Ecole Nationale Supérieure des Techniques Avancées

RAPPORT DE PROJET

AMS301 - Projet 1 : Problème avec grille structurée

Nour El Haddad et Adrien Sardi

Octobre 2023

Table des matières

1 Introduction

Dans la première partie de notre projet, nous visons à résoudre numériquement un problème en utilisant d'abord la méthode de Jacobi. Nous débutons par la création d'un code séquentiel, suivi de sa parallélisation pour évaluer ses performances sur une station de travail de l'ENSTA. Parallèlement, nous développons également un code séquentiel pour la méthode de Gauss-Seidel. L'objectif est de comparer les performances de ces deux méthodes après leur validation. Cette phase initiale est cruciale pour établir une base solide pour la résolution ultérieure du problème et l'optimisation des méthodes numériques.

L'ensemble des codes de notre projet sont disponible sur le lien suivant : `https://github.com/Aupeka/Calcul-scientifique-en-parallele-AMS301-Projet-1`

1.1 Problème Continu

Étant donné un domaine Ω défini comme l'intervalle ouvert $]0, a[\times]0, b[$, nous cherchons à déterminer le champ $u(x, y)$ soumis à l'équation suivante :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \quad \text{pour } (x, y) \in \Omega, \tag{1}$$

Tout en prenant en compte les conditions limites suivantes :

$$u(x, 0) = U_0, \quad u(x, b) = U_0, \quad u(a, y) = U_0, \quad u(0, y) = U_0(1 + \alpha V(y)), \tag{2}$$

avec $x \in]0, a[$ et $y \in]0, b[$, $V(y) = 1 - \cos\left(\frac{2\pi y}{b}\right)$ et $f(x, y) = 0$, On a a, b, et $\alpha > 0$, avec $\alpha < 1$.

1.2 Problème Discrétisé

Pour aborder ce problème dans sa forme discrétisée, nous définissons les valeurs du champ comme étant $u_{i,j} = u(i\Delta x, j\Delta y)$, où i varie de 0 à $Nx + 1$ et j varie de 0 à $Ny + 1$. On veut résoudre l'équation discrétisée suivante :

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f_{i,j}, \quad \text{pour } i = 1, \ldots, Nx \text{ et } j = 1, \ldots, Ny. \tag{3}$$

Notons ici que $\Delta_x = \frac{a}{N_x+1}$ et $\Delta_y = \frac{b}{N_y+1}$.

En mettant notre problème sous forme matricielle, on obtient :

$$Au = f$$

Avec

$$A = \begin{bmatrix} -2\left(\frac{1}{\Delta_x^2}+\frac{1}{\Delta_y^2}\right) & \frac{1}{\Delta_y^2} & \frac{1}{\Delta_x^2} & & \\ \frac{1}{\Delta_y^2} & -2\left(\frac{1}{\Delta_x^2}+\frac{1}{\Delta_y^2}\right) & \frac{1}{\Delta_y^2} & & \\ \frac{1}{\Delta_x^2} & \frac{1}{\Delta_y^2} & -2\left(\frac{1}{\Delta_x^2}+\frac{1}{\Delta_x^2}\right) & \ddots & \frac{1}{\Delta_x^2} \\ & \ddots & \ddots & \ddots & \frac{1}{\Delta_y^2} \\ & & \frac{1}{\Delta_x^2} & \frac{1}{\Delta_y^2} & -2\left(\frac{1}{\Delta_x^2}+\frac{1}{\Delta_y^2}\right) \end{bmatrix}$$

et

$$u = \begin{bmatrix} u_{0,0} \\ u_{0,1} \\ \vdots \\ u_{i,j} \\ \vdots \\ u_{N_x+1,N_y} \\ u_{N_x+1,N_y+1} \end{bmatrix}, \qquad f = \begin{bmatrix} f_{0,0} \\ f_{0,1} \\ \vdots \\ f_{i,j} \\ \vdots \\ f_{N_x+1,N_y} \\ f_{N_x+1,N_y+1} \end{bmatrix}$$

Dans ce cas, A est à diagonale dominante (i.e $|a_{ii}| > \sum_{j\neq i} |a_{ij}|, \forall i$). Alors, l'algorithme séquentiel de Jacobi est sensé converger.

