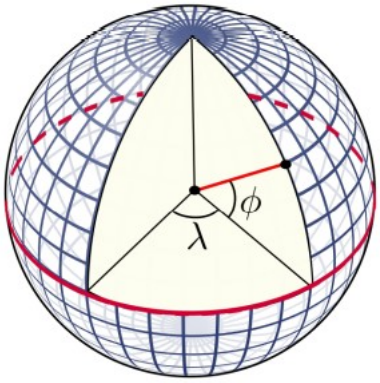
Sorbonne Université

Parallel Programming

Spherical Harmonic Representation of Earth Elevation Data



Méline Trochon
Anatole Vercelloni

Teacher : Charles Bouillaguet

Octobre 2022 — Novembre 2022

Table des matières

1 Introduction

Le but de ce programme est d'effectuer la résolution d'un système linéaire afin d'avoir une approximation de l'altitude d'un point sur Terre. Pour cela, nous avons les fichiers ETOPO avec une série de points, et comme paramètres npoint et lmax qui sont respectivement le nombre de points que l'on prend des fichiers pour l'approximation et l'ordre à laquelle on effectue le calcul.

Le programme séquentiel effetue donc la résolution d'un système linéaire $(Ax = v)$ en utilisant une QR-factorisation. Il part d'une matrice A de taille npoint x nvar ($nvar = (lmax+1)^2$) et d'un vecteur v de taille npoint en lisant les points dans un fichier de données. À partir de cette matrice, il calcule la QR-factorisation, à l'aide de matrices de Householder. Puis, il résout l'équation en calculant tout d'abord $Q^T v$ puis en calculant le vecteur x par résolution triangulaire.

L'objectif est donc d'utiliser grid'5000 pour paralléliser ce code dans le but d'améliorer aussi bien la gestion de la mémoire que le temps de calcul.

2 Parallélisation

2.1 Problématique de mémoire

Pour la mémoire, on remarque facilement que la matrice A est ce qui prend de loin le plus de place. Elle est de taille npoint x nvar, ce qui représente l'écrasante majoritée de l'espace mémoire alloué. En effet, les autres structures de données, sont au maximum de l'ordre de npoints, ce qui est négligeable devant la taille de A.

L'idée est donc de répartir A entre tous les processeurs de manière équilibrée. Un choix s'impose alors, diviser par colonne ou par ligne. Pour la mémoire, ce choix ne change abolument rien, nous avons choisis une division par ligne, dont nous détaillerons la raison plus tard.

Nous avons d'un côté implémenté un code où la matrice A est divisée par groupe de lignes (dans model_par3.c) et un autre code où la matrice est divisée par cycle de lignes. (dans model_par4.c) Puis nous avons testé les fonctions et comparer les temps d'exécution.

2.2 Problématique de temps

Pour le temps d'exécution, nous avons commencé par chercher les fonction/parties du code qui prenait le plus de temps. Pour cela, nous avons utilisé une option de compilation qui donne le temps d'execution pour chaque fonction. Les résultats étaient clairs, la fonction QR_factorize prenait 99% du temps d'execution. Il était donc essentiel pour nous de réfléchir à la subdivision en fonction de cette fonction. Puis paralléliser le reste seulement après si c'est possible (étant donné le fait que le temps des autres fonctions est négligeable devant celui de QR_factorize).

3 Division de la matrice

Nous allons parler dans cette partie des différentes possibilités qu'on a imaginé pour répartir la matrice entre les processeurs, et les raisons des différents choix que l'on a pu faire. On a commencé par analyser QR_factorize pour les raisons données précedemment.

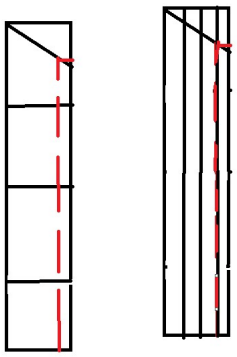
3.1 QR_factorize

Dans les grandes lignes, la fonction boucle sur nvar(colonnes), et fait des opérations (calcul de la norme) sur les vecteurs v_i en dessous de la diagonale i (par une boucle sur npoint), et elle appelle multiply_householder sur les rectangles à droite de v_i (dont la compléxité est en $O(n^2)$).

A première vue, on voulait paralléliser la grande boucle sur nvar et la répartir entre les processeurs, mais on s'est vite rendu compte que cette boucle était de plus en plus rapide. Or, si les travaux des processeurs sont déséquilibrés, le gain de temps par la parallélisation ne sera pas optimal. Et comme multiply_householder est le calcul qui prends le plus du temps dans QR_factorize, nous nous sommes dit qu'on pourrait paraléliser cette dernière et exécuter séquentiellement le reste de la fonction. (qui ne fait pas d'opération coûteuse). Ce qui permet d'équilibrer le travail des processeurs et parallélise la partie du code qui coûte le plus en temps. (On se rendra compte en codant que le travail des processeurs n'était en fait pas tout à fait équilibré puisque les opérations sur les colonnes se font qu'en dessous de la diagonale ce qui laisse les premiers processeurs inactifs). On décide donc de paraléliser multiply_householder.

