Projet de programmation CHP

Badr CHERQAOUI et Thomas MAITRE

Table des matières

1 Introduction

On propose dans ce papier de résoudre l'équation de la chaleur numériquement, d'abord séquentiellement puis à l'aide du calcul parallèle. Nous effectuons d'abord une analyse mathématique comprenant l'écriture du schéma numérique proposé, la mise en forme matricielle du problème ainsi qu'une description détaillée de la structure de la matrice et de ses propriétés. Suite à cela, nous passons brièvement en revue le fonctionnement du code séquentiel qui sert de point de départ à la version parallèle. Ensuite, nous expliquons alors précisément le parallèlisme mis en oeuvre dans ce projet en décrivant la répartition des inconnues entre les différents processeurs et en détaillant les différentes communications réalisées. Pour finir, nous comparons les deux versions du code produit.

2 Analyse du problème

On se place dans le domaine $[0, L_x] \times [0, L_y]$ et on résout l'équation de la chaleur :

$$\begin{cases} \partial_t u(x,y,t) - D\Delta u(x,y,t) = f(x,y,t) \\ u|_{\Gamma_0} = g(x,y,t) \\ u|_{\Gamma_1} = h(x,y,t) \end{cases} \tag{1}$$

où Γ_0 et Γ_1 sont les bords du dit domaine (voir Figure 1).

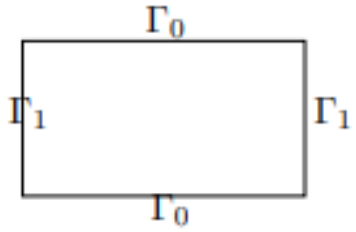

FIGURE 1 – Représentation du domaine considéré

On discrétise ensuite l'espace à la façon de la Figure 2 : soient $(i,j) \in [\![1, N_x]\!] \times [\![1, N_y]\!]$, on pose $\Delta x = \frac{L_x}{N_x+1}$ et $\Delta y = \frac{L_y}{N_y+1}$. On obtient la discrétisation $\begin{cases} x_i = i\Delta x \\ y_j = j\Delta y \end{cases}$. Le temps est également discrétisé de façon à ce que pour $n \in [0,T]$, $t_n = n\Delta t$. On approche alors $u(x_i, y_j, t_n)$ par $u^n_{i,j}$ et $f(x_i, y_j, t_n)$ par $f^n_{i,j}$.

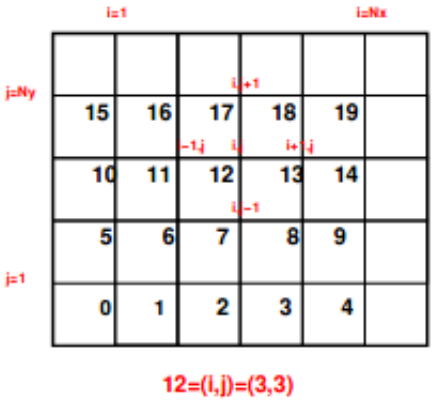

FIGURE 2 – Maillage du domaine

On peut alors écrire le schéma d'Euler Implicite à l'aide des différences finies centrées du second ordre en espace pour

l'équation (1) pour $(i,j) \in [\![1, N_x]\!] \times [\![1, N_y]\!]$:

$$\frac{U_{i,j}^{n+1} - U_{i,j}^{n}}{\Delta t} - D(\frac{U_{i+1,j}^{n+1} - 2U_{i,j}^{n+1} + U_{i-1,j}^{n+1}}{\Delta x^2} + \frac{U_{i,j+1}^{n+1} - 2U_{i,j}^{n+1} + U_{i,j-1}^{n+1}}{\Delta y^2}) = f_{i,j}^{n} \tag{2}$$

Le schéma reste valide pour $(i,j) \in \{0, N_x+1\} \times \{0, N_y+1\}$ en posant $\begin{cases} u_{0,j}^{n} = h(0, j\Delta y, t_{n+1}) \\ u_{N_x+1,j}^{n} = h(L_x, j\Delta y, t_{n+1}) \end{cases}$ pour $j \in [\![1, N_y]\!]$ et

$\begin{cases} u_{i,0}^{n} = g(i\Delta x, 0, t_{n+1}) \\ u_{i,N_y+1}^{n} = g(i\Delta x, L_y, t_{n+1}) \end{cases}$ pour $i \in [\![1, N_x]\!]$ conformément aux conditions aux limites.

