RAPPORT PROJET: Sherical Harmonic Representation of Earth Elevation Data

Ranto RANAIVO
M1 SFPN

Pour chacun des executions de programme dans ce rapport, nous considèrerons :
[
frontend : nancy

compilation :
Via le fichier Makefile avec la commande ‘make’
Pour le programme parallélisé, la ligne CC = mpicc est rajoutée

Reservation de ressources :
oarsub -p grisou-9 -l core=16 -I

Fichiers:
pour --data :
/srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv

pour --model :
m.txt
]

I – La parallélisation :

La première étape de ce projet était de repérer la partie parallélisable où le programme passe le plus de temps. Après avoir mesuré le temps d’éxection des principaux fonctions, on constate que c’est dans la fonction QR_factorize() que le programme s’éxecute le plus longtemps.
Pour faire fonctionner le programme sur plusieurs coeurs dans l’optique de diminuer ce temps d’éxecution, il faut :

- Le processus 0 qui initialise la matrice A (charger les données et faire les calculs pour remplir la matrice entière)
- La matrice A est équitablement répartie entre les processus en utilisant MPI_Scatter() sachant que les autres processus n’allouent que leur partie et non la matrice entière
- Chaque processus fait le calcul de leur partie dans la fonction QR_factorize()
- A la fin de la fonction, on regroupe les parties dans la matrice A du processus 0 via MPI_Gather()
- Seul le processus 0 continuera l’execution du programme après la fonction

II – Expériences réalisées :

Après plusieurs tests sur différentes valeurs des paramètres ‘npoint’ et ‘lmax’, les meilleurs en terme de précision sont respectivement 5,000,000 et 12.

Executions :

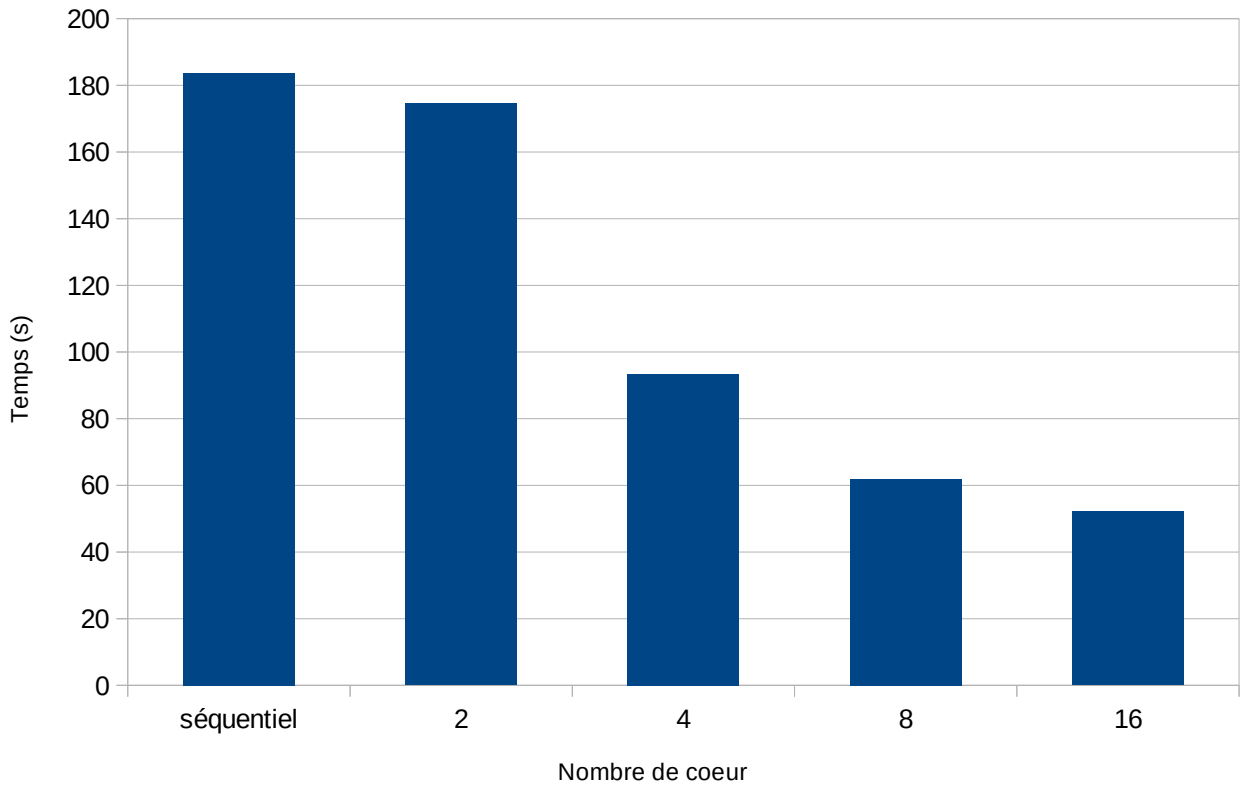
Sur le programme séquentiel :

./model --data /srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv --model m.txt --npoint 5000000 --lmax 12

Sur le programme parallèle :

mpiexec -mca pml ^ucx --n X ./model --data /srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv --model m.txt --npoint 5000000 --lmax 12 (avec X le nombre de coeurs à utiliser)

Nombre de coeurs	séquentiel	2	4	8	16
Temps (s)	183,7	174,6	93,5	62	52,2



Avec seulement deux coeurs, on observe une baisse du temps d'execution assez faible. Cela peut être du à l'implémentation de l'algorithme de parallélisation. En effet, en se penchant sur l'algorithme dans QR_factorize(), on s'aperçoit qu'un processus ne fait plus rien lorsque toute les colonnes de sa partie ont été traitées. Chaque itération de la boucle correspond à une colonne de la grande matrice.

