Mathématiques et Mécanique 2A

Projet Programmation CHP

Parallélisation d'un code de calcul : résolution de l'équation de la chaleur

---

Auteurs :
M. Quentin BORBEAU
M. Thibaud LOISEAU
M. Thibaud VERDIER

Enseignant :
Mme. Heloise BEAUGENDRE

27 novembre 2019

Sommaire

Chapitre 1

Introduction

L'objectif de ce projet est d'implémenter et de paralléliser un code de calcul de résolution de l'équation de la chaleur à deux dimensions. En effet afin de gagner en temps de calcul, il s'agit d'utiliser l'ensemble des processeurs d'une machine. Dans le cadre de ce projet, nous utiliserons la librairie OpenMPI qui permet notamment d'établir des communications entre les différents processeurs d'une même machine.
Une fois le code parallélisé, il s'agira de s'assurer de la performance de celui-ci et de la qualifier par des valeurs.

Chapitre 2

Annalyse mathématique

On cherche à résoudre l'équation de la chaleur dans le domaine $[0,L_x]\times[0,L_y]$:

$$\begin{cases} \partial_t u(x,y,t) - D\Delta u(x,y,t) = f(x,y,t) \\ u|\,\Gamma_0 = g(x,y,t) \\ u|\,\Gamma_1 = h(x,y,t) \end{cases}$$

f représentant le terme source, g la condition aux bords horizontaux Γ_0 du domaine, et h la condition aux bords verticaux Γ_1.

On cherche les solutions à ce problème pour trois cas différents :

— $f = 2(y - y^2 + x - x^2) \qquad g = 0 \qquad h = 0$

— $f = sin(x) + cos(y) \qquad g = sin(x) + cos(y) \qquad h = sin(x) + cos(y)$

— $f = e^{-\left(x-\frac{L_x}{2}\right)^2} e^{-\left(y-\frac{L_y}{2}\right)^2} cos\left(\frac{\pi}{2}t\right) \qquad g = 0 \qquad h = 1$

2.1 Schéma numérique

En différences finies centrées du second ordre en espace, le schéma d'Euler implicite donne :

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} - D\frac{u_{i+1,j}^n - 2u_i, j^n + u_{i-1,j}^n}{h_x^2} - D\frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h_y^2} = f_{i,j}$$

où :
$u_{i,j}^n$ est la température au noeud (i,j) et à l'itération n, et compose le vecteur solution U

Δt est le pas de temps $h_x = \frac{L_x}{N_x}$ est le pas d'espace suivant la direction x, $L_x = 1.0$ $h_y = \frac{L_y}{N_y}$ est le pas d'espace suivant la direction y, $L_y = 1.0$ $D = 1.0$ est la diffusivité thermique

2.2 Forme matricielle

Le schéma numérique peut se mettre sous la forme matricielle $AU = F$, avec :

$$A = D * dt * \begin{pmatrix}
\frac{2}{h_x^2} + \frac{2}{h_y^2} & \frac{-1}{h_x^2} & \cdots & \frac{-1}{h_y^2} & & & & & & & \\
\frac{-1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & \frac{-1}{h_x^2} & \cdots & \frac{-1}{h_y^2} & & & & & & \\
\vdots & \frac{-1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & \frac{-1}{h_x^2} & \cdots & \frac{-1}{h_y^2} & & & & & \\
 & & \ddots & \ddots & \ddots & & \ddots & & & & \\
\frac{-1}{h_y^2} & & \cdots & \frac{-1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & \frac{-1}{h_x^2} & \cdots & \frac{-1}{h_y^2} & & & \\
 & \ddots & & & \ddots & \ddots & \ddots & & \ddots & & \\
 & & & & & & & & & \frac{-1}{h_x^2} & \\
 & & & & & & \ddots & \ddots & \ddots & \vdots & \\
 & & & & & & \cdots & \frac{-1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & \frac{-1}{h_x^2} & \\
 & & & & & \frac{-1}{h_y^2} & & \cdots & \frac{-1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} &
\end{pmatrix}$$

A est une matrice pentadiagonale formée de trois diagonales centrées puis deux diagonales de part et d'autre de cette tridiagonale. Les deux diagonales solitaires sont espacées de Ny termes du bloc central. A est symétrique par construction, et définie positive.

