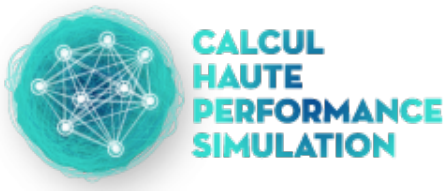

Projet de Programmation Numérique "Méthodes de krylov"

Dépôt Git

Encadré par : **Mr.Thomas Dufaud**

Réalisé par :

Mlle. Thiziri SALEM

Mlle.Hassani Fatma

M1 CHPS

6 mai 2022

Introduction

La résolution des systèmes linéaires est l'une des problématiques majeures en calcul numérique, cette résolution intervient dans plusieurs domaines (chimie, électricité, physique..).Dans notre projet on s'est intéressé à la résolutions numérique des EDP's obtenue par la méthode des éléments finies (équations algébriques matricielles obtenues à partir de la discrétisation d'équations aux dérivés partielles où les matrices sont tri-diagonales), dans l'intégralité de ce projet on se focalise sur le problème de transfert de la chaleur 1D pour tester et valider nos programmes.

Afin de résoudre un système linaire Ax=b , on distingue deux catégories de méthodes de résolution numérique, les méthodes directes et les méthodes itératives :

Les méthodes directes : Consistent à chercher l'inverse de la matrice A ($A^{-1}b$) ,cette opération est très coûteuse , si la taille du problème est grande, cela implique une complexité spatiale(au terme de stockage des grandes matrices, nombre d'opérations) et temporelle(temps de calcul)très élevé. Pour résoudre un système d'équations linéaires, on cherche à remplacer le système initial par un autre,plus simple, ayant la même solution, où la matrice A est (diagonale , orthogonale, tridiagonale, triangulaire, creuse). Les méthodes les plus utilisées dans le même contexte sont les suivantes :la méthode de Gauss et la factorisation LU. La méthode de factorisation consiste à réécrire la matrice A sous forme d'un produit de deux matrices triangulaires,donc la résolution de système Ax = b se réduit à la résolution des deux systèmes triangulaires (où L matrice triangulaire supérieure, et U matrice triangulaire inférieure) afin d'améliorer la complexité du problème. Avec la méthode de Gauss ou LU, la résolution de système Ax = b de taille n coûte O(n^3) n opérations arithmétiques par contre, si la matrice est creuse, telle que m « n^2 coefficients non nuls, un produit Ax (par l'algorithme le plus évident) réduit la complexité de O(n^2) à O(m), mais pour les matrices de taille très grande, ces méthodes sont considérées chéres en mémoire et en temps. Pour cette catégorie de matrices, l'utilisation des méthodes directes n'est pas très performant. [1]

Les méthodes itératives : À partir d'une approximation initiale, elles permettent d'obtenir des approximations de plus en plus " proche " (dans le sens d'une norme donnée) de la solution exacte du système en construisant une suite (x_n)n convergente vers la solution. Parmi ces méthodes(Jacobi, et Gauss-Seidel).

La résolution des problèmes d'analyse numérique se présente généralement par des équations aux dérivées partielles transformées à des problèmes linéaires , ces versions linéaires obtenus sont souvent mal conditionnés et de tailles très grandes, par conséquent l'utilisation des méthodes itératives n'est pas toujours éfficaces pour la résolution ce type de problème, un appel aux méthodes de projections est fortement recommandé, dans notre projet ont s'est intéressé aux méthodes de krylov qui sont plus efficaces dans le cas des matrices creuses et du grande taille. Leurs avantage : elles optimisent la mémoire et le temps de calcule et donc d'améliorer la complexité, ces méthodes évitent de faire le produit matrice matrice , ces méthodes manipulent le système linéaire au travers de produits "matrice-vecteur", [3]ce qui minimise le coût de calcul.

Pour les matrices au format particulière(creuses) de grande tailles, l'idéal est d'utiliser des méthodes de projection sur des sous-espaces de Krylov vu leurs efficacité et rapidité.dans ce travail,on s'est projeté sur les méthodes itératives de type de projection sur des sous-espaces de Krylov, précisément : méthode de **Gradient conjugué** qui sera exploitée prochainement dans la résolution des équations matricielles dédiées au transfert de la chaleur .

1 L'équation de poisson

Le problème de transfert de chaleur fait l'objet de plusieurs études, dans notre projet on a opté pour la résolution de l'équation de poisson à une dimension, [?] la particularité de sa matrice(tridiagonale creuse) est parfaite pour introduire la méthode du gradient .

Le problème se pose comme suit :

$$-\frac{\partial^2 U}{\partial x^2} = f(x) \qquad \forall\ x \quad \in]x_d, x_g[\qquad U(x_d) = U_d, \quad U(x_g) = U_g \tag{1}$$

L'intervalle $]x_d, x_g[$ est découpé en $N+1$ subdivisions $]x_i, x_i+1[$ comme suit :

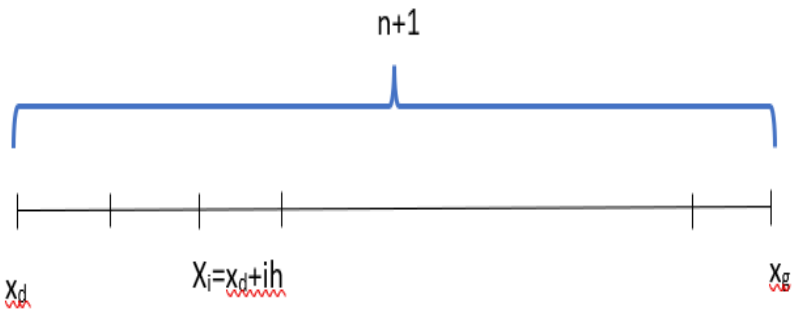

FIGURE 1 – schéma de discrétisation de l'intervalle

telle que $h = \dfrac{x_g - x_d}{n+1}$
la version discrète de problème (1), s'écrit alors(en utilisant une approximation de la dérivée seconde de U basée sur un développement de Taylor d'ordre 2) sous cette forme :

$$\frac{-U_{i-1} + 2U_i - U_{i+1}}{h^2} = f_i, \qquad \forall\ \ i \in \ \ \{1, N\} \quad U_0 = U_d, \quad et \ \ U_{N+1} = U_g \tag{2}$$

telle que U_i désigne U(x_i) et f_i désigne f(x_i) .