2 Méthode Jacobi

2.1 Méthode Jacobi : Code séquentiel

Cette méthode est une approche itérative pour résoudre des systèmes linéaires de la forme $Au = f$. Cette méthode implique les étapes suivantes : on exprime A comme la somme de trois matrices D, $-L$, et $-U$, où D est une matrice diagonale, $-L$ est une matrice triangulaire inférieure et $-U$ est une matrice triangulaire supérieure.

Cela permet de résoudre le système itérativement en mettant à jour la valeur de u jusqu'à convergence vers la solution souhaitée.

Plus concrètement, cette méthode consiste à poser $A = M - N$ avec $M = D$ et $N = -(L+U)$. Notre système devient donc

$$Mu^{(\ell+1)} = Nu^{(\ell)} + f.$$

Dans notre cas $M = Id$, elle est bien diagonale et inversible.

On ne pourra pas itérer un nombre infini de fois. Pour cela, on utilise un critère d'arrêt sur le nombre d'itérations ℓ_{stop} et sur la norme du résidu $r^{(\ell)} := f - Au^{(\ell)}$:

$$\ell_{stop} \leq L \text{ et } \frac{||r^{(\ell)}||}{||r^{(0)}||} \leq \varepsilon$$

$u^{(0)} \in \mathbb{R}^{N_{x+2}N_{y+2}}$;
while $\ell = 0, 1, \ldots, L$ ***and*** $||r^{(\ell)}|| \leq ||r^{(0)}||\epsilon$ **do**
  $Du^{(\ell+1)} = f - (L+U)u^{(\ell)}$;
end

On obtient le calcul suivant pour le résidu au temps ℓ :

$$\begin{aligned} r^{(\ell)} &= f - Au^{(\ell)} \\ &= f - (M-N)u^{(\ell)} \\ &= f - Mu^{(\ell)} + Nu^{(\ell)} \\ &= -Mu^{(\ell)} + \underbrace{(b + Nu^{(\ell)})}_{\text{Mu}^{(\ell+1)}} \\ &= \underbrace{M}_{=1 \text{ Jacobi}} (u^{(\ell+1)} - u^{(\ell)}). \end{aligned}$$

Dans notre code séquentiel cela se retraduit par :

```
// Residu
  for (int i = 0; i < (Nx+2)*(Ny+2); i++){
    res[i] = sol[i] - solNew[i];
  }
```

On peut réecrire (3) de la façon suivante :

$$u_{i,j}^{(\ell+1)} = \frac{\Delta_x^2 \Delta_y^2}{2(\Delta_x^2 + \Delta_y^2)} \left[\frac{1}{\Delta_x^2}(u_{i+1,j}^{(\ell)} + u_{i-1,j}^{(\ell)}) + \frac{1}{\Delta_y^2}(u_{i,j+1}^{(\ell)} + u_{i,j-1}^{(\ell)}) - f_{i,j} \right]$$

En ce qui concerne la boucle temporelle, on a pu l'implémenter de la façon suivante :

```
// Spatial loop
  for (int i=1; i<=Nx; i++){
      for (int j=1; j<=Ny; j++){
          solNew[i+(Nx+2)*j + 1] = coeff * ((sol[i+1+(Nx+2)*j+1]+sol[i-1+(Nx+2)*j +
          ↪  1])/(dx*dx) + (sol[i+(Nx+2)*(j+1) + 1]+sol[i+(Nx+2)*(j-1) + 1])/(dy*dy)
          ↪  - f[i + (Nx+2)*j + 1]);
      }
  }
```

2.1.1 Validation de la convergence du code Jacobi séquentiel

Nous validons notre code en vérifiant la convergence numérique du schéma de différences finies et cela en utilisant une solution manufacturée. Nous choisissons :

$u_{th}(x, y) = sin(2\pi x)sin(2\pi y)$ ce qui revient à dire que $\Delta u_{th}(x, y) = f(x, y) = -8\pi sin(2\pi x)sin(2\pi y)$.

On fixe la valeur de U_0 à 0, on a les conditions limites en (2) qui sont vérifiées par u_{th}.