$$U = \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ \vdots \\ u_{1,N_y} \\ u_{2,1} \\ u_{2,2} \\ \vdots \\ \vdots \\ u_{N_x,N_y} \end{pmatrix}$$

U est le vecteur solution du maillage, stocké dans celui-ci par ligne.

$$F = \begin{pmatrix} f_{1,1} + g_{1,1}h_y^2 + h_{1,1}h_x^2 \\ f_{1,2} + g_{1,2}h_y^2 \\ f_{1,3} + g_{1,3}h_y^2 \\ \vdots \\ f_{1,N_y-1} + g_{1,N_y-1}h_y^2 \\ f_{1,N_y} + g_{1,N_y}h_y^2 + h_{1,N_y}h_x^2 \\ f_{2,1} + h_{2,1}h_x^2 \\ f_{2,2} \\ \vdots \\ f_{2,N_y-1} \\ f_{2,1} + h_{2,N_y}h_x^2 \\ f_{3,1} + h_{3,1}h_x^2 \\ \vdots \\ \vdots \\ f_{N_x,N_y} + g_{N_x,N_y}h_y^2 + h_{N_x,N_y}h_x^2 \end{pmatrix}$$

F est le vecteur second membre du système matriciel et est stocké par ligne comme le vecteur U. Il comprend le terme source de chaque noeud, et les termes des conditions aux bords pour les noeuds situés aux bords.

Chapitre 3

Implémentation informatique

Notre code de calcul, aussi bien dans sa version séquentielle que parallèle, s'articule autour d'un fichier principal *main.f90*, d'un fichier de paramètres *parametres.txt*, et de différents modules :

- *matA.f90* qui permet de génèrer la matrice A (ou plutôt AA, IA et JA)
- *fonctions.f90* qui contient les fonctions f, g et h
- *second_membre.f90* qui permet de construire le second membre du système matriciel
- *sys_lin.f90* qui contient les subroutines permettant de résoudre le système

Le code séquentiel comprend également un module *mod_parallele.f90* qui contient la subroutine *charge* permettant de répartir les inconnues sur les différents processeurs. + mat A
Dans un but d'efficacité, la matrice A sera stockée en *sparse*. C'est à dire sur trois vecteurs AA, IA et JA contenant respectivement les éléments non-nulles de la matrice (parcourue par ligne), leurs indices i et leurs ordonnées j. Les algorithmes utilisés, notamment celui du gradient conjugué, seront donc ajustés en conséquence.

3.1 Code séquentiel

3.1.1 Implémentation

Il s'agit dans cette première partie d'implémenter un code séquentiel de référence, qui en plus de servir de base au code parallèle, permet d'établir les courbes de speed_up.

3.1.2 Validation

La validation du code séquentiel passe par l'appréciation des résultats et comportements de la solution. Ainsi, on doit pouvoir observer un état initial correspondant aux conditions initiales, un état intermédiaire régit par la diffusion thermique, et un état final composé du terme source et des conditions aux bords.
On valide le code pour les trois cas vus en introduction du chapitre 1.
La solution initiale est une constante égale à 2.

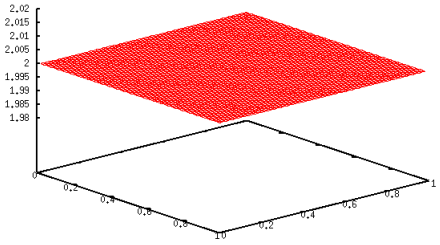

FIGURE 3.1 – Solution à $t = 0s$

Les trois figures suivantes illustrent la solution à $t = 1.0s$ pour nos trois cas respectifs.