Le problème discrète se réduit alors à la résolution du système linéaire AU=F , où A $\in \mathcal{R}^N$ et sa représentation matricielle comme suit :

$$A = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \dots & \dots 0 \\ -1 & 2 & -1 & \dots & \dots 0 \\ 0 & -1 & 2 & -1 & \dots 0 \\ \vdots & & & \ddots & \\ 0 & \dots & \dots & 0 & -1 \quad 2 \end{pmatrix}$$

U et F sont les vecteurs de $\mathcal{R}^N$ définis par : $U = \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_N \end{pmatrix}$ et $F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$

1.1 Solution analytique du problème

On a $U''(x) = f(x)$ la solution de l'équation est U(x)=x. Dans notre cas la solution discrète est $x_i = x_0 + ih$ ou i=1 . . . N. et les conditions des bords sont : $U_0 = U_d$ et $U_{N+1} = U_g$.

2 Méthode itérative de Krylov

Les méthodes de krylov sont les plus adaptées pour la résolution des systèmes linéaires de grande taille dont les matrice sont creuses, ces méthodes utilisent mieux la structure creuse de la matrice et converge rapidement vers la solution , elles se basent sur la technique de projection sur un sous-espaces de Krylov .
Ces méthodes de projection sont basées sur un choix particulier du sous-espaces $\mathcal{K}$, le choix de x=$\mathcal{K}$ est lié au théorème de Cayley-Hamilton qu'on verra dans la suite.

Theorem 1 *Soit A une matrice carrée d'ordre n et p(λ)= det(λI-A), est le polynôme caractéristique de A ,on note p(λ)=0.*

Si on utilise ce théorème, on obtient l'existence d'un polynôme de degré (n-1) vérifiant A^{-1}=P(A), cela signifie que la solution du problème (AX=b) s'écrit maintenant A^{-1}, v=P(A)r_0, x=x_0+v ceci suggère à définir le sous-espace suivant :

$$K_m(A, r_0) = Span\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\}$$

telle que r_0=b-Ax_0, on note $K_m(A, r_0)$par K_m, les différentes versions de Krylov diffère selon le choix de $\mathcal{L}_m$, et le conditionnement de système, selon le théorème précèdent on écrit

$$A^{-1}b \approx x_0 + q_{m-1}(A)r0$$

q_{m-1} : un polynôme de degrée m-1, et x_0 on le pose $x_0 = 0$, on aura alors $A^{-1}b \approx q_{m-1}(A)b$.
Dans tous les techniques on applique la même approximation polynomiale, par contre le choix de $\mathcal{L}_m$, et les contraintes utilisées dans l'approximation se diffère d'une méthode à une autre, en générale on prend $\mathcal{L}_m = \mathcal{K}_m$ ou $\mathcal{L}_m = A\mathcal{K}_m$.

2.1 Sous-espace de Krylov

On définit un sous-espace de krylov :

$$K_m(A, r_0) = Span\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\}$$

La dimension de sous espace augmente à chaque étape de l'approximation, tout sous espace Krylov vérifie les propriétés suivantes :

- ⋆ K_m est le sous-espace de tous les vecteurs de $\mathcal{R}^n$, et on écrit v=P(A)r_0, x=x_0+v, telle que P est le polynôme caractéristique de degré< m-1.
- ⋆ P(A)v=0, le degré n le plus bas dans le polynôme est P(A) vaut 0.
- ⋆ Le degré de polynôme v est <n.

Une propriété assez importante sur les sous espaces de krylov est que dans un algorithme dont les itérés sont de l'espace de Krylov vérifient : $K_m(r_0) \subset K_{m+1}(r_0)$ [2]

3 La méthode du gradient conjugué

La méthode du Gradient Conjugué est très largement utilisée pour la résolution des systèmes linéaires, et très adaptée pour le cas des systèmes larges et creux, dernièrement elle s'est imposée comme une méthode itérative pour la résolution de système linéaire $Ax = b$, où A est symétrique définie positive et b le vecteur solution, notre motivation pour adapter cette méthode est simplement pour sa rapidité de convergence, et comme elle fait partie des méthodes de krylov elle manipule des produit matrices- vecteurs est donc considérée comme méthode de résolution moins coûteuse en terme mémoire et nombre d'opérations de calcul .[3]

3.1 Définition

Le gradient conjugué, est parmi les bonnes méthodes itératives pour la résolution des systèmes linéaires symétriques définies positives, son concept est de faire une projection orthogonale sur le sous-espace de krylov $k_m(r_0, A)$, en effet l'idée principale du gradient conjugué consiste à construire itérativement une suite(x_m) de sous espace krylov, qui est croissante , en minimisant l'erreur |r=Ax-b|. La suite (x_m) est définie par son premier terme x_0 et une relation de récurrence $x_{m+1} = x_m + \alpha_m P_m$ où α ,ainsi le pas, et P_k la direction.
L'aspect de la méthode est de trouver une suite de "n" directions conjuguées ensuite calculer les coefficients α_m.

3.1.1 Propriétés de la méthode du gradient conjugué

x_m vérifie les les deux conditions suivantes :
i) $x_m \in x_0 + K_m(r_0)$

ii) $x_m \perp K_m(r_0)$

3.2 Développement de l'algorithme

Le développement de la version de gradient conjugué est basée sur les propriétés i) et ii)

Lemma 1 *Pour l<m on a* $r_l \in K_m(r_0)$, $(r_l, r_m) = 0$ *alors* $K_m(r_0) = Span\{r_0, \ldots, r_{m-1}\}$.

En appuyant sur le lemme, on peut définir un vecteur P_m comme direction du changement entre deux itérés :
$x_{m+1} = x_m + \alpha_m P_m$, $P_0 = r_0$ telle que α est non nul.

