Mines ParisTech - Ms HPC-AI

Projet d'Algèbre Linéaire Distribuée : Profiling de BoomerAMG

Julian AURIAC & Aymeric MILLAN

Cours de Christophe BOVET

17 Janvier 2022

Résumé

Voici le document regroupant nos benchmarks de BoomerAMG, inclus dans la librairie HYPRE.

L'objectif du projet est double : il s'agit d'abord d'appliquer les méthodes de résolution multigrilles algébriques de BoomerAMG sur des matrices de problématiques réelles au format `.mtx` (Matrix Market). Le deuxième objectif du projet est la réalisation de deux études d'extensibilités :

- **Faible :** En partant de 1 processus MPI et d'un problème de taille 256∗256, nous doublons la taille du problème **et** le nombre de CPU qui travaillent sur le problème.
- **Forte :** Avec un problème de taille fiche au format Matrix Market, nous augmentons le nombre de processus MPI/CPU.

Nous mesurons dans les deux cas les temps d'exécutions de la résolution d'un système de la forme $Au = f$, avec A la matrice de départ (i.e l'observation), f la solution vers laquelle on veut converger (i.e le résultat) et u le système que l'on cherche a trouver.

En plus de ces outils, ce projet apporte une vision générale sur les méthodologies de développement sur des clusters de calculs comme celui du CEMEF. Nous avons pu approfondir les concepts tels que la réservation des coeurs de calcul, le partage de ressources ou encore les scripts automatisés, ici avec `OAR`.

1 Récupérer le code

Le code est disponible sur ce repo GitHub. Vous pourrez aussi trouver le code et une archive dans nos repertoires personnels sur le CEMEF. Le code est déjà compilé sur le cluster.

Nous avons ajouté l'option `-file` à l'exécution qui permet de préciser un chemin d'accès relatif vers un fichier au format .mtx. Le code fonctionne pour des **matrices carrées** uniquement. Attention, avec les options par défaut (e.g `-gamma`, le cycle AMG utilisé, V par défaut), seulement les matrices diagonales permettent de converger vers une solution. Nous avons téléchargé quelques matrices de ce type dans le répertoire `matrices`, à la racinne du projet.

2 Logique d'implémentation

Pour le benchmarking, nous avons utilisé un script shell qui formate les données des logs OAR en fichiers csv. Ensuite, à l'aide de python et surtout de la librairie `pandas`, nous avons traité les données pour les visualiser ici. Les données brutes sont disponibles dans le répertoire `logs/` du projet.

Pour ce qui est de la lecture de fichier `.mtx` et le découpage sur les différents coeurs de calculs, nous avons utilisé la librairie Eigen. La difficulté était premièrement de s'approprier la librairie, notamment les méthodes pour accéder aux données brutes (i.e. `raw buffers`), dans le but de les envoyer à l'autre librairie, HYPRE. L'interfaçage entre les deux librairies a été pour nous la difficulté de ce projet (et donc l'axe d'amélioration).

Nous avons essayé de réutiliser au maximum le code qui était déjà développé par le professeur, vous trouverez le code commenté dans le fichier `tp.cpp`

3 Produit Matrice Dense & Vecteur

L'implémentation du produit matrice dense et vecteur a été la plus simple à réaliser, autant pour les communications MPI que la logique.

Le graphique ci-dessous montre l'évolution du temps de calcul en fonction du nombre de processeurs, pour différentes tailles de matrices. Les traits pointillés représentent le temps de calcul en séquentiel, les traits pleins représentent le temps de calcul du noeud MPI le plus lent. Les couleurs correspondent entre les temps séquentiels et parallèles.

Nous pouvons constater que, pour un noeud, les temps de calcul sont relativement similaires, mais dès que le nombre de noeuds augmente,le coût de calcul baisse drastiquement pour les matrices de grande taille. Cela est moins vrai pour les matrices de petites tailles.