Il ne reste plus qu'à mettre le problème sous forme matricielle. On réécrit le schéma (2) sous la forme :

$$-\frac{D\Delta t}{\Delta x^2}u_{i+1,j}^{n+1} - \frac{D\Delta t}{\Delta x^2}u_{i-1,j}^{n+1} - \frac{D\Delta t}{\Delta y^2}u_{i,j+1}^{n+1} - \frac{D\Delta t}{\Delta y^2}u_{i,j-1}^{n+1} + (1 + \frac{2D\Delta t}{\Delta x^2} + \frac{2D\Delta t}{\Delta y^2})u_{i,j}^{n+1} = \Delta t f_{i,j}^{n+1} + u_{i,j}^{n} \tag{3}$$

pour nous y aider. En notant $\begin{cases} \alpha = 1 + 2D\Delta t(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}) \\ \beta = -\frac{D\Delta t}{\Delta x^2} \\ \gamma = -\frac{D\Delta t}{\Delta y^2} \end{cases}$, on pose :

$$U^n = \begin{pmatrix} u_{1,1}^n \\ u_{2,1}^n \\ \vdots \\ u_{i,j}^n \\ \vdots \\ u_{N_x,N_y}^n \end{pmatrix} \text{ le vecteur d'inconnues de taille } N_xN_y,$$

$$B = \begin{pmatrix} \alpha & \beta & 0 & & & & \\ \beta & \alpha & \beta & 0 & & & \\ 0 & \beta & \alpha & \beta & 0 & & \\ & & \ddots & \ddots & \ddots & & \\ & & 0 & \beta & \alpha & \beta & 0 \\ & & & 0 & \beta & \alpha & \beta \\ & & & & 0 & \beta & \alpha \end{pmatrix} \text{ de taille } N_x \times N_x$$

$$C = \gamma \mathbf{I} \text{ de taille } N_x \times N_x$$

$$A = \begin{pmatrix} B & C & 0 & & & & \\ C & B & C & 0 & & & \\ 0 & C & B & C & 0 & & \\ & & \ddots & \ddots & \ddots & & \\ & & 0 & C & B & C & 0 \\ & & & 0 & C & B & C \\ & & & & 0 & C & B \end{pmatrix} \text{ de taille } N_xN_y \times N_xN_y$$

$$\text{et } F^{n+1} = U^n + \Delta t \begin{pmatrix} f_{1,1}^{n+1} \\ f_{2,1}^{n+1} \\ \vdots \\ f_{i,j}^{n+1} \\ \vdots \\ f_{N_x,N_y}^{n+1} \end{pmatrix} + \frac{D\Delta t}{\Delta y^2} \begin{pmatrix} u_{1,0}^n \\ \vdots \\ u_{N_x,0}^n \\ 0 \\ \vdots \\ 0 \\ u_{1,N_y+1}^n \\ \vdots \\ u_{N_x,N_y+1}^n \end{pmatrix} + \frac{D\Delta t}{\Delta x^2} \begin{pmatrix} u_{0,1}^{n+1} \\ 0 \\ \vdots \\ 0 \\ u_{N_x+1,1}^{n+1} \\ u_{0,2}^{n+1} \\ 0 \\ \vdots \\ 0 \\ u_{N_x+1,2}^{n+1} \\ \vdots \end{pmatrix}$$

On obtient alors l'équation

$$AU^{n+1} = F^{n+1} \tag{4}$$

où A est une matrice symétrique définie-positive.

On pose les fonctions tests suivantes :

-**Cas stationnaire :**

Cas 1 :

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

Cas 2 :

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

-**Cas instationnaire :**

$$f(x,y) = \exp\left(-(x - \tfrac{L_x}{2})^2\right) \exp\left(-(y - \tfrac{L_y}{2})^2\right) \cos(\tfrac{\pi}{2}t) \text{ et } g = 0 \text{ et } h = 1$$

Ces fonctions ont été choisis car nous obtenons une solution exacte pour les deux premiers cas. Cela nous permettera d'obtenir une validation de nos différents codes.