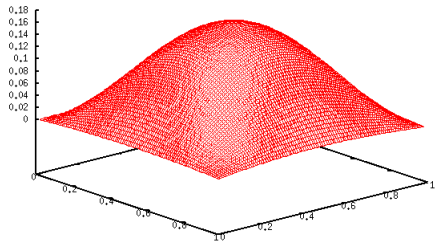

FIGURE 3.2 – Solution pour le cas stationnaire à $t = 1.0s$

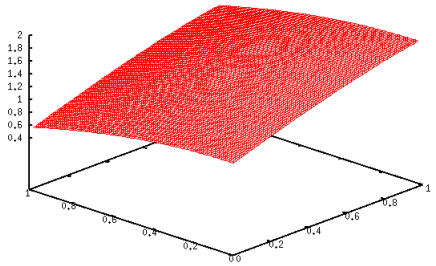

FIGURE 3.3 – Solution pour le cas périodique à $t = 1.0s$

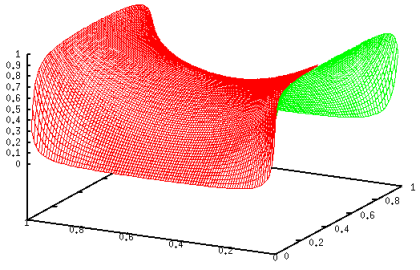

FIGURE 3.4 – Solution pour le cas instationnaire périodique à $t = 1.0s$

3.2 Code parallèle

L'implémentation du code parallèle s'est faite à partir du code séquentiel. Ainsi, les modules *matA.f90*, *fonctions.f90* et *second_membre* sont réutilisés tels quels, tandis que le fichier *main.f90* et le module *sys_lin.f90* sont parallélisés. Ils sont renommés *main_parll.f90* et *sys_lin_parll.f90*.

3.2.1 Répartition des inconnues

Nous avons choisi de répartir les inconnues de la façon suivante : nous séparons le nombre d'éléments non-nuls de la matrice A de façon équivalente sur tous les processeurs. Cependant nous ajustons ce nombre afin d'avoir un nombre entier de lignes par processeur. En effet, nous facilitons ainsi l'implémentation des produits des produits matrice-vecteur : le processeur dispose de toute les éléments de la ligne, et n'a donc pas à communiquer d'autres éléments de A avec le suivant ou le précédent pour faire le produit matrice-vecteur dans sa totalité. (Il doit cependant communiquer avec ses homologues pour avoir les parties manquantes de U.)
La subroutine *charge* (du module *mod_parallele.f90*) est la fonction qui répartit les (i1 :iN) pour chaque processeur. Elle est appelée par tous les processeurs, génère le tableau global des (i1 :iN), et chaque processeur sélectionne les valeurs qui le concerne.

3.2.2 Implémentation des communications MPI

Les communications interprocesseurs sont nécessitées à plusieurs moments dans le code parallèle, principalement dans l'algorithme du gradient conjugué :

— Lors du produit matrice-vecteur : chaque processeur a besoin de certaines valeurs de U qu'il doit demander à ses voisins.
— Lors des produits scalaires, on calcule d'abord le produit scalaire pour chaque processeurs ("pr1" par exemple), ensuite on effectue un ALLREDUCE et un BCAST

"à la main" afin de stocker le résultat global du produit scalaire ("pr1global" par exemple).

— Lors de la réunion de tous les morceaux de solution, on génère un nouveau vecteur solution global. Ce-dernier, stocké dans le processeur Np-1, reçoit les solutions de chaque processeurs.