Il faut quadrupler le nombre de coeurs pour que le temps d'éxecution soit réduit d'environ la moitié, on le constate notamment sur X=4 et X=16.

Les résultats de ces expériences nous permettent de dire que la parallélisation fonctionne bien parce que le temps d'execution a énormemment baissé lorsqu'on a utilisé 16 coeurs. La question qui peut se poser est : est-ce optimal ? La réponse est non à cause des processus qui restent en 'idle' lorsqu'ils ont terminé leur partie.

III – Difficultés rencontré :

La boucle dans la fonction QR_factorize() n'est pas parallélisable car les données de l'étape i+1 dépendent de l'étape i. De ce fait, la parallélisation ne vise pas à réduire les itérations de la boucle mais à réduire au maximum le temps passé dans chaque itération.

La fonction QR_factorise() est un code qui boucle sur le nombre de colonne de la matrice. Pour pourvoir faire le calcul dans de sa partie, le processus x a besoin de données qui se trouve dans la colonne i de la grande matrice. Sachant que la matrice a été divisé équitablement, si la colonne i n'appartient pas au processus x, le calcul sera faussé car x aura des données différentes.
Pour éviter cela, on a mis au point un algorithme : si la colonne i appartient au processus x, celui-ci partagera les données nécessaires pour faire le bon calcul à tous les autres processus.

Au delà des valeurs 5,000,000 et 12 pour les paramètres 'npoint' et 'lmax', pour le programme parallèle, le processus 0 est arrêté car la matrice est trop grande (pas assez de mémoire) donc l'execution du programme échoue. Ce problème de gestion de mémoire fait aussi que l'algorithme n'est pas encore optimal.

On observe aussi que si au début (2 coeurs) on gagne grandement en temps d'execution en doublant le nombre de coeurs utilisé, si on continue à doubler, le gain de temps devient de plus en plus faible. On peut alors estimer qu'à partir d'un nombre de coeurs assez grand, le gain de temps d'éxecution est négligeable donc on utilisera beaucoup de ressources pour pas grand chose.

IV – Amélioration de la précision :

Le programme parallèle est restreint par la mémoire. Il faut donc trouver autre chose pour améliorer la précision du programme. Pour ce faire, au lieu de prendre les n-premiers points du fichier de données, on va répartir les points de façon équitable.

- Compter le nombre de données dans le fichier c'est-à-dire le nombre de lignes
- Savoir combien de lignes on doit sauter dans le fichier avant de lire la donnée. Pour cela il faut faire le rapport entre le nombre de lignes du fichier et le nombre de points à lire.
  k = nb_lignes/nb_point_a_lire.
- Lire les n points en sautant k lignes à chaque fois.

Avec cette technique la précision est améliorée et le temps d'execution du programme parallèle reste le même. Cependant, on observe un problème qui ne dépend pas des paramètres 'npoints' et 'lmax' : la lecture du fichier.
En suivant cet algorithme :

- on lit tout le fichier une première fois pour compter les lignes
- on revient au début
- on relis presque tout le fichier une deuxième fois pour prendre les données sachant que « sauter » une ligne revient à lire la donnée mais ne pas la stocker.

Sur les fichiers assez petit (ETOPO1_small/med/hi.csv) Ce n'est pas vraiment un problème. Mais sur le fichier ETOPO1_ultra.csv qui compte plus de 200,000,000 données, la lecture de tout le fichier peut prendre beaucoup de temps (>4mn). La question était alors de savoir si la précision gagné vaut le coup d'attendre plus de 4mn juste pour la lecture du fichier.

Execution de validate :

./validate --data /srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv --model m.txt --lmax 12

	Average Error	Max error	Standart Deviation	Temps d'execution
Répartition équitable des données	849,26m	8756,15m	1116,97m	52s + ~4mn
Répartition NON équitable des données	5.2374e+14 m	9.71256e+15m	1.41748e+15m	54,7s

On voit qu’il y a une grande différence lorsqu’on réparti équitablement les données. Plus la taille des données est élevée, plus cette différence sera aussi élevée. En effet, en ne prenant que 5,000,000 points sur 233,280,000 on couvre à peu près toutes les données lorsqu’on les répartis équitablement, dans le cas contraire on ne couvre qu’une partie du début. De ce fait, le temps d’attente pour la lecture du fichier est largement compensé car on a obtenue une différence de grandeur ~10^13.

V – Conclusion :

Le programme parallèle répond à nos attentes c’est-à-dire diminuer le temps d’éxectution du programme lorsqu’on le lance sur plusieurs coeurs. On a aussi observé que le gain de temps devient de plus en plus faible même si on augmente le nombre de coeur.
L’algorithme n’est pas encore optimale du fait que les processus deviennent inactifs à certains points de l’éxecution et que si on prends des paramètres trop grand, il se peut qu’il n’y ait pas assez de mémoire.