Lemma 2 *La direction* P_k *doit vérifier :*

i) $P_m \in K_{m+1}(r_0)$

ii) $((y, Ap_k) = 0, \forall y \in K_m(r_0))$

La propriété ii) implique $r_{m+1} = b - Ax_x - \alpha AP_m = r_m - \alpha AP_m$, c'est à dire la différence entre $r_{m+1} - r_m$ est orthogonal à $K_m(r_0)$, P_k est orthogonal à un sous-espace de dimension 1 dans $k_{m+1}(r_0)$, alors cela montre que la direction sera déterminée d'une manière unique. On dit que P_m est A-conjugué à $k_m(r_0)$, ce qui explique l'origine du nom de la méthode.

3.3 GC algorithme version séquentielle

Algorithm 1 Version séquentielle

$\mathbf{p} = \mathbf{b}$ /*$\mathbf{p}$ décrit les directions possibles/*
$\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ /* le résidu/*
$\mathbf{x} = \mathbf{x_0}$ /*on initialise x à x0/*.
for($i = 0$ to max_iter)
$\mathbf{r_{old}} = \mathbf{r}$
$\alpha = \frac{\mathbf{r} \cdot \mathbf{r}}{\mathbf{p} \cdot \mathbf{Ap}}$ /*calcule la proportion à rajouter à la solution esimé pour chercher une nouvelle direction/*
$\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$ /*la nouvelle solution approché/*
$\mathbf{r} = \mathbf{r} - \alpha \mathbf{Ap}$ /* mettre a jour comme on rajoute/* $\alpha\mathbf{p}$ à $\mathbf{x}$, ce qui change aussi la différence $\mathbf{b} - \mathbf{Ax}$.
/*tester la convergence/*
if $\mathbf{r} \cdot \mathbf{r} <$ tolerance
break
$\beta = \frac{\mathbf{r} \cdot \mathbf{r}}{\mathbf{r_{old}} \cdot \mathbf{r_{old}}}$ /* mettre à jour le changement dans la direction/*
$\mathbf{p} = \mathbf{r} + \beta \mathbf{p}$ /* ajuster la recherche sur la direction/*
endfor
la solution est $\mathbf{x}$

3.4 Mise en oeuvre

L'algorithme de gradient conjugué est simple à mettre en oeuvre : pas coûteux, et on utilisant cet algorithme, on aura pas besoin de stocker la valeur de(r,x,β), elles sont écrasées à chaque itération ce qui permet d'avoir un gain très important en mémoire.
Il se compose de 4 opérations opérations vectorielle (3 sont de type AXPY) et le calcul de produit matrice vecteur. Dans la plupart des cas, les matrices des systèmes EDP possèdent une format particulière (creuses), comme ce n'est pas nécessaire de stocker toutes la matrice(uniquement les éléments non nuls), alors cela améliore la complexité spatial car cette particularité de la matrice implique une linéarité de coût par rapport au nombre d'inconnues.
En poisson 1D la matrice A est tridiagonale, alors le coût de produit matrice vecteur se réduit proportionnellement aux nombres d'éléments non-nuls de la matrice.[4]

3.5 Convergence

La méthode du gradient conjugué peut théoriquement être considérée comme une méthode exacte où la convergence est assurée en n itération au maximum, telle que, n est la taille de la matrice dans le cas d'absence d' erreur arrondi. En pratique, la méthode du gradient conjugué est instable par conséquent la notion de solution exacte n'est pas vrai, car la plupart des directions ne sont pas conjuguées en pratique, en raison de la nature dégénérative de la génération des sous-espaces de Krylov.
La méthode de gradient est souvent utilisée comme méthode itérative dans ces dernières années, elle fournie une très bonne approximation de la solution exacte du système en un nombre limité d'itérations, en d'autre terme converge rapidement .[4]

4 Tester et valider l'algorithme avec scilab

Suivant les étapes inscrites dans l'algorithme, afin de tester la convergence, on calcule la norme de résidu "r " qui est $r = \frac{|Ax-b|}{|b|}$, et on la compare à notre epsilon fixé qui est l'erreur toléré, ensuite si la condition : l'erreur calculé <= epsilon est vérifiée la solution est atteinte .
Le nombre maximale des itérations est fixé au début(ne pas dépasser n car la méthode ne dépasse pas n pour converger vers la solution).
Le programme est testé sur l'exemple initial **Poisson 1D** dont sa solution analytique est connue déjà afin valider notre programme, et tracer les graphe des convergences.

4.1 Problème teste :

On cherche à résoudre le système d'équations linéaires : A x = b, où :

- A est une matrice symétrique définie positive

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Afin de tester nos programmes (scilab). On garde les même exemple(Poisson 1D), et on fait varier la taille et les conditions du bord dans le second membre.

- avec b=(1,0,0,0).
- x_0=(0,0,0,0), solution initiale
- Pour n=10 le programme converge vers la solution en 4 itérations.
- Pour n=50, le programme converge vers la solution en 50 itérations.
- Pour n=100, le programme converge vers la solution en 100 itérations.

- Pour n=200, le programme converge vers la solution en 200 itération.

- Pour n=300, le programme converge vers la solution en 300 itérations, alors pour une matrice d'ordre n dans notre exemple(poisson 1D), le programme converge vers la solution en n itérations.

4.2 Représentation d'erreur

On présente la norme résiduelle associée à chaque itération, en variant la taille du système et les bornes du second membre, dans chaque cas,on trace son graphe de convergence par rapport aux nombres d'itérations en log(10).

- on pose b=(1,0...,0) et 10<=n<=300.