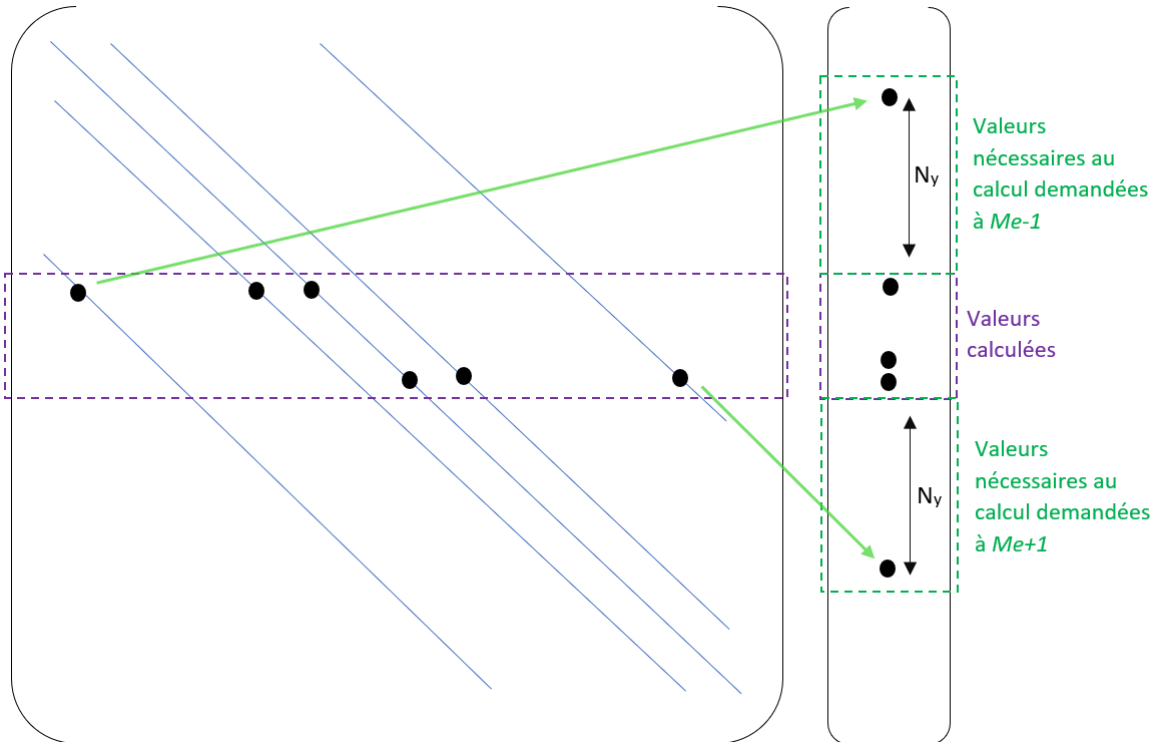

FIGURE 3.5 – Illustration des inconnues nécessaires au produit matrice-vecteur pour un processeur quelconque *Me*

Chapitre 4

Résultats et analyse

4.1 Consigne de compilation

Afin de compiler et exécuter le code parallèle, il faut taper les deux commandes suivantes :

mpif90 fonctions.f90 second_membre_sparse.f90 syslin_parll.f90 mod_parallele.f90 main_parll.f90 -o run

mpirun -n 4 –mca pnl ob1 run

4.2 Validation du code parallèle

Le critère de validation du code parallèle est la valeur de alpha dans le gradient conjugué. En effet, cette dernière est comparée pour la première itération avec la valeur obtenue dans le code séquentiel. De plus, on utilise les affichages des solutions avec le code séquentiel et le code parallèle afin de les comparer dans des configurations identiques. L'exacte similarité des résultats est bien observée.

4.3 Courbes de speed-up

Le speed-up permet de caractériser la parallélisation d'un code de calcul. Il ne peut dépasser Np, le nombre de processeurs, et est défini par :

$$Speedup = \frac{Chargetotale}{max[charge(me)]}$$

Après avoir exécuté le code séquentiel et le code parallèle pour 2, 3, 4, 5, 6 et 8 processeurs, nous pouvons tracer la courbe de speed-up ci après :

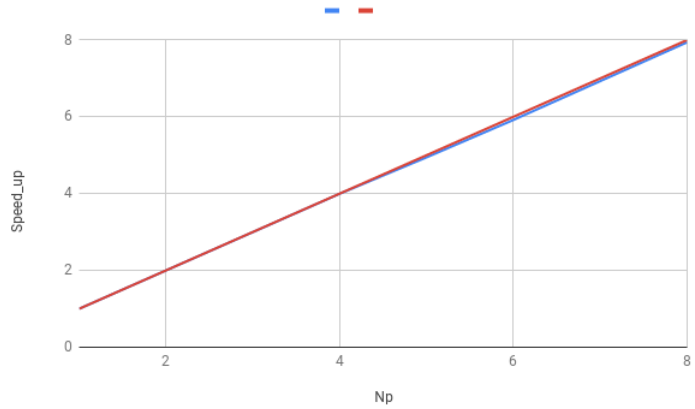

FIGURE 4.1 – Courbe de speed-up du code parallèle (en rouge la courbe idéal, en bleu la courbe réel)

Pour le speed_up, on observe bien que la courbe réelle se différencie de la courbe théorique lorsqu'on augmente le nombre de processeurs.