On affiche sur `Matlab` la solution u obtenue qui est conforme à la solution u_{th} :

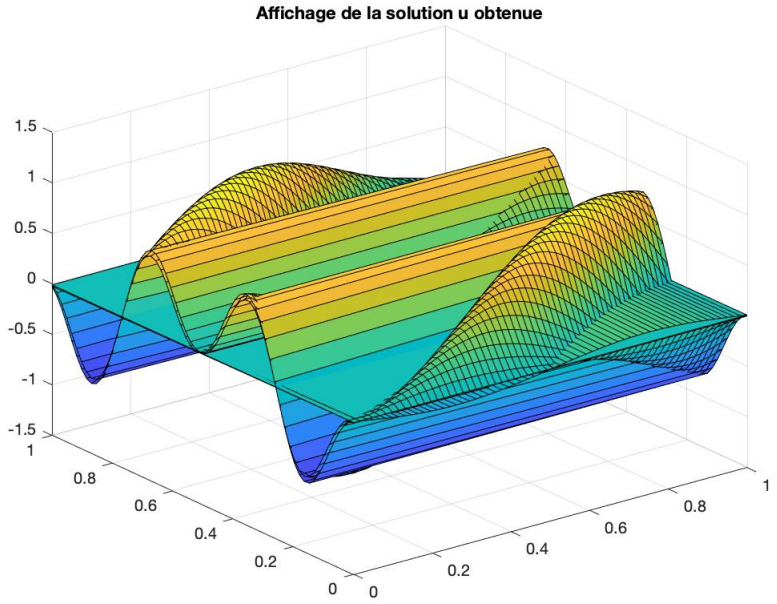

FIGURE 1 – La solution u obtenue grâce au code séquentiel

Pour vérifier la convergence de ce modèle, on s'intéresse à la norme L^2 de l'erreur entre u_{th} et notre solution numérique u.

En prenant les valeurs numériques $L = 10^6$, $\varepsilon = 10^{-6}$, $a = b = 1$, et $\alpha = 0.5$, et on obtient :

ℓ_{stop}	3221
$\|\|u_{th} - u\|\|_{L^2}$	0.0288837
$\frac{\|\|r^{(\ell)}\|\|}{\|\|r^{(0)}\|\|}$	9.99587×10^{-7}

TABLE 1 – Valeurs en sortie du code séquentiel Jacobi pour $N_x = N_y = 60$

On affiche l'erreur en fonction de l'inverse du pas du maillage $h = \frac{1}{\sqrt{N_x N_y}}$.

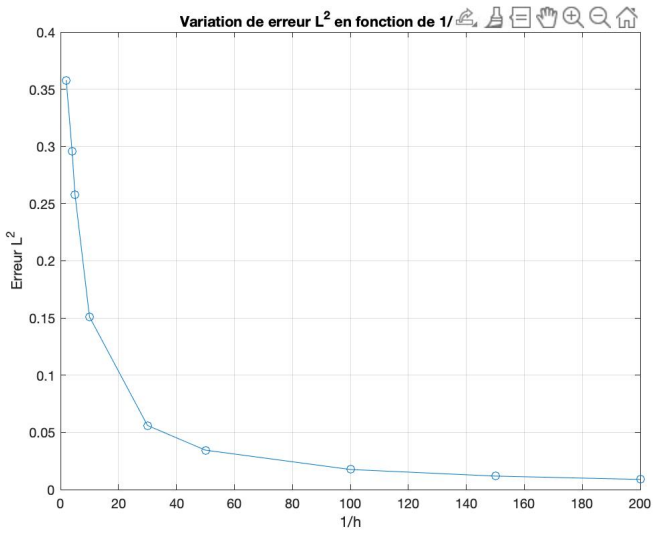

FIGURE 2 – L'erreur L^2 en fonction de $\frac{1}{h}$

Ici nos valeurs sont $L = 10^6$, $\varepsilon = 10^{-5}$, $a = b = 1$, et $\alpha = 0.5$, Comme nous le montre la figure (2), plus le maillage est fin, (i.e plus $\frac{1}{h}$ est grand) et plus l'erreur diminue rapidement et finit par rejoindre une sorte palier, ce qui est logique et accentue la convergence de notre modèle de Jacobi séquentiel.

2.2 Méthode Jacobi : code parallèle

2.2.1 Méthode de parallélisation

Maintenant que notre code est fonctionnel il peut être intéressant de le paralléliser afin d'augmenter sa vitesse d'exécution. Pour ce faire, nous allons nous concentrer sur la partie principale du code qui est la plus pertinente de paralléliser : **l'intérieur de la boucle temporelle**. En effet, le facteur déterminant pour que notre algorithme converge est le nombre de boucle temporelle (ie la valeur finale de l). La boucle temporelle étant, comme son nom l'indique, **temporelle**, i.e. elle est non parallélisable. En revanche, les deux boucles (celle en i et en j) le sont. Nous faisons le choix de paralléliser la première boucle (celle en i).