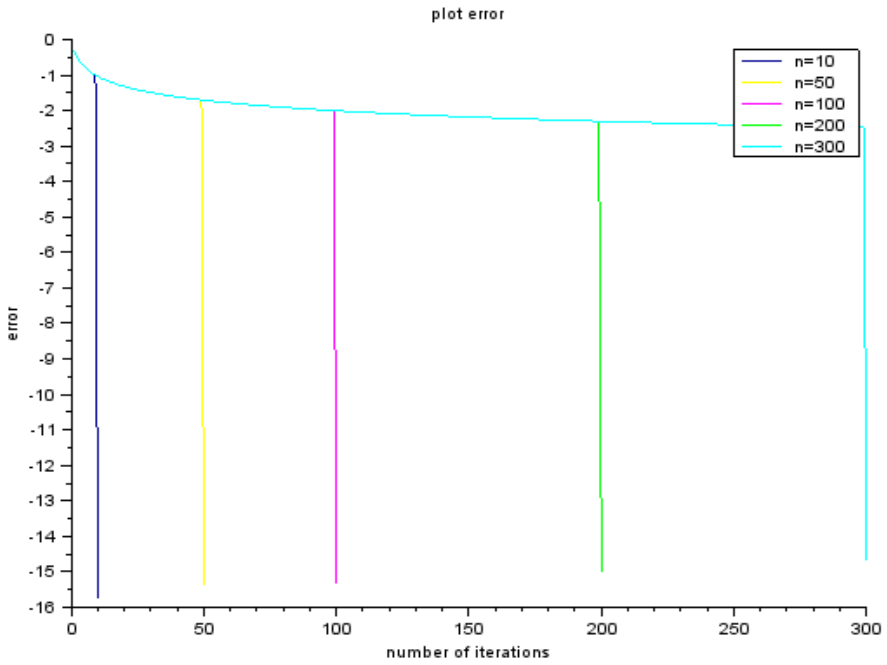

FIGURE 2 – **Cas1** : graphe de convergence en log10 en fonction de nombre d'itérations

- dans ce cas on pose b=(4,0. . .,0,10) et 10<=n<=300.

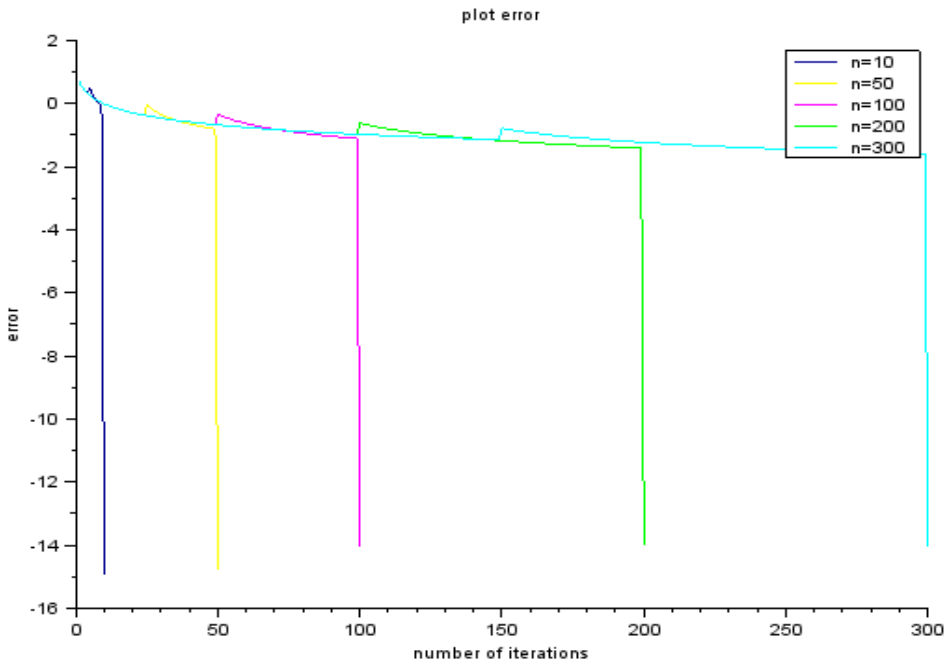

FIGURE 3 – **Cas 2** : graphe de convergence en log10

- Cas 3 : b=(50,0,. . .,0,45) et 10<=n<=300.

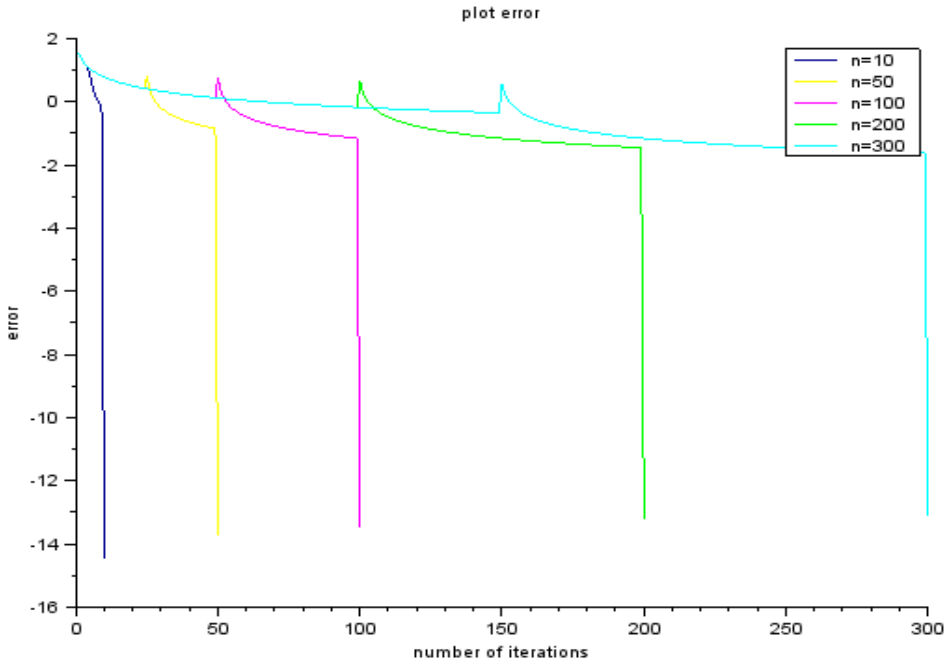

FIGURE 4 – **Cas 3** : graphe de convergence en log10

Analyse des résultats

Comme c'est déjà présenté au-dessus, on a présenté l'erreur en log(10) par rapport au nombre d'itérations.
on voit clairement que la solution calculée converge lentement, ensuite à la dernière étape, on remarque une réduction importante de l'erreur, et cela quelque soit la taille de la matrice. La méthode converge en 1er cas quand l'erreur est au bord de 10^{-16} , par contre dans les autres cas la convergence est atteint avec une erreur de 10^{-14} .

