Projet 1 MPI

K. A, M. A, C. M

March 2020

Introduction

L'objet de ce travail est de présenter une parallélisation de calcul pour un problème inverse. En effet, il s'agit de calculer une série de Fourier en vue de la résolution d'un problème de Laplace mal-posé.A travers ce travail nous pourrons comparer un code séquentiel avec un code parallélisé, faire varier le paramètre α.Calculer l'erreur commise entre la solution analytique et la solution approchée et enfin montrer numériquemenet le résultat de convergence du Théorème 1.

On considère les équations données dans l'article , on commence par exprimer $f(x)$ comme suit :

$$f(x) = \frac{1}{\alpha}h(x) - \frac{2}{\alpha l}\sum_{k=1}^{m} \frac{k\pi}{\alpha l sin(\frac{bk\pi}{l}) + k\pi} \int_0^l sin(\frac{k\pi x}{l}) sin(\frac{k\pi \xi}{l}) h(\xi) d\xi$$

Ensuite, On considère la fonction $u^\alpha(x, y)$ elle est donnée par les équations suivantes :

$$a_k^\alpha = \frac{2sinh(\frac{bk\pi}{l})}{\alpha l sin(\frac{bk\pi}{l}) + k\pi} \int_0^l sin(\frac{k\pi \xi}{l}) h(\xi) d\xi$$

$$u^\alpha(x, y) = \sum_{k=1}^{\infty} \frac{a_k^\alpha sinh((b-y)\frac{k\pi}{l})}{sin(b\frac{k\pi}{l})} sinh(\frac{k\pi x}{l})$$

L'objet dans le travail effectué est de voir comment réagit le calcul de cette série On rappelle le théorème 1 du document étudié : Si la donné h(x) est bornée sur $[0, l]$ alors :
$\forall \alpha > 0 \;\; y_0 > 0$ La regularisation $u^\alpha(x, y)$ converge uniformément vers la solution exacte $\forall x \in [0, l] \;\; \forall y \in [y_0, b]$

1 Implémentation séquentielle

On commence par implémenter le problème de manière séquentielle. Cela nous permet d'étudier la compléxité de la résolution et de séparer les différentes étapes de la résolution. Pour ce faire, on commence par gérer à l'aide d'un module, les différents paramètres. De plus, cette modularité nous permet de réutiliser le code entre les différentes version du code. Ce module permet de charger les paramètres depuis un fichier à l'aide de **namelist**, Une distinction entre le cas séquentiel et l'utilisation de MPI est éffectuée de facon à réduire l'IO disque.
Un second module, permet de mettre en commun les parametres de précision et certaines constantes. Pour finir, un dernier module commun regroupe plusieurs fonctions liés à MPI et la distributions des taches.

Pour résoudre le problème on implémente chacune des fonctions nécésssaires à la résolution. Pour ce faire, nous séparons chaque étape de calcul de l'expression final en bloc implementés séparement. Cela permet de simplifier l'écriture des expressions mathématiques et par exemple comme dans le cas de l'intégrale d'une expression, d'avoir une fonction simple à appeller et non une expression plus élaborée. On a alors une vision claire de ce que chaque fonction éffectue. C'est donc le but d'un module spécifique que nous implémentons dont l'objectif est de permettre l'obtention de la fonction a_k^α.