-**Solution pour le cas 1 :**

$$f_{ex}(x,y) = (x-1)(y-1)xy$$

-**Solution pour le cas 2 :**

$$f_{ex}(x,y) = \sin(x) + \cos(x)$$

Enfin, on initialise les valeurs de L_x et L_y à 1 tout au long de notre projet.

3 Mise en oeuvre du code séquentiel

Nous avons donc un système matriciel à résoudre au vu du fait que l'on a utilisé un schèma implicite en temps. Inverser la matrice A coûterait beaucoup trop cher en temps et cela n'est pas du tout recommandé. De ce fait, on va faire une analyse de la matrice afin d'utiliser un solveur numérique adapté.

On sait que la matrice est symétrique définie positive et celle ci doit être inversée à chaque pas de temps, donc la méthode du gradient conjugué est une bonne méthode à prendre pour notre problème.

3.1 Structure du Code

On va donc structurer notre code avec plusieurs fichiers, tout d'abord le main.cc où l'on va éxectuter nos fonctions principales. Puis on va créer 3 grands fichiers,

- Finit_Diff.cpp fichier permettant de construire de faire le produit de la matrice A et un vecteur

- Grad_Conjuge.cpp fichier permettant d'effectuer l'approximation de la solution du système linéaire

- Function.cpp fichier contenant les fonctions f, g, h défini précédement et des fonctions permettant d'obtenir l'erreur entre notre solution numérique et la solution exacte

Tout d'abord, on peut voir que la matrice est de taille $N_x N_y$, donc la taille de la matrice peut vite atteindre des valeurs très élevées, donc on va se servir du fait que la matrice est très creuse pour n'effectuer que les calculs nécessaires et éviter les calculs superflus.
Après avoir codé toutes les fonctions utiles, on va construire le second membre F et utiliser le gradient conjugué à chaque pas de temps. De plus, on pourra aussi calculer l'erreur entre la solution numérique et exacte, et sauvegarder la solution numérique dans un fichier à chaque pas de temps.

3.2 Validation des résultats

Tout d'abord, il a été important de valider notre fonction gradient conjugué. Celui-ci étant facile à valider. Il a suffit de calculer la norme L^2 de b - Ax du problème Ax = b. Le code nous permet de vérifier que l'erreur du gradient conjugué est bel et bien de l'ordre de la tolérance voulue. De plus, on peut obtenir des erreurs de l'ordre de 10^{-15} au maximum, ce qui représente l'erreur machine.

Ensuite, en ce qui concerne la validation de notre solution, comme nous l'avons dit précédemment, nous pouvons valider notre code avec les 2 premiers cas puisque les solutions exactes sont connues. On peut donc calculer la différence en norme 2 de la solution exacte avec la solution numérique. De même, en exécutant le code, l'erreur diminue assez rapidement et atteint aussi la tolérance voulue sans dépasser la valeur de 10^{-15}.

4 Parallélisation

Après avoir bien vérifié notre code séquentiel, et bien avoir saisi les différentes parties délicates, nous pouvons donc passer à la partie parallélisation afin de rendre notre plus rapide lors de l'exécution. Nous voyons bien que lorsque N_x et N_y sont de l'ordre de la centaine, le code séquentiel met un certain temps à faire ces calculs dépassant même les 15 à 20 secondes selon les cas. Il est alors d'usage d'optimiser notre code. On va donc utiliser la librairie MPI, on va donc répartir la charge de travail de manière convenable, et avec l'outil MPI on va permettre de faire communiquer les différents processeurs afin que chaque processeur utilise les résultats des autres.

4.1 Procédé de parallélisation

La philosophie de code est simple,

D'abord, on posera tout au long du code $s_{loc} = iEnd - iBeg + 1$, et $s_{loc} > N_x$ ce qui revient à dire que N_y supérieur aux nombres de processeurs pris en compte.

Ensuite, on va découper notre vecteur solution de taille $N_x N_y$ de manière équitable entre les processeurs grâce à la fonction charge. La fonction qui fera le produit entre A et un vecteur x sera remanié afin d'effectuer le produit matrice/vecteur d'un vecteur solution d'un processeur particulier. De même, on crée un vecteur second membre de taille s_{loc} correspondant pour chaque processeur.