5 L'implémentaion et La forme de stockage

5.1 Format de stockage :

En global les matrices résultantes après la discrétisation des EDPs comporte de nombreux éléments non nuls, alors l'idéal dans ces cas est de ne pas stocker toute la matrice (creuse) et de considérer un stockage creux pour ne pas stocker une grande partie des coefficients nuls afin de réduire considérablement la complexité des résolutions de systèmes.
Il existe plusieurs méthodes de stockage, dont le stockage en général bande, l'efficacité de ce stockage (mais aussi la résolution) est d'autant meilleure que les termes non nuls sont regroupés autour de la diagonale. Le cas idéal pour ce type de stockage est le cas où la matrice à une structure par bandes avec un nombre de bandes très faible par rapport à la dimension de la matrice [6].
Dans le stockage bande on stocke uniquement la bande, autrement toutes les diagonales situées entre les deux diagonales contenant les éléments non nuls les plus éloigné de la diagonale principale comme colonnes d'une matrice rectangulaire dont le nombre de lignes est égale à l'ordre de la matrice creuse tandis que le nombre de colonnes de la matrice est égale à la largeur de la bande.
[7]

5.2 BLAS et LAPACK :

Les bibliothèques BLAS et LAPACK fournissent une implémentation pour les matrices stockées par bande . L'implémentation des fonctions pour les matrices bandes est notée GB pour général bande, tandis que pour les matrices denses, on note GE pour général.

Stockage GB :

Une matrice de dimension m × n avec kl sous-diagonales et ku sur-diagonales peut être stockée de manière compacte dans un tableau à deux dimensions avec kl + ku + 1 lignes et n colonnes.
Les colonnes de la matrice sont stockées dans les colonnes correspondantes du tableau, les diagonales de la matrice sont stockées dans les lignes du tableau. Ce stockage ne doit être utilisé que si kl, ku min(m, n).dans notre cas (kl=1 et ku=1)[**?**]

Format Col-majeur

Format col-majeur indique que les tableaux sont de colonne principale(priorité colonne), c'est par rapport à la méthode de stockage de la matrice, elle est stockée d'une manière que chaque colonne est placée dans une ligne. Afin d'améliorer la complexité des calculs, par exemple si on veut calculer b=A*x avec x un vecteur colonne suivant le stockage col-majeur, on a qu'à multiplier un scalaire $*vecteur$, $som_i A(:,i)x(i)$.
ce format on le trouve en Fortrans et en quelques bibliothèques comme LAPACK.
En générale c'est plus efficace en terme de bande passante en mémoire, d'usage des caches d'une façon plus performante et d'accès dans le même ordre telle qu'ils sont stocké en mémoire, alors dépendant de la méthode de stockage d'une matrice qu'on choisi l'algorithme qui convient à notre méthode adapté pour stockage interne.
Dans notre programme on utilise col-majeur comme on fait appel à bibliothèque cblas.

5.3 L'implémentation en C

Afin d'implémenter la méthode du gradient, on avait besoin d'implémenter des fonctions qui réalisent chaque-unes une tache particulière (multiplication matrice vecteur, multiplication vecteur vecteur, somme de deux vecteurs) , la fonction principale "gradient-conjugué" fait appel à ces fonctions afin de réaliser les calcules (multiplication, somme...) .
On a fait appel à la bibliothèque BLAS pour effectuer le produit matrice vecteur, ensuite pour optimiser la façon de stockage des matrice, on a utilisé le stockage bande en cool-majeur avant de faire appel à CBLAS, cette façon de stockage est très efficace pour les matrices creuses.

6 Parallélisation

6.1 Notion de parallélisme

La parallélisation dans le domaine de calcul haute performance est un ensemble de techniques logicielles et matérielles permettant l'exécution de codes en parallèle sur plusieurs coeurs de calcul dans l'intérêt de réduire le temps de restitution, effectuer de plus gros calculs et d'exploiter le parallélisme des processeurs modernes (multi-coeurs, multithreading).

Notions indispensable pour le parallélisme

Système à mémoire partage

C'est un système informatique multiprocesseur avec une mémoire répartie en plusieurs noeuds , chaque portion n'étant accessible qu'à certains processeurs comme le système SMP ou Symmetrical Multi-Processing qui est un système avec une machine constituée de plusieurs processeurs identiques connectés à une unique mémoire physique.

Système à mémoire distribuée

On dit qu'un système est à mémoire distribuée lorsque sa mémoire est répartie en plusieurs coeurs, chaque portion n'étant accessible qu'à certains processeurs, comme le systeme NUMA ou Non-Uniform Memory Access qui un systeme avec est une machine constitué de plusieurs processeurs connectés à plusieurs mémoires distinctes.

Thread ou flot d'exécution

C'est une suite logique séquentielle d'actions résultat de l'exécution d'un programme.

Processus

Instance d'un programme. Un processus est constitué d'un ou plusieurs threads qui partagent un espace d'adressage commun.

Le calcul parallèle

: Consiste en le découpage d'un programme en plusieurs tâches qui peuvent être exécutées en même temps dans le but d'avoir une exécution plus rapide du programme et une résolution de problèmes plus gros (plus de ressource matérielle accessible, notamment la mémoire).

6.2 Présentation d'MPI

MPI est un outil indispensable dans le domaine de calcul haute performance. C'est une norme conçue en 1993-19994 pour le passage de message entre ordinateurs distants ou entre processus dans un ordinateur multiprocesseur.[8]

Avant de paralléliser notre programme, on effectue d'abord une analyse pour identifier les opérations les plus coûteuses, ainsi que les dépendances entre les tâches.

6.3 Analyse de l'algorithme GC

Les opérations de l'algorithme du Gradient Conjugué en utilisant la notion de BLAS peuvent être résumées comme suit :

L'itération du base de l'algorithme du Gradient Conjugué s'écrit :

1) Axpy est un calcul scalaire
2) Produit Matrice-vecteur : Calcul de A * p_k
3) Axpy pour la mise à jour de x_k et r_k
4) Dot est un calcul scalaire
5) Axpy pour la mise à jour de p_{k+j}

on ordonne les coûts de ces opérations de calculs par itération comme suit :

* 1 produit matrice vecteur :O(n^2)
* 2 produit scalaires O(n)
* 3 mises à jours de vecteurs (AXPY) O(n)

Le choix de parallélisation

On constate que les produits matrice-vecteurs (A*x) que ça soit dans l'initialisation où à l'intérieur de la boucle constituent la partie la plus chére dans l'algorithme(mémoire et calcule) , comment peut on décomposer une telle opération en sous taches qui se réalisent en parallèle ?