Pour ce faire, nous initialisons les élements de parallélisation dans le code, comme il est classique de faire. Puis nous définissons pour chaque i des élements : i_{start} et i_{end} qui valent respectivement :

$$i_{start} = \text{myRank} * \frac{N_x + 2}{\text{nbTasks}},$$

et

$$i_{end} = \min\left((\text{myRank}+1) * \frac{N_x+2}{\text{nbTasks}}, N_x+2\right).$$

Dès lors, notre boucle temporelle en i devient :

```
1     // Spatial loop
2     for (int i=1; i<=Nx; i++){
3         for (int j=1; j<=Ny; j++){
4             solNew[i+(Nx+2)*j + 1] = coeff * ((sol[i+1+(Nx+2)*j+1]+sol[i-1+(Nx+2)*j +
                  ↪  1])/(dx*dx) + (sol[i+(Nx+2)*(j+1) + 1]+sol[i+(Nx+2)*(j-1) + 1])/(dy*dy)
                  ↪  - f[i + (Nx+2)*j + 1]);
5             n_res_0 += (solNew[i+(Nx+2)*j + 1] - sol[i+(Nx+2)*j +
                  ↪  1])*(solNew[i+(Nx+2)*j + 1] - sol[i+(Nx+2)*j + 1]);
6         }
7     }
```

Pour que cela fonctionne il nous faut, évidemment, faire communiquer les « parties » du vecteur `sol` entre elles. c'est ce que nous faisons à l'aide des fonctions `MPI_Isend` et `MPI_Irecv` (voir code pour plus de détails).

Finalement, on peut noter qu'un nouvel élément est venu s'ajouter dans notre boucle spatiale. En effet, par mesure de simplicité nous avons décidé de calculer le résidu directement en norme à cette étape. Cela nous permet en chaque sorti de boucle spatiale d'avoir les valeurs des résidus issus des parties du vecteur `solNew` calculé. On assemble, ou plutôt on somme la valeur des résidus de chaque processus grâce à la fonction : `MPI_Reduce` en précisant `MPI_SUM` pour l'opération.

2.2.2 Validation du code parallèle

Pour vérifier la convergence de ce modèle, on s'intéresse à la norme L^2 de l'erreur entre u_{th} et notre solution numérique u, exactement comme dans le cas séquentiel.

En prenant les valeurs numériques $L = 10^4$, $\varepsilon = 10^{-4}$, $a = b = 1$, $\alpha = 0.5$, et en fixant le nombre de processus à 2 on obtient :

ℓ_{stop}	231
$\|\|u_{th} - u\|\|_{L^2}$	0.375574
$\frac{\|\|r^{(\ell)}\|\|}{\|\|r^{(0)}\|\|}$	9.4655×10^{-5}

TABLE 2 – Valeurs en sortie du code parallélisé Jacobi pour $N_x = N_y = 30$

2.3 Jacobi parallèle : le cluster Cholesky

2.3.1 Validation de la performance avec Cholesky

Maintenant que notre code parrallèle pour la méthode de Jacobi est fonctionnel nous pouvons tester ses performance sur une machine plus puissante : le cluster de Cholesky (Ecole Polytechnique). L'idée ici est de faire tourner notre code pour plusieurs valeurs de processus à $Nx = Ny = 30$ fixés. Nous obtenons les courbes suivantes :