4 Produit Matrice (CSR) & Vecteur

Pour le SpMV CSR, j'ai d'abord essayé d'utiliser un seul et même buffer pour envoyer les éléments aux différents nodes. Ce n'était pas une bonne idée, car le programme ne fonctionnait pas avec des matrices de trop grandes tailles. Le problème venait d'une commande `clear()` sur les `vector` contenant les buffers à envoyer. Parfois, si la taille du buffer était trop grande, comme l'envoi était fait en asynchrone, le buffer avait le temps d'être `clear` avant d'être réellement envoyé. Cette erreur était très compliquée à débugger car si les matrices étaient trop petites, l'envoi était fait quasi instantanément et tout fonctionnait bien. Je suis donc parti sur du synchrone pour l'envoi des valeurs, de toute façon, pour un problème de cette taille, les communications sont bien trop couteuses et biaisent la durée de calcul. C'est pourquoi elles ne sont pas prises en compte dans les benchmarks.

De plus, il est bon de noter que le découpage a été réalisé par lignes et non pas par valeurs. Bien entendu, cela n'est pas fidèle à la logique de la parallélisation MPI qui voudrait que les charges soient équilibrées pour chaque processus. Cependant, comme nous résolvons un produit par une matrice Laplacienne, qui est tridiagonale, les charges seront équilibrées et nous n'avons pas à gérer les pondérations des valeurs entre deux processus si une ligne est coupée en deux.

Pour varier un peu, ce graphe affiche le temps de calcul en fonction de la taille de la matrice, pour différents nombres de coeurs. Les variables sont donc inversées par rapport au premier graphique. La courbe en pointillée correspond au temps de calcul séquentiel. On constate que le calcul SpMV est toujours plus rapide que celui de DenseMV avec MPI, et que le temps de calcul réduit drastiquement au début (lorsqu'on passe de 1 a 2 nodes), puis ce temps de calcul réduit de moins en moins.

5 Life-of-boids

Pour l'application life-of-boids, j'ai isolé la boucle principale de calcul dans un exécutable `main_profiling`, en retirant la partie graphique. J'ai ensuite chronométré les exécutions d'un tour de boucle en fonction de la taille de départ du "flock". Cela correspond au nombre d'agents dans notre système et ce paramètre est intrinsèquement lié au nombre de calculs effectués à chaque tour de boucle.

Les calculs ont été réalisés sur un **Intel Core i7-10870H CPU @ 2.20GHz**. Possédant **8 coeurs physiques, et 16 coeurs logiques** grâce à l'hyperthreading.

Ce premier graphique trace le temps d'éxecution en fonction de la taille de la population (flock) et du nombre de threads. Les traits pointillés correspondent à l'éxecution avec OpenMP, les traits pleins à TBB. Enfin, le trait plein noir correspond à l'éxécution séquentielle.

Nous pouvons constater que passer d'une exécution séquentielle à une éxecution parallèle avec ne serait-ce qu'avec 2 threads apporte un gain de performance considérable.

Cependant, intéressons nous à la performance des deux librairies, OMP et TBB, pour ceci, appuyons nous sur ce même graphique sans la version séquentielle. On éxécute 100 tours de boucle cette fois : la finalité des calculs est de l'affichage graphique, donc ils vont être réalisés un grand nombre de fois à la suite. C'est ce que nous essayons de simuler ici.

Ce graphique nous montre que, sur cette machine et ce processeur, TBB semble être légèrement meilleur que OMP. Lorsqu'on passe de 2 à 4 threads ou de 4 à 8, la durée d'exécution tend à être divisé par deux. Cela parait logique car la machine possède 8 coeurs. Sur 100 exécutions, l'hyperthreading semble avoir plus d'impact, le gain de performance entre 8 et 16 threads reste intéressant. Comme nous l'avons vu sur le graphique précédent, au delà d'un certain seuil, augmenter le nombre de threads n'est pas utile.

Conclusions

METTRE CONCLUSION TECHNIQUE ICI
L'utilisation d'une librairie telle que HYPRE, qui masque l'implémentation des communications MPI est quand même assez agréable et rassurant.

Axes d'amélioration

- Split par value et pas par row car on est en CSR - eviter le loading de la matrice pour tous les CPU pour voir la différence entre le coup de la communication, ou alors le chargement dans toutes les RAM (qui est la solution naive) - Donner la possibilité d'importer le vecteur solution depuis un fichier -