6.4 Première approche

Soit np le nombre de processus esclaves , la matrice A d'ordre n est stockée en stockage bonde par col-major(dans notre cas c'est le vecteur AB) alors le produit matrice vecteurs ici va être décomposé en plusieurs produits scalaires en "n produit scalaire" telle que n est l'ordre de la matrice (en plusieurs produits lignes* colonnes).

On considère une tache de calcul de b_i comme un produit scalaire de la iéme ligne* n ième colonne $b_i = a_i * x_i$, et on garde toujours même version séquentielle du programme, le changement sera dans le calcule de produit matrice vecteurs qu'on répartie sur les "np" processus comme suit ou chaque processus s'occupe de calculer un élément b_i :

Algorithm 2 Version0 parallèle

les np processus se repartie pour effectuer le calcule
calculer $b(i), \quad i = 1, .., n$
for$(i = 1; i < n - 1; i + +)$
for$(j = 0, j < lab; j + +)$
$b(i) = AB[i * lab + j] * x[j + i - 1]$
endfor
endfor
calculer b(0) :
for$(j = 1; j < lab; j + +)$
b(0)= AB[j].x(j-1)
for
calculer b(n-1) :
endfor
for$(j = 0; j < 2; j + +)$
b(n-1)=AB[(n-1)*lab+j]*x[j+(n-1)-1]
endfor

Problème de granularité : Dans le cas , où la matrice A est de taille très grande(n> 10^8),on se retrouve avec **n** processus pour réaliser chaqu'un "1 produit scalaire" ce qui implique un coût de communications assez grand face à la réduction estimée, donc on constate que la version parallèle est plus coûteuse par rapport à la version séquentielle (besoin de nombreux unités de traitement physiques), d'où la nécessité d'augmenter la granularité de la tâche considérant **une tâche** comme un ensemble de produit scalaire autrement dit un produit sous-matrice vecteur.

6.5 Deuxième approche :

En cette deuxième approche, on prend une tâche comme étant un produit sous-matrice vecteur, on présente la problématique sous cette forme :
on considère $y = Ax$, où A$\in \mathcal{R}^{n*n}$, $y \in \mathcal{R}^n$, $x \in \mathcal{R}^n$,
la matrice A est une matrice de poisson 1D tridiagonale, sous la forme suivante :

$$A = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots 0 \\ -1 & 2 & -1 & \dots & \dots 0 \\ 0 & -1 & 2 & -1 & \dots 0 \\ \vdots & & & \ddots & \\ 0 & \dots & \dots & 0 \ -1 & 2 \end{pmatrix}$$

On découpe la matrice A en bloque lignes A_i telle que A_i est détaille $n_i \times n$, où n_i nombre de lignes et n le nombre de colonne, et $\mathrm{A}_i \in \mathcal{R}^{n_i * n}$, soit **np** le nombre de processus alors initialement pour **n** on à :

$$y = [0 \dots 0]$$

soit y_i le vecteur résultat de la multiplication sous matrice $A_i * x$

$$y = \sum_{i=0}^{P-1} R_i^T A_{i*} x = \overset{p_0}{\begin{pmatrix} y_0 \\ 0 \\ 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \end{pmatrix}} + \dots + \overset{p_i}{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ y_i \\ 0 \\ \vdots \\ 0 \end{pmatrix}} + \dots\dots \dots + \overset{p_{np}}{\begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ 0 \\ y_{np} \end{pmatrix}}$$

$$\text{All-reduce-p}_i = \quad (p_0, \dots \quad \dots, \text{p}_i, \dots \quad \dots, p_{np})$$

ALL-reduce sur tous les processus .

problème : Comment stocker la matrice A_i en prenant en consédiration la particularité de notre matrice tridiagonale(matrice creuse) ?

Solution 1 :Stockage dense

Stocker la matrice en stockage dense est simple pour le mettre en oeuvre, la matrice va être découpée de la manière suivante :

$$A = \begin{array}{c} A_0 \longrightarrow \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ A_{np-1} \longrightarrow \end{array} \left(n \left\{ \begin{array}{|cccccc|} \hline 2 & -1 & 0 & 0 & \dots & \dots \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \hline 0 & \dots & -1 & \ddots & 2 & -1 \ \ 0 \\ \vdots & \dots & 0 & \ddots & -1 & 2 \ -1 \\ \vdots & \dots & 0 & & 0 & -1 \ \ 2 \\ \hline \vdots & \dots & \dots & \dots & \dots 0 -1 & \ddots \\ \vdots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & 0 \\ \hline \end{array} \right. \right. \begin{array}{|ccc|} \hline 0 & \dots\dots & 0 \\ \vdots & \dots\dots & \vdots \\ \vdots & \dots\dots & \vdots \\ \hline \vdots & \dots\dots & \vdots \\ 0 & \dots\dots & \vdots \\ -1 & \dots\dots & 0 \\ \hline 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \\ \hline \end{array} \left. \right)$$

Inconvénient : On analysant la **complexité** spatiale et temporale, on trouve qu'elles sont estimées : à **O(n$_i$*n)**, et si on la compare à la complexité du code séquentielle (dans laquelle la

matrice est stocké en stockage bande format col-majeur) qui est de **O(3*n)**, on constate que la version parallèle dans ce cas est plus coûteuse ($O(n_i * n)$>O(3*n)), d'où une contradiction de but de la parallélisation qui est l'optimisation de programme afin réduire sa complexité.

Solution 2 :Stockage en générale bande

On considère le même découpage de la matrice A , par contre A dans ce cas est stockée en générale bande, alors pour le 1er bloc A_i, on stocke juste la matrice bande en commençant par le 1er élément de ligne en couvrant les trois diagonales, par contre pour les autres blocs $A_i, i = 0 \dots np-1$, le stockage de la bande qui se constitue de 3 diagonales implique le stockage de tous les éléments nulles du A_i qui se positionnent avant la diagonale, dans ce cas sa complexité sera approximativement la même avec le stockage dense(quadratique).