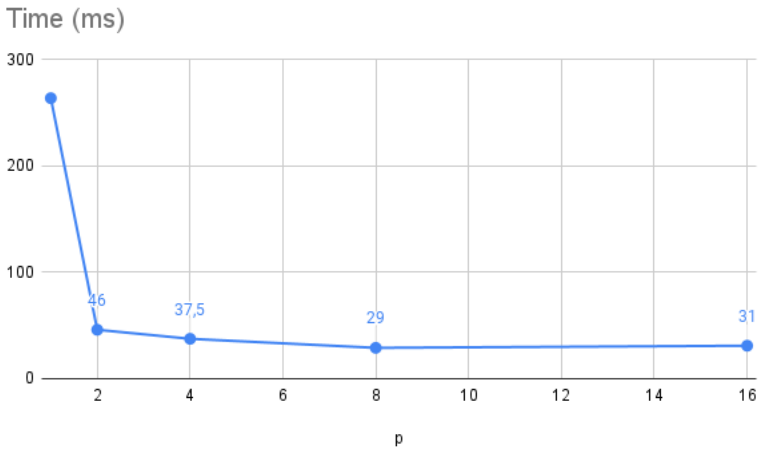

FIGURE 3 – Temps d'exécution de Jacobi parallèle avec Cholesky en fonction du nombre de processus

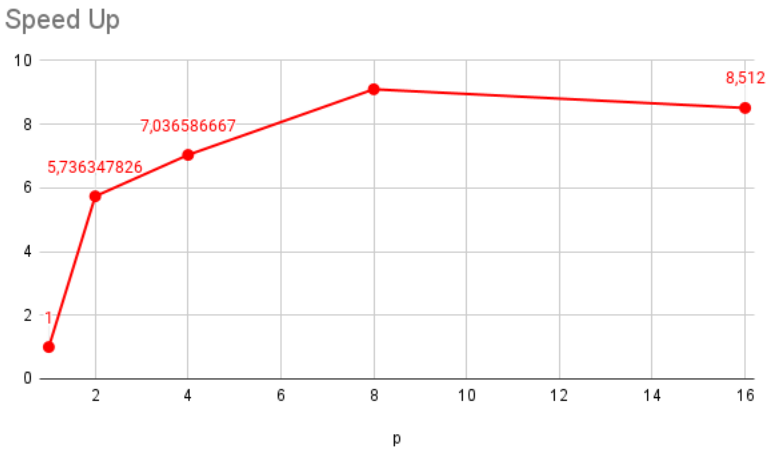

FIGURE 4 – Analyse de la scalabilité forte

Analysons les résultats de notre implémentation parallèle du code Jacobi en utilisant le cluster Cholesky. Pour commencer, étudions la variation du temps de calcul de notre algorithme qui est un bon critère pour juger l'efficacité d'une parallélisation. On voit rapidement que découper les calculs sur 2 puis 4 processus permet au code d'être executé plus rapidement. Sans surprise le calcul du Speed Up nous montre également que l'amélioration est croissante suivant une droite.

Une fois que le nombre de processus dépasse le seuil de 4 on voit que l'accelération est moins prononcée, voir le code est même plus long à s'executer que pour un nombre de processus plus faible. Cela peut s'expliquer par l'augmentation du nombre de communications et donc au bout d'un certain nombre de processus le gain de temps devient plus faible.

Finalement, on peut remarquer que malgrès tout l'efficacité de notre code n'est pas abérante. Modulo un écart pour $p = 2$ la courbe est globalement constante puis elle se met à décroitre. Ce qu'on s'attend à obtenir en pratique.

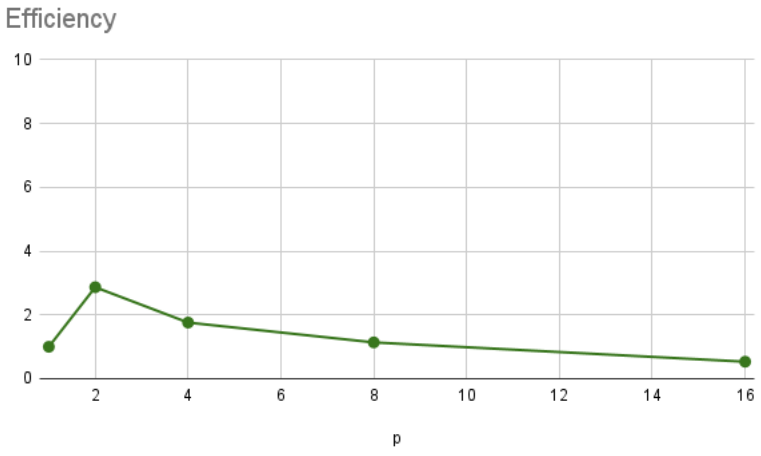

FIGURE 5 – Analyse de la scalabilité faible

3 Méthode Gauss-Siedel

3.1 Méthode Gauss-Siedel : Code séquentiel

La différence ici c'est que l'on a $Mu^{(\ell+1)} = Nu^{(\ell)} + f$ où $M = D + L$ et donc $N = -U$ Et donc on veut implémenter