Enfin, grâce à ces deux outils, on va pouvoir utiliser le gradient conjugué uniquement sur les s_{loc} lignes de la matrice A, cependant, pour un processeur donné, on doit faire le produit matrice/vecteur et donc on a besoin d'éléments qui ne font pas parti du vecteur solution de taille s_{loc} du processeur donné. De ce fait, on a besoin de communiquer entre les processeurs. Ces communications se feront donc surtout dans la fonction gradient conjugué.

Afin de sauvegarder la solution, chaque processeur va créer un fichier solution avec un indice de son rang dans lequel il sauvegarde son vecteur de taille s_{loc}, puis afficher simultanément tous les fichiers crées par tous les processeurs à l'aide du fichier "plot.txt".

4.2 Cohérence des solutions

Dans un premier temps, on peut voir la cohérence des figures que donne la solution pour le code séquentiel et parallèle. Sur les Figures 4.2, 4.2 et 4.2, on peut voir à droite la solution numérique du code parallèle sur 3 processeurs et à gauche la solution du code séquentiel. On voit donc que les solutions sont visuellement identique.

Cependant, il n'est pas suffisant de se contenter d'un résultat visuel, donc on a utilisé une fonction erreur dans notre code afin de vérifier que l'erreur est du même ordre de grandeur que le code séquentiel. On a donc calculé l'erreur en norme L^2 comme pour le code séquentiel, cependant ici on l'a calculé sur tous les processeurs puis on fini par ajouter toutes les erreurs de tous les processeurs avec la fonction MPI_Allreduce.

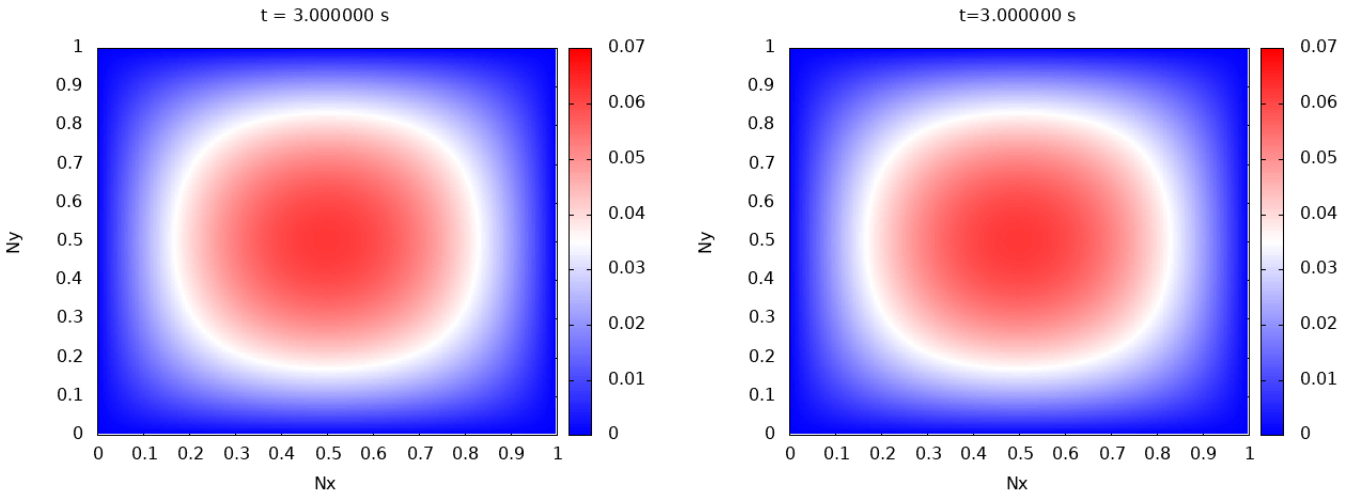

FIGURE 3 – Solution de l'équation de la chaleur pour Nx=Ny=300 et t = 1s pour le cas 1