Solution 3 : Stockage des sous matrices en stockage bande

Le découpage proposé dans cette solution consiste à décomposer la matrice A à des sous matrices $A_{i,j}, i = 0 \dots np-1, j = 0 \dots np-1$, où **np** représente le nombre de processus, par contre on stockera uniquement $A_{i,i}$ en stockage générale bande ce qui l'idéale pour notre cas où les éléments de la matrice sont déjà connu, alors A sera découpé en sous matrices $A_{i,j}$, on s'intéresse uniquement aux sous- matrices de la diagonale $A_{i,i}$(matrice de de poisson) .
Afin de simplifier encore le produit matrice vecteur A.x, on applique le produit uniquement sur des sous-matrices de poisson présentées par : $A_{i,i}$ de taille $n_i * n_i$ par conséquent on peut faire appelle à notre fonctions précédente du version séquentielle pour la construction de la matrice $A_{i,i}$, $A_i \in \mathcal{R}^{n_i * n_i}$.
En résumé cette méthode remplace l'opération produit matrice A.x par $A_{i,i} * x_i$, ou x_i représente les "n_i" éléments de vecteur x associé au bloc de la matrice $A_{i,i}$.
Le vecteur x existe sur chaque processeur pour éviter les communications, de e toute manière c'est pas le plus gros consommateur en mémoire.

$$A = \begin{pmatrix} \boxed{\begin{matrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{matrix}} & \begin{matrix} 0 & \dots & \dots \\ 0 & \dots & \dots \\ -1 & 0 & \dots \end{matrix} & \begin{matrix} 0 & \dots\dots & 0 \\ \vdots & \dots\dots & \vdots \\ \vdots & \dots\dots & \vdots \end{matrix} \\ \begin{matrix} 0 & \dots & -1 & \ddots \\ \vdots & \dots & 0 & \ddots \\ \vdots & \dots & 0 & \\ \vdots & \dots & \dots & \\ \vdots & \dots & \dots & \\ 0 & \dots & \dots & \end{matrix} & \boxed{\begin{matrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{matrix}} & \begin{matrix} \vdots & \dots\dots & \dots \\ 0 & \dots\dots & \vdots \\ -1 & \dots\dots & 0 \end{matrix} \\ \begin{matrix} \dots & 0 & -1 & \ddots \\ \dots & \dots\dots & 0 & \ddots \\ \dots & \dots\dots & 0 & \end{matrix} & \boxed{\begin{matrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{matrix}} \end{pmatrix}$$

$$X = \begin{pmatrix} x_0^1 \\ \vdots \\ x_{n_0}^0 \\ \hline \vdots \\ \hline x_0^i \\ \vdots \\ x_{n_i}^i \\ \hline \vdots \\ \hline x_0^{np} \\ \vdots \\ x_{np}^{np} \end{pmatrix}$$

Produit matrice vecteur en parallèle :

Le but dans cette section est de paralléliser le produit matrice vecteur, après la décomposition de A en sous matrice $A_{i,i}$ et X en x_i, alors on peut considérer une $tache_i$: le produit $A_{i,i} * x_i$, donc paralléliser le produit consiste à repartir ces tâches entre l'ensemble de processus "np"

Si on analyse la matrice A, et le découpage proposé au dessus, on remarque qu'aux bords des blocs des sous matrices $A_{i,i}$, on trouve des éléments non nuls qui sont pas prise en compte dans le produit, dans le 1er bloc au bord de $A_{0,0}$ une valeur a droite, ainsi dans le dernier bloc $A_{np,np}$ une valeur a guache, et dans les blocs intérieurs(i=1...np-2) aux bords de chaque $A_{i,i}$, on à deux éléments un à gauche et l'autre à droit de valeur -1 qui sont pas pris en considération dans l'opération produit $A_{i,i} * x_i = y_i$, la solution proposée consiste à calculer le produit telle qu'il est, ensuite rajouter leurs produit à la composante correspondante(début, où fin) dans le vecteur y_i , l'enregistrer dans un vecteur y de taille n, et enfin un All-reduce() somme des y sur chaque p_i, comme c'est décrit dans l'algorithme suivant :

Algorithm 3 algorithme parallèle de produit de sous-matrice vecteur

sur chaque processus p_i faire :

–> calculer la langeur du bloc (nloc)

–> calculer les indices de début et fin de bloc (ideb,ifin)

–> copier $x_i[ideb, ifin]$ dans le sous vecteur x_i de taille nloc

–> faire le produit sous-matrice vecteur $y_i = A_{i,i} * x_i$

–>changer les valeurs début et fin dans le vecteur y_i selon le rang de processus

if (rang=0)

–>changer $y_0[nloc-1] = y_0[nloc-1] - x[ideb-1]$

else – if (rang=np-1)

–>changer $y_{np-1}[0] = y_{np-1}[0] - x[fin]$

else

–>changer $y_i[0] = y_i[0] - x[fin]$

–>changer $y_i[nloc-1] = y_i[nloc-1] - x[fin]$

–>copier y_i dans y

–>**All-reduce** sum de y dans y

le All-reduce sum de y sur chaque p_i y=$(y_0, y_1, \ldots\ldots, y_i, \ldots\ldots, y_{np-1})$, s'explique comme suit :

$$\begin{pmatrix} y_0 \\ 0 \\ 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \end{pmatrix} + \ldots + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ y_i \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \ldots\ldots \ldots + \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ 0 \\ y_{np-1} \end{pmatrix}$$

Analyse complexité :

On fait une analyse comparative selon les différentes méthodes de stockage de la sous-matrice $A_{i,i}$, si on la stocke en stockage dense sa complexité en mémoire et arithmétique est de $O(n_i^2)$, elle est moins coûteuse par rapport au première alternative de découpage en bloc ($A_i \in \mathcal{R}^{n_i * n}$ dont la méthode de stockage est générale bande ($O(n_i * n)$).