En réecrivant (3), on obtient :

$u^{(0)} \in \mathbb{R}^{N_{x+2}N_{y+2}}$;
while $\ell = 0, 1, \ldots, L$ ***and*** $||r^{(\ell)}|| \leq ||r^{(0)}||\epsilon$ **do**
  $(D + L)u^{(\ell+1)} = f - Uu^{(\ell)}$;
end

$$u_{i,j}^{(\ell+1)} = \frac{\Delta_x^2 \Delta_y^2}{2(\Delta_x^2 + \Delta_y^2)} \left[\frac{1}{\Delta_x^2}(u_{i+1,j}^{(\ell)} + u_{i-1,j}^{(\ell+1)}) + \frac{1}{\Delta_y^2}(u_{i,j+1}^{(\ell)} + u_{i,j-1}^{(\ell+1)}) - f_{i,j} \right]$$

3.1.1 Validation de la convergence du code Gauss-Siedel séquentiel

Pour vérifier la convergence de ce modèle, on calculela norme L^2 de l'erreur entre u_{th} et notre solution numérique u.

En prenant les valeurs numériques $L = 10^6$, $\varepsilon = 10^{-6}$, $a = b = 1$, et $\alpha = 0.5$, et on obtient :

ℓ_{stop}	2546
$\|\|u_{th} - u\|\|_{L^2}$	0.028884
$\frac{\|\|r^{(\ell)}\|\|}{\|\|r^{(0)}\|\|}$	9.97732×10^{-7}

TABLE 3 – Valeurs en sortie du code séquentiel Gauss-Siedel pour $N_x = N_y = 60$

On affiche l'erreur en fonction de l'inverse du pas du maillage $h = \frac{1}{\sqrt{N_x N_y}}$.

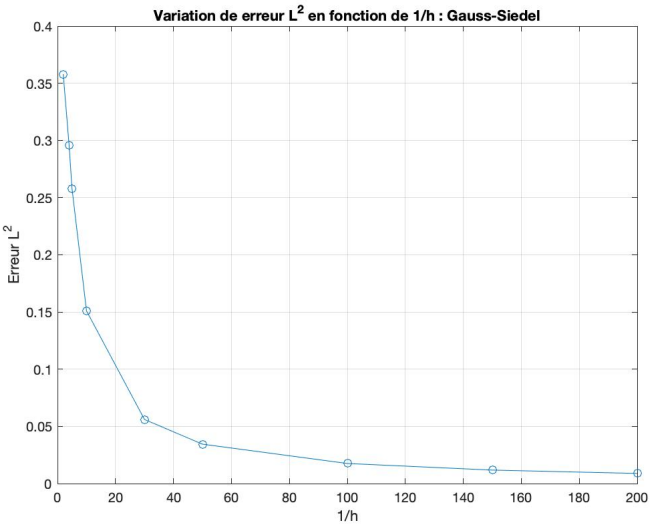

FIGURE 6 – L'erreur L^2 en fonction de $\frac{1}{h}$

Ici nos valeurs sont $L = 10^6$, $\varepsilon = 10^{-5}$, $a = b = 1$, et $\alpha = 0.5$, Comme nous le montre la figure (6) et comme dans le cas de Jacobi séquentiel, plus le maillage est fin et plus l'erreur diminue rapidement et finit par se stabiliser, ce qui est logique et accentue la convergence de notre modèle de Gauss-Siedel séquentiel.

3.1.2 Comparaison entre Jacobi et Gauss-Siedel séquentiel

On a comparé dans ce tableau la norme L^2 de l'erreur $||u - u_{th}||_{L^2}$ et le temps d'exécution en millisecondes pour chacun de nos 2 codes séquentiels pour $\varepsilon = 10^{-5}$ et $L = 10^6$:

	h	0.5	0.2	0.1	0.033	0.01	0.0066	0.005
Erreur L^2	*Jacobi*	0.357601	0.150989	0.056047	0.0344435	0.0175472	0.0117729	0.00885808
	Gauss-Siedel	0.357601	0.150992	0.05605	0.0344469	0.0175487	0.0117732	0.00885786
Temps (ms)	*Jacobi*	0.29	13	263	1578	21343	104159	328687
	Gauss-Siedel	0.14	15.63	212	1179	12083	56084	172705

TABLE 4 – Tableau des valeurs

On remarque que le code Gauss-Siedel séquentiel est toujours plus rapide que le Jacobi séquentiel (environ 2 fois plus rapide) et que les erreurs ne bougent quasi-pas, elles restent du même ordre. Nous déduisons que l'algorithme de Gauss-Siedel est plus performant.