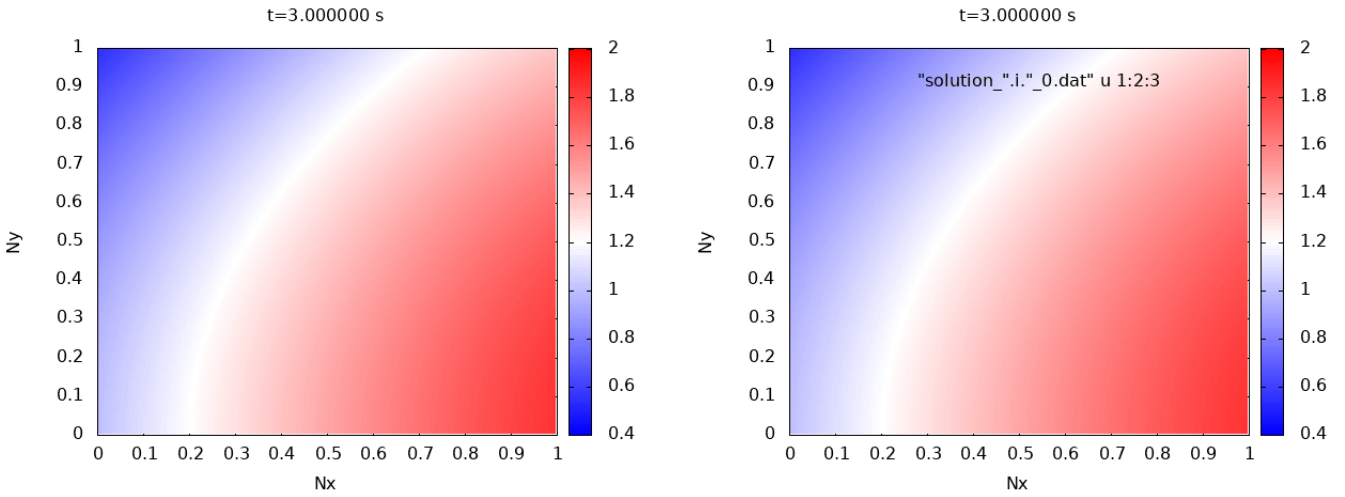

FIGURE 4 – Solution de l'équation de la chaleur pour Nx=Ny=300 et t = 1s pour le cas 2

L'erreur est finalement du même ordre que le code séquentiel, c'est à dire que la solution numérique est bien consistante.

4.3 Gradient conjugué parallèlisé

Supposons que le code soit exécuté par un processeur de rang rank quelconque, on introduit un vecteur de taille $s_{loc} + 2N_x$ dans lequel on stockera le vecteur solution depuis l'indice $iBeg - N_x + 1$ à $iBeg + N_x$. Ce vecteur sera utilisé lors du produit de la matrice A avec le vecteur solution. En effet, la matrice A étant tridiagonale par blocs avec des blocs de taille N_x, cette manipulation est nécessaire pour obtenir le produit local souhaité.

On introduit alors une fonction *loc_to_globloc* qui remplit les N_x premières composantes de ce vecteur avec les valeurs du vecteur solution de rang rank - 1, les s_{loc} d'après avec les valeurs du vecteur solution de rang rank et les N_x d'après avec les valeurs du vecteur solution de rang rank+1. On a donc besoin de faire deux communications de réception de taille N_x (et une seule si on est aux rangs 0 ou $N_p - 1$). On en profite aussi pour envoyer les infomations contenues dans le vecteur solution du rang rank aux rangs rank - 1 et rank + 1 suivant une logique similaire (deux communications d'envoi de taille N_x en règle générale, 1 seule si on est aux rangs 0 ou $N_p - 1$).