4.4 Courbes d'efficacité

L'efficacité permet également de caractériser la parallélisation d'un code et est définie par :

$$Speedup = \frac{Chargetotale}{max[charge(me)]}$$

Les mêmes exécutions du code que pour le speed-up nous donnent la Figure 3.2 (haut de la page suivante).

De même que pour le speed-up, on observe bien que la courbe réelle s'éloigne également de la courbe théorique. Ces résultats coïncident bien avec les résultats présentés dans le cours.

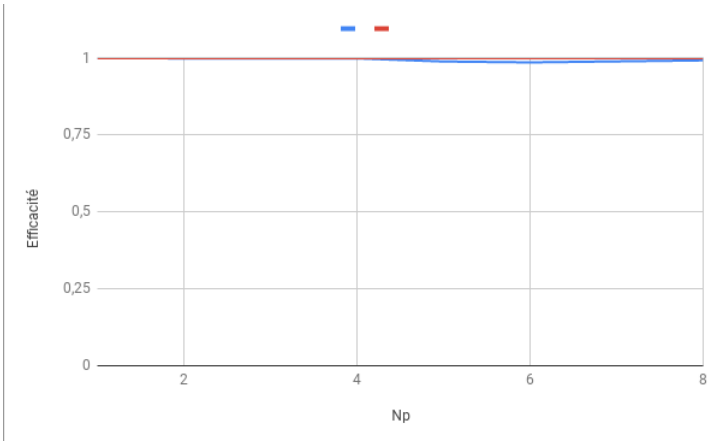

FIGURE 4.2 – Courbe d'efficacité du code parallèle (en rouge la courbe idéal, en bleu la courbe réel)

4.5 Autre courbe d'intérêt

Une autre courbe qui nous a semblé pertinente est celle du temps d'exécution en fonction du nombre de processeurs :

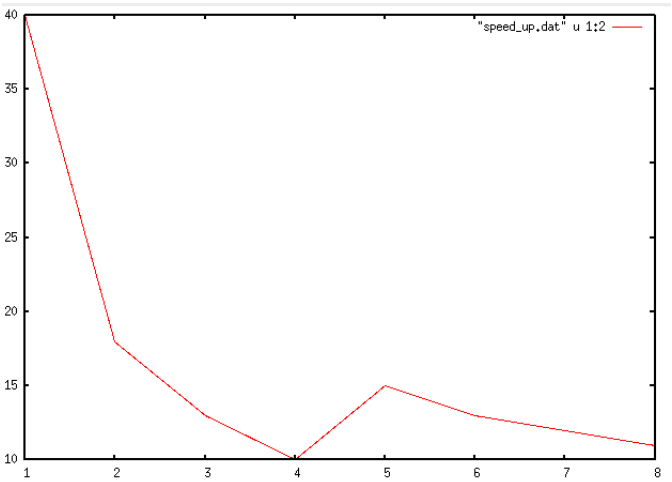

FIGURE 4.3 – Courbe d'efficacité du code parallèle (en rouge la courbe idéal, en bleu la courbe réel)

On peut y voir que le code s'exécute le plus rapidement pour $Np = 4$, soit le nombre réel de processeurs de la machine utilisée.

4.6 Difficultés rencontrées

Au cours de la parallélisation, nous avons rencontré un certains nombre de difficultés. Parmi celles-ci :

- L'impossibilité de faire fonctionner différentes fonctionnalités de la librairie OpenMPI comme *MPI_BARRIER*, *MPI_ALLREDUCE* ou *MPI_BCAST*.
- L'impossibilité de résoudre de multiples erreurs à l'exécution (qui ne semblent pas empêcher le bon fonctionnement du code toutefois) : *"21854hfi_wait_for_device : The /dev/hfi1_0 device failed to appear after 15.0 seconds : Connection timed out"*

Chapitre 5

Conclusion

Ce projet constitue une première introduction à la parallélisation d'un code de calcul. Nous avons réussi à obtenir des résultats probants, c'est à dire avec une efficacité et un speed-up proche de l'idéal. L'obtention de ce gain de temps de calcul a été au prix d'un long temps de codage. Nous voyons ainsi le grand intérêt de paralléliser un code utilisé fréquemment afin d'optimiser son temps d'exécution, au détriment d'un temps de développement supérieur.