9s. Sans la parallélisation sur **SMVP**, on est à 19s ce qui montre bien l'impact de cette fonction dans l'algorithme. On peut souligner le fait qu'on a un meilleur temps juste avec l'optimisation séquentiel qu'avec la parallélisation OpenMP.
Voila ce qu'on a fais jusqu'à présent pour ce 1er rendu. On voulait s'assurer que les modifications faite jusqu'à présent ainsi que la parallélisation soient correctement implémentés avant de passer à la partie MPI.