3.2 Division par colonne/ligne

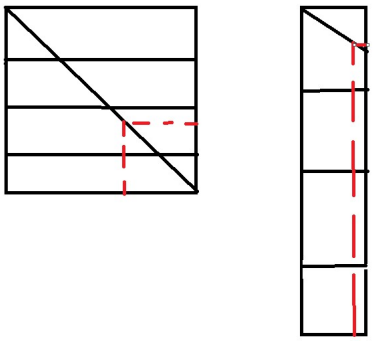
La raison pour laquelle on a divisé par ligne et non pas par colonne est illustrée par le schéma ci-dessous.



Etant donné que la matrice est réctangulaire dans le sens de la hauteur, et que dans multiply_householder on fait des opérations sur les rectangles inferieurs droites de la diagonale, on voit bien sur le schéma qu'une division par colonne déséquilibre plusle travail des processeurs.

3.3 Division par slice/cycle

Comme on l'a dit précedemment, lors des opérations sur les rectangles en bas à droite de la diagonale (dans multiply_householder), à partir d'un certain i, les premiers processeurs ne sont plus utilisés dans le cas d'une subdivision par parties. On a alors trouvé très pertinent de diviser la matrice en cycle de lignes afin de combler ce problème. Ainsi, on aurait tous les processeurs tout le temps occupés ce qui devrait grandement améliorer le temps de calcul.



Cependant, comme le montre le schéma ci-dessus cela dépend en réalité de la forme de la matrice. Nous allons donc vérifier ces hypothèses par des mesures de temps par la suite.

4 Résultats

Dans un premier temps, nous avons voulu vérifier que notre code était valide, pour cela nous avons fait quelques comparaisons avec le code séquentiel et les résultats étaient concluants.

Ensuite, on a voulu analyser le temps d'exécution de notre code, comment il évolue en fonction du nombre de processeurs/de lmax et la différence de temps entre la division par slice et celle par cycle.

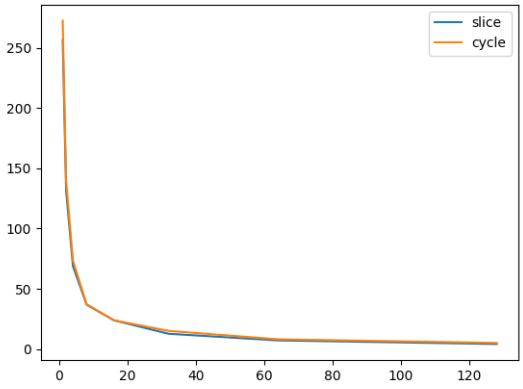


FIGURE 1 – Temps d'exécution en fonction du nombre de processeurs

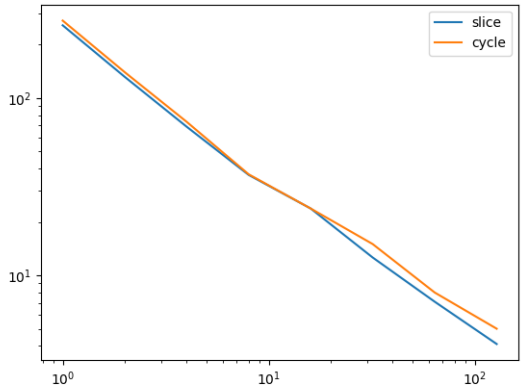


FIGURE 2 – Temps d'exécution en fonction du nombre de processeurs en échelle logarithmique

On peut remarquer ici que le temps décroit exponentiellement en fonction du nombre de processeurs, comme le montre la droite qu'on obtient en figure 3 en passant à l'échelle logarithmique. À chaque fois que l'on multiplie le nombre de processeurs par 2, le temps est divisé par 2, enfin au maximum divisé par deux. Le fait que notre courbe soit trés proche de ce résultat indique que notre paralélisation est satisfaisante.

Nous avons ensuite regardé l'évolution de l'erreur en fonction de lmax.