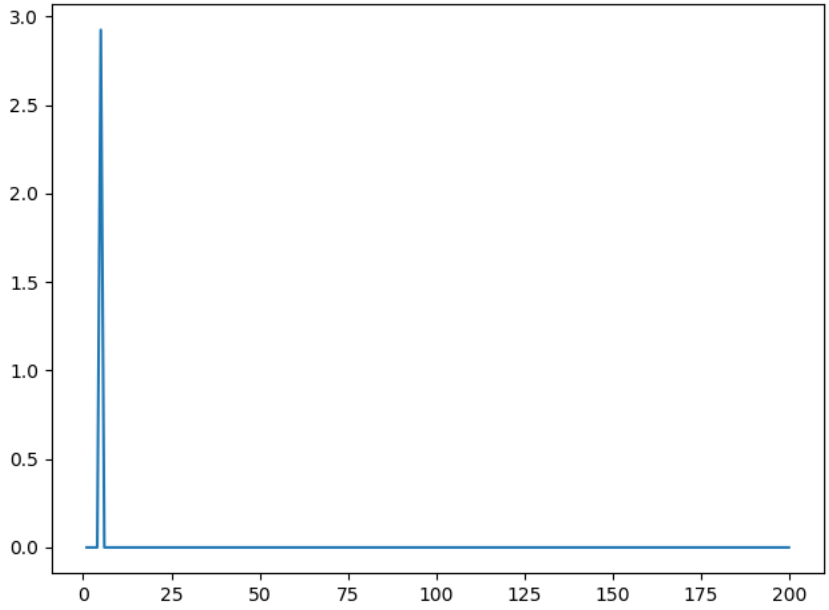
Nous pouvons maitenant implémenter la résolution du problème. La premiere étape consiste à charger les paramètres. Celle ci nous permet alors les différentes ressources nécésssaire. Nous sommes alors en mesure de calculer la solution sur le maillage définis par les paramètres donnés. Pour ce faire nous choissisons d'unifier le calcul de la solution sur la frontière et sur l'intérieur du domaine. Nous avons donc trois boucles imbriqués, deux d'espace, et une pour la série. Pour l'ordre des boucles, nous prenons en compte l'organisation en mémoire et le cout de calcul. C'est pour cela, que nous calculons la valeur de a_k^α en premier. De plus, nous sortons les expressions constantes des sous boucles, de manière à limiter leur évaluation.

2 Implémentation avec MPI

La premiere implémentation consiste à reprendre la version séquentielle et à découper le travail sur les termes de la séries entre plusieurs processeurs. Cette approche n'est pas la plus efficace mais permet de vérifier le bon fonctionnement des fonctions de répartition du travail et de l'utilisation correcte des réductions nécessaire. Une attention a été portée sur l'équivalence entre la précision pour les flottants de Fortran et MPI. Pour ce faire, une utilisation des avancées des nouvelles normes Fortran est utilisé. Cela nous assure la consistance et la bonne définition de simple et double précision. Après une étude de convergence de la série a_k^α, pour confirmer les résultats du papier, nous obtenons bien le fait que k peut être petit. L'intéret de distribuer le travail sur cet indice est alors minime. C'est pour ce fait, que nous choisissons dans une seconde version de calculer l'ensembe des a_k^α nécéssaire et de les stocker. On peut alors distribuer les calculs en fonction de chaque point du maillage. Pour ce faire, on calcule le nombre de taches qu'un processeur doit faire. Or nous avons une matrice 2D, on doit donc retrouver les positions i et j à partir du numéro de tache, c'est pour cela que nous avons introduit les différents modules commun. Pour regrouper l'ensemble des résultats nous utilisons encore une fois une réduction. Il serait possible de réduire l'empreinte mémoire du programme en allouant seulement la partie nécéssaire du tableau pour chaque processus. Le problème étant de savoir comment décaller le début d'une zone mémoire pour la variable finale de la reduction avec MPI.

3 Analyse de la convergence de la série a_k^α

L'intéret d'étudier la convergence de la série est de pouvoir justifier le choix de la borne sup de la série pour avoir une somme. De plus, cette série étant indépendante du maillage, la borne est utilisable peut importe le maillage choisi. Enfin, la convergence de la série est déterminé par la régularitée de la fonction h. En étudiant la convergence de la série, on s'assure d'une régularité suffisante.



4 Vérification de la validité de l'implémentation

Pour vérifier nos résultats, on se propose de les comparer avec ceux de l'article. On va donc étudier les solutions 1 et 5 qui sont proposés dans l'exemple 1. Pour commencer, observons le résultat que l'on obtient pour chaque.

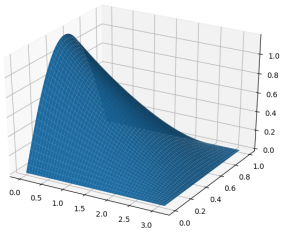


FIGURE 1 – Solution 1

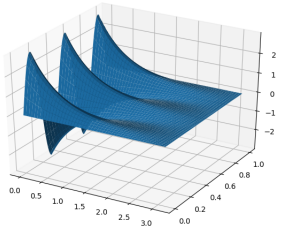


FIGURE 2 – Solution 5

D'un point de vue macro, les résultats obtenus ressemble a ceux attendus. Etudions alors le comportement de l'erreur sur le bord.