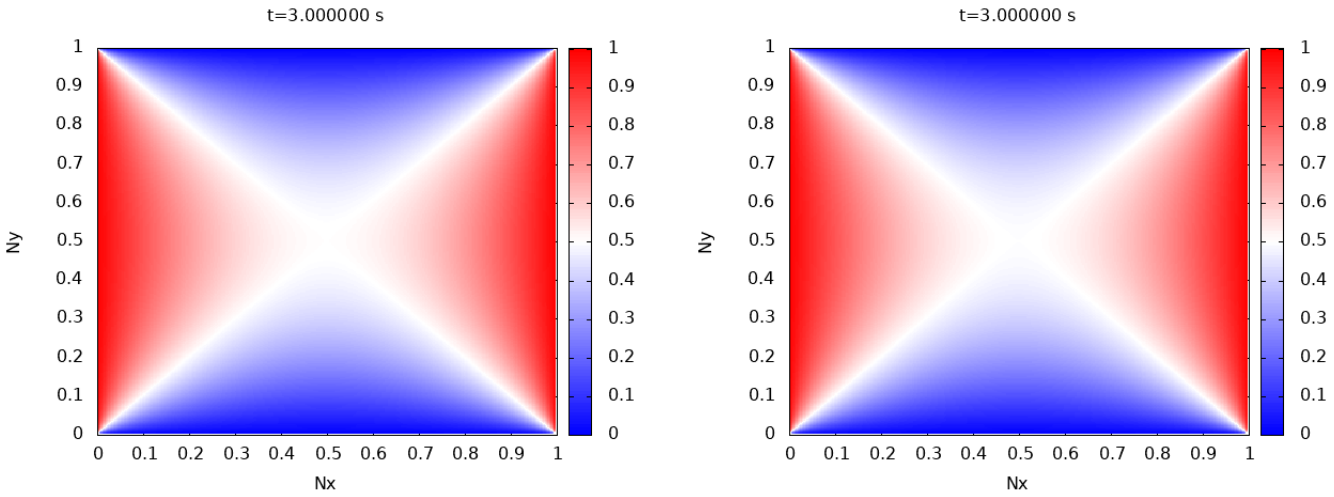

FIGURE 5 – Solution de l'équation de la chaleur pour Nx=Ny=300 et t = 1s pour le cas 3

Enfin, on utilise la fonction MPI_Allreduce à chaque fois que l'on doit calculer les produits scalaires de certains vecteurs dans l'algorithme classique du gradient conjugué car c'est bien le vecteur global qui doit être utilisé et non pas un vecteur local.

5 Courbes de Speed-Up

On définit le Speed-up S d'un programme parallèle comme

$$S = \frac{\text{charge totale}}{\max_{me}(\text{charge}(me))} \approx \frac{\text{temps séquentiel}}{\text{temps parallèle}} \tag{5}$$

où me désigne le numéro du processeur. On définit aussi l'efficacité E comme

$$E = \frac{S}{N_p} \tag{6}$$

avec N_p le nombre de processeurs utilisés.

On a donc ensuite voulu les courbes de Speed-up du code programmé pour $N_x = N_y = 300$ et $L_x = L_y = D = dt = 1$ pour les trois cas étudiés de façon à montrer l'intérêt du code parallèle sur le code séquentiel. Pour cela, on a mesuré les temps d'exécutions des programmes à l'aide de la librairie "chrono". Pour les code parallèle, une communication de réduction supplémentaire a été implémentée de façon à ce que le code ne retourne que le temps maximum qu'ont mis chaque processeurs à éxecuter sa version du programme. Pour chaque nombre de processeurs utilisé à l'éxécution du code, on a répété la mesure de temps au moins 3 fois et pris une valeur médiane afin d'avoir des résultats plus probables.

On a d'abord tracé la Figure 6 représentant la courbe de Speed-up dans le cas numéro 1 en fonction du nombre de processeurs utilisés jusqu'à 50 processeurs. Sur cette Figure, on a observé une nette coupure après l'abscisse $n_p = 12$, ce qui nous a permis de réaliser que l'ordinateur sur lequel les mesures temporelles ont été réalisées ne possède que 12 processeurs (information vérifiée après coup dans les dossiers systèmes de la machine). On s'est donc restreint pour la suite de nos mesures (Figures 7 et 8) à 12 processeurs, mais il nous a paru intéressant de remarquer que le code parallèle fonctionnait même si le nombre de processeurs demandé était supérieur au nombre de processeurs de la machine.

Ensuite, on a tracé les Figures 7 et 8 dans les trois cas étudiés tout au long de ce projet.