3.2 Méthode Gauss-Siedel : code parallèle

3.2.1 Méthode de parallélisation

Maintenant que notre code séquentiel est fonctionnel nous allons pouvoir le paralléliser. L'idée est la même que pour le problème de Jacobi : paralléliser à l'intérieur de la boucle temporelle. Cependant, nous ne pouvons pas paralléliser directement car il y a dans la formule de Gauss-Siedel une dépendance entre l'état d'ordre $\ell + 1$ et l'état d'ordre ℓ. Nous devons donc partitionner intelligemment nos points afin que notre algorithme parallélisé nous continue de nous donner la bonne valeur.

Pour cela, nous utilisons la méthode dite des points « rouge et noir ». En effet, avec le shéma de Gauss-Siedel chaque point est dépendant des points voisins d'une arête mais indépendants des points voisins de deux arêtes. Nous pouvons alors attribuer à chaque point une couleur rouge ou noir et mettre à jour notre vecteur solution d'abord sur les points rouges puis sur les points noirs. C'est ce que nous faisons grâce au code suivant :

```
// Spatial loop
    for (int i=1; i<=Nx; i++){
        for (int j=1; j<=Ny; j++){
            solNew[i+(Nx+2)*j + 1] = coeff * ((sol[i+1+(Nx+2)*j+1]+solNew[i-1+(Nx+2)*j
            ↪  + 1])/(dx*dx) + (sol[i+(Nx+2)*(j+1) + 1]+solNew[i+(Nx+2)*(j-1) +
            ↪  1])/(dy*dy) - f[i + (Nx+2)*j + 1]);
            n_res_0 += (solNew[i+(Nx+2)*j + 1] - sol[i+(Nx+2)*j +
            ↪  1])*(solNew[i+(Nx+2)*j + 1] - sol[i+(Nx+2)*j + 1]);
        }
    }
```

3.2.2 Validation du code parallèle

Afin de valider le code parallèle nous calculons les résidus ainsi que les erreurs relatives pour une solution manufacturées. On obtient bien la convergence des résidus, on peut en déuire la validation de notre code.

En prenant les valeurs numériques $L = 10^6$, $\varepsilon = 10^{-5}$, $a = b = 1$, $\alpha = 0.5$, et en fixant le nombre de processus à 2 on obtient :

ℓ_{stop}	133
$\|\|u_{th} - u\|\|_{L^2}$	0.000593135
$\frac{\|\|r^{(\ell)}\|\|}{\|\|r^{(0)}\|\|}$	9.7237×10^{-5}

TABLE 5 – Valeurs en sortie du code parallélisé Gauss-Siedel pour $N_x = N_y = 30$

3.3 Comparaison globale entre les code Jacobi et Gauss-Siedel

L'idée ici est de comparer les codes parallèles Jacobi et Gauss-Siedel. Malheureusement, bien que nosu ayons fait tourner les deux codes sur la machine Cholesky, le code Gauss-Siedel a un temps d'execution très grand, et surtout trop grand pour qu'il soit pertinent de le comparer au code de Jacobi. Nous n'avons pas eu le temps de continuer l'analyse plus poussée.

4 Conclusion

Dans ce projet nous avons implémenté deux méthodes de résolutions itératives pour un problème aux différences finies. Nous l'avons d'abord fait pour des codes séquentiels suivant les mé-

thodes de Jacobi et de Gauss-Siedel puis nous avons parallélisé ces deux codes.

Les résultats obtenus sont plutôt encourageants, même si le code de Gauss-Siedel en parallèle est plus lent que le code de Jacobi en parallèle, ce qui ne devrait pas être le cas. Pour réussir à trouver l'origine du problème ou avancer plus loin dans la reflexion il faudrait tester nos codes sur des domaines avec un maillage plus fin car nous nous sommes limités à des domaines petits ($h \approx 0.033$) ce qui peut être la source de notre problème.