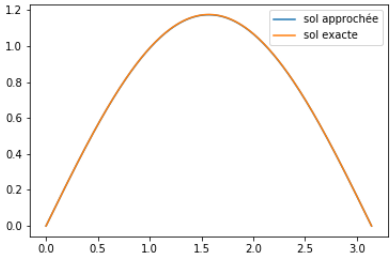


FIGURE 3 – Solution 1, $\alpha = .001$

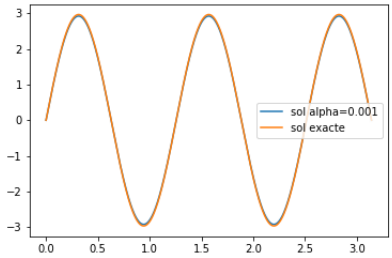


FIGURE 4 – Solution 5, $\alpha = .001$

On a bien le meme comportement sur la frontiére sous déterminée. Etudions alors l'erreur pour valider le bon fonctionnement du code.

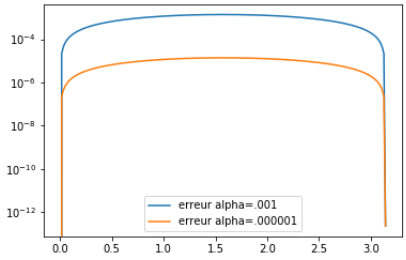


FIGURE 5 – Solution 1

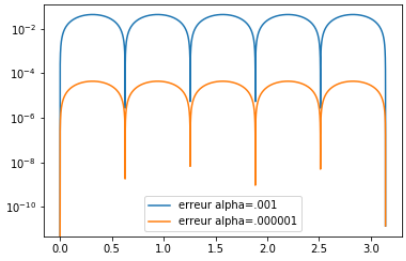


FIGURE 6 – Solution 5

Pour la premiere solution, on obtient bien le résultat attendu. La solution 5, quand à elle, est un peu différente. Cela s'explique dans les choix de la discrétisation spatiale, du nombre de k utilisés et de la qualité de l'intégration numérique.

5 Analyse des performances

La méthodologie consiste à executer plusieurs fois le meme programme avec les meme parametre pour obtenir un temps moyen. On obtient alors les temps suivant :

	Sequentiel	Par V0	Par V1
500	0,433	0,64	0,811
750	0,908	0,855	1,364
1000	1,565	1,263	2,292
2000	6,255	3,912	7,708
3000	13,933	8,952	18,021
5000	39,626	24,1	48,244

FIGURE 7 – Temps de calcul

Pour les versions parallèle, étant limité à 3 core, c'est le nombre que l'on utilise pour MPI. La première version parallèle se comporte bien au niveau des

temps. On retrouve bien le fait que si le nombre de calcul est trop petit, la compléxité supplémentaire engendré par MPI nous pénalise.

Pour la seconde version parallèle, les résultats semble catastrophique. Cependant, c'est simplement le cout de la réduction finale. Du à un problème pour savoir comment recopier un sous tableau, l'utilisation d'une réduction pour résoudre le problème en pose un nouveau.

6 Utilisation des sources

Pour la compilation, on utilise CMake qui permet de créér le makefile nécéssaire au bon déroulement de la compilation. Pour ce faire, on créé alors un répertoire *build* dans le répertoire du projet par exemple et on s'y place. Ensuite, on indique a CMake l'emplacement du fichier *CMakeLists.txt*, dans le cadre de notre exemple, *cmake ..* On obtient alors par default un makefile permettant la compilation, *make* permet alors de créer l'ensemble des executable. Pour plus d'information liés au cible du makefile, éffectué *make help*. Enfin, lors de l'éxecution de cmake, un rappel de comment executer un programme avec MPI est affiché.

7 Conclusion

En conclusion, ce projet nous a permis de mettre en perspective les avantages du calcul parallèle mais également, les difficultés sous-jacentes a une bonne implémentation. En effet, afin d'atteindre des performances optimales il a fallu proposer des stratégies efficaces de parallélisation. D'autre part la version v1 qui devait être plus rapide s'avère moins performante en temps cpu. Ceci est du notamment au fait que la gestion dynamique des tableaux (augmenter des dimensions en cours de calcul par exemple) n'est pas aisée en Fortran.Les erreurs obtenues pour les différentes méthodes restent relativement faible. On peut améliorer en choisissant une discrétisation spatiale non uniforme et également par des méthodes d'intégration plus évoluées cela aura nécessairement un impact sur le temps de calcul.