Sur la Figure 7, on a aussi tracé la droite d'équation $y = x$ (courbe de Speed-up parfaite). Les résultats observés pour les premiers processeurs sont incohérents puisque les courbes de Speed-up obtenues pour ces trois cas se situent au dessus de celle

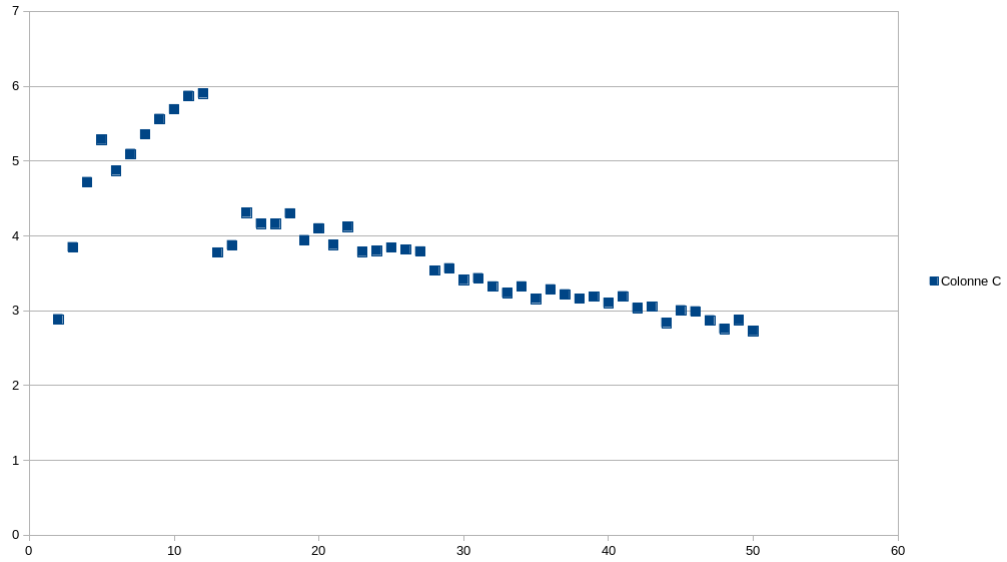

FIGURE 6 – Courbes de Speed-up en fonction du nombre de processeurs jusqu'à $np = 50$

du Speed-up parfait. Nous n'avons pas réussi à localiser l'origine de ce problème malgré le temps passé dessus. Cependant on peut quand même voir que les courbes de Speed-up ont tendance à s'éloigner de la première bissectrice. Ceci peut-être expliqué du fait des communications entre les processeurs qui rallongent le temps de calcul du code parallèle.