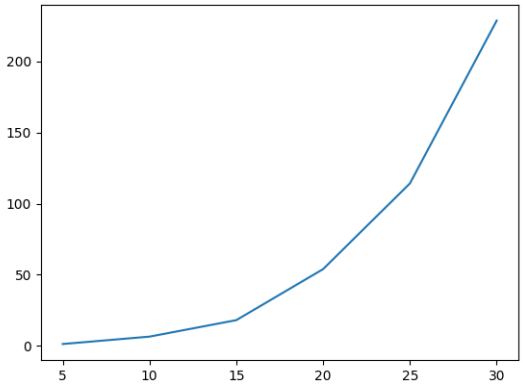


FIGURE 3 – Temps d'exécution en fonction de lmax

Le temps augmente de manière exponentielle, on peut s'y attendre étant donné que lmax est la racine du nombre

de colonne (et qu'on itère sur le nombre de colonne lors de QR-factorize). Nous n'avons pas eu l'occasion de tester notre code sur des valeurs plus élevé, mais on peut estimer grâce à ces courbe des ordre de grandeurs. Ainsi, si comme dans la figure 3, on utilise 128 processeurs, essayer de faire le calcul avec lmax = 100 prendrait environ 10 000 heures (approximation).

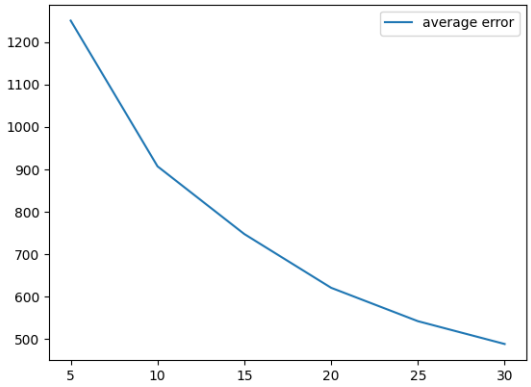


Figure 4 – L'erreur moyenne en fonction de lmax

Si on regarde l'erreur moyenne (probablement le résultat le plus intéréssant) comme le montre la figure suivante, on peut voir qu'elle décroit avec la tendance d'une exponentielle négative, ce qui signifie que chercher un bon résultat nous force a augmenter lmax exponentiellement, qui fait aussi augmenter le temps exponentiellement, et cela montre la grande nécessité de paralléliser pour pouvoir accéder à de meilleurs résultats avec des temps décents.

5 Erreur que nous avons rencontré

Après avoir décidé comment nous voulions modifier le code afin de le paralléliser, nous avons tout d'abord modifié le code afin de diviser A par parties. Pour chaque fonction, nous avons donc fait une distinction de trois cas ; si le pivot Aii appartient à la sous-matrice, si le pivot est en dessous ou si le pivot est au dessus de la sous-matrice du proceseur. Ainsi nous avons obtenu un code qui fonctionnait séquentiellement, mais nous avions une erreur lors de la parallélisation et surtout, le code était redondant et difficile à lire (voir model_par2.c).

Nous avons donc mis en place la variable "index" qui nous a permis d'avoir un pointeur sur les éléments à modifier (soit après le dernier élément lorsqu'il n'y avait rien à faire, soit sur Aii, soit le premier élément lorsque toute la sous-matrice est à modifier) Ainsi, la fonction était pus claire et il n'y avait plus besoin de faire des conditions trop longues (voir model_par3.c).

Mais il restait une petite erreur lors de la parallélisation (qui s'avérera être une erreur d'indice dans multiply_Qt), nous avons donc combiner un code parallélisé avec le code séquentiel afin de voir exactement à quel moment l'erreur se faisait (voir model_par.c). On pouvait ainsi se concentrer sur une plus petite partie du code et trouver l'erreur. Nous avons aussi utilisé cette méthode pour la seconde implémentation, dont on peut encore voir les tests dans le fichier.

Enfin, nous avons implémenté une fonction où A est divisé par cycle de lignes. Ce qui, nous pensions, permettrait d'obtenir un code plus rapide. Mais, il s'est avéré que le code par cycle était plus long à cause de la communication de data.V entre les fonctions multiply_Qt et triangular_solve qui est beaucoup plus longue par cycle que par slice car il est nécessaire de faire une boucle. Or le résultat ne change que "residual sums of squares", qui n'est pas nécessaire pour le résultat ni pour vérifier l'efficacité de l'approximation.

6 Conclusion

Pour conclure, nous pouvons dire qu'il y a de multiples manières de paralléliser ce code, et que leurs efficacité peut dépendre des paramètres choisis. Toutefois, dans ce problème, le nombre de point sera a priori beaucoup plus grand que lmax (la taille des colonnes). En effet, augmenter npoints est moins coûteux en temps qu'augmenter lmax, comme nous avons pu le montrer, donc pour avoir de bons résultats, on va commencer par prendre le plus de points possible puis augmenter lmax. Ainsi, la matrice aura tendance a être plutôt rectangulaire que carré. Donc, on peut conclure qu'une subdivision par ligne (ou colonne que nous n'avons pas faite mais qui selon nous, aurait des résultat proches) en slice est une bonne subdivision. Et, cela est confirmé par les temps que l'on a pu mesurer.