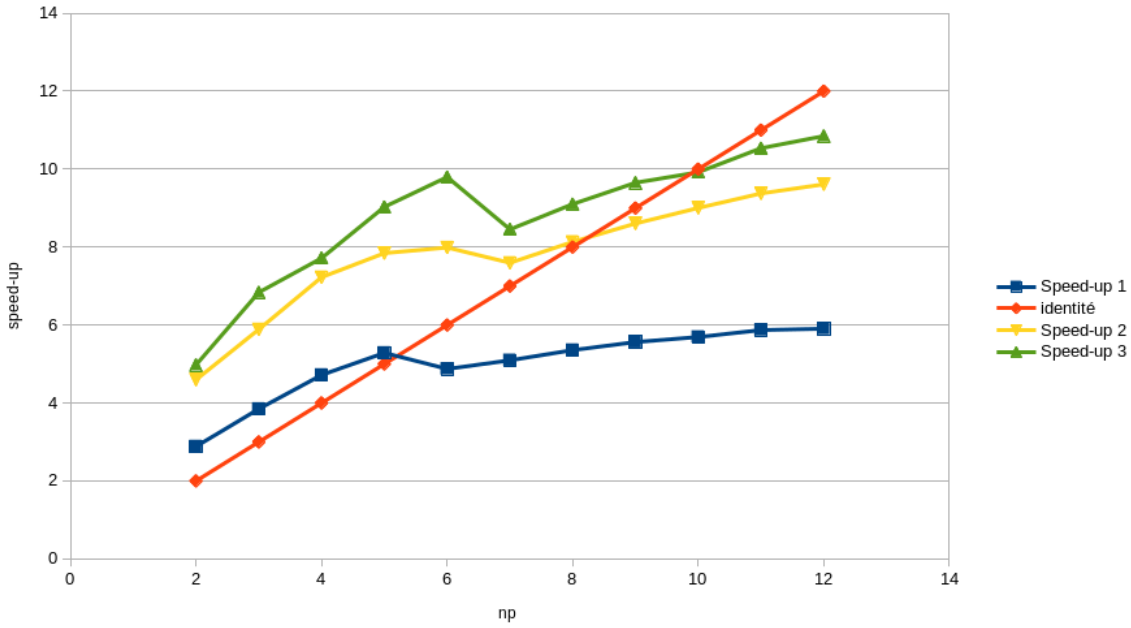

FIGURE 7 – Courbes de Speed-up en fonction du nombre de processeurs utilisés

Le problème observé sur les courbes de Speed-up apparait aussi sur les courbes d'efficacité tracées en Figure 8. Théoriquement, l'efficacité devrait être inférieur à 1 et diminuer quand le nombre de processeurs augmente. On observe bien cette diminution sur notre Figure.

De manière générale, on observe une réduction du temps de calcul importante lors du passage du code séquentiel au code parallèle.

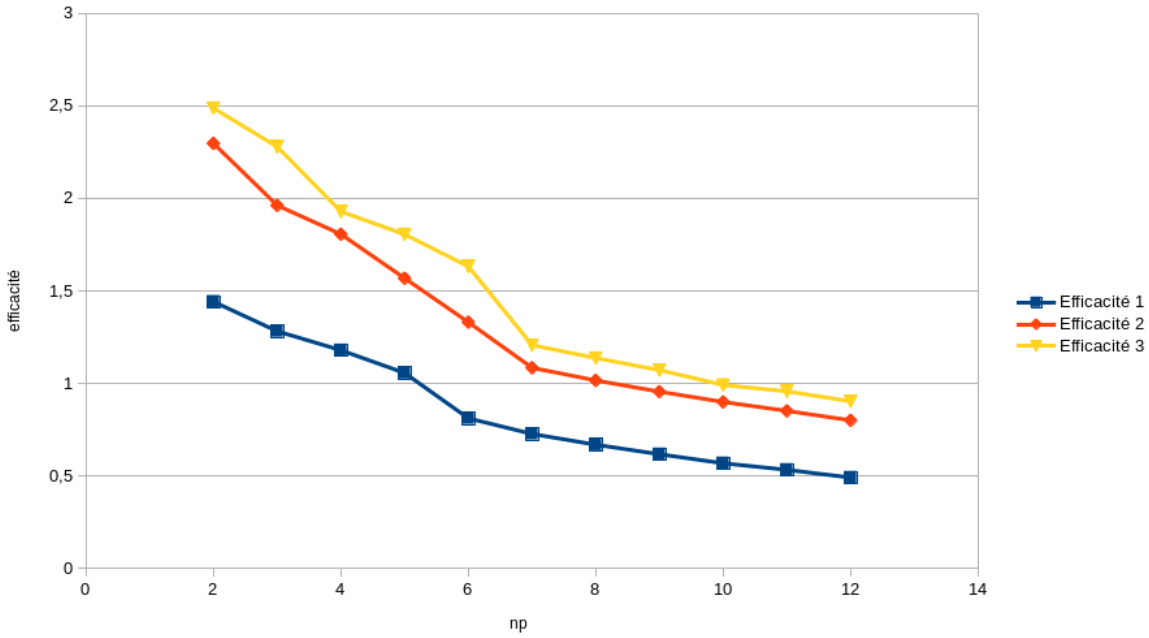

FIGURE 8 – Courbes d'efficacité en fonction du nombre de processeurs utilisés

6 Conclusion

Pour conclure, on est parti d'un code séquentiel ce qui nous a permis dans un premier temps de vérifier nos résultats sur lesquels on peut s'appuyer de manière fiable, mais aussi de comprendre le problème et de s'habituer aux différents points délicats. Puis, nous sommes passés au sujet qui nous intéresse dans ce projet, c'est-à-dire celui de paralléliser notre code. On remarque qu'une parallèlisation du code est nécessaire puisque lorsque l'on prend N_x et N_y de l'ordre de la centaine, le code a un temps d'éxécution de plusieurs dizaines de secondes.

Lors de la parallèlisation, on obtient alors un rapport de temps bien plus intéressant. Le temps de calcul est beaucoup moins grand et permet donc d'affiner nos résultats numériques.