La complexité d'un stockage dense de sous matrice $A_{i,i}$ est inférieur au coût de stockage d'une matrice complète en stockage bande O(3n), ce qu'est le cas du la version séquentielle ($O(n_i^2) < O(3n)$), cela est du au stockage dense de $A_{i,i}$.
Avec un stockage générale bande de sous-matrice $A_{i,i}$, la complexité devienne linéaire $\mathbf{O(3}n_i\mathbf{)}$, ce qui implique une réduction de la complexité initiale de O(3n) à O($3n_i$) en espace et en temps, telle que $n_i = n/np$.

L'intérêt d'utilisation d'un calcul parallèle ici est de deviser la taille de domaine de recherche par le nombre de processus , le gain en performance devrait être important selon la théorie et nous cherchons la qualité dans la section suivante(scalabilité).

7 Option de l'optimisation de temps

L'ajout des flags de compilation pour l'optimisation du code peut réduire le temps de compilation et les performance de code , pour cela, on essayera les trois flags -O2, -O3, -Ofast pour le code séquentiel, ainsi que parallèle, on mesurant le temps d'exécution pour chaque flags, on fait l'étude sur :
on ordonne les coûts de ces opérations de calculs par itération comme suit :

* une matrice de taille n=1000
* second membre $b = [5, 0, \ldots, 0]$
* précision=0.000001

Version séquentielle

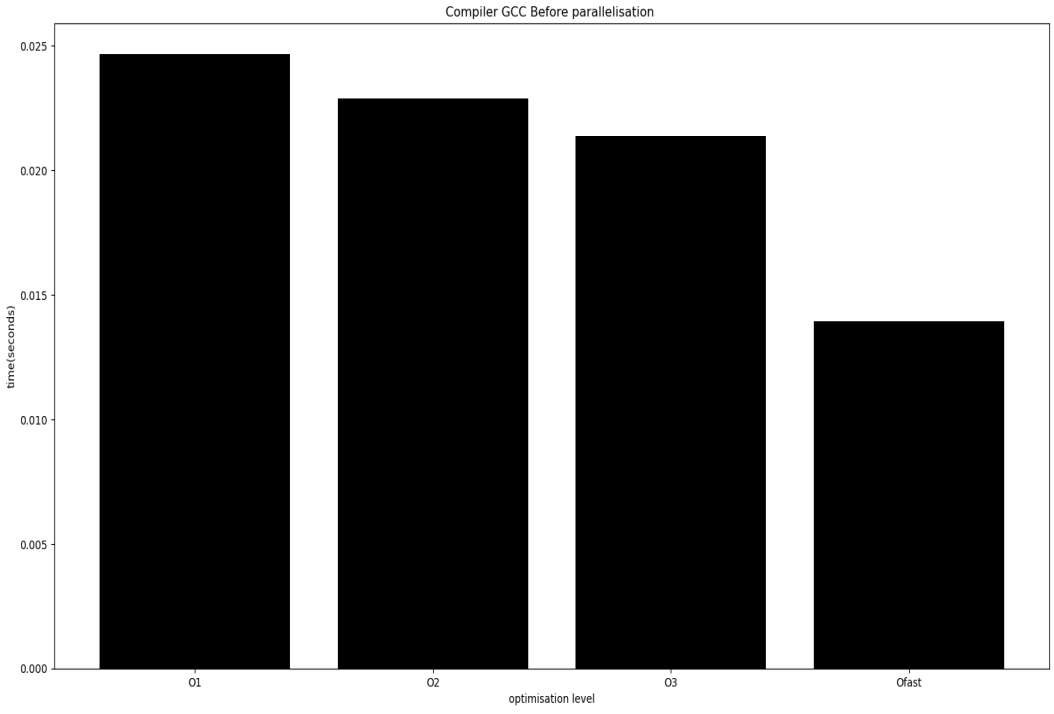

FIGURE 5 – Version séquentielle

Version Parallèle

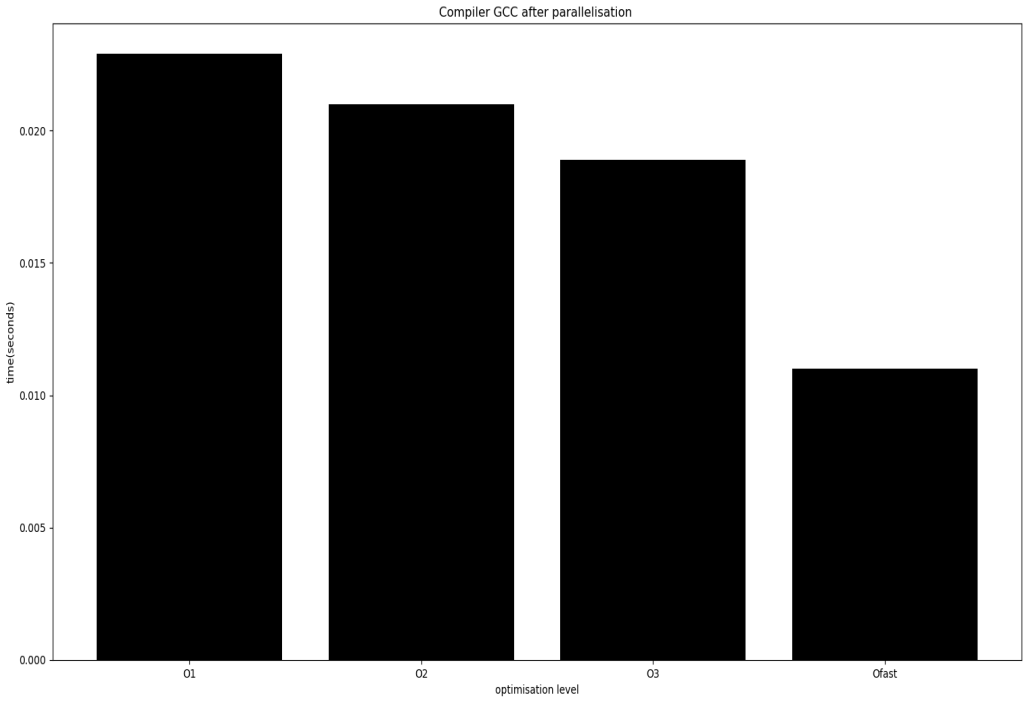

FIGURE 6 – Version parallèle

On constate une réduction importante dans le temps d'exécution dans le code parallèle et séquentielle, du à l'utilisation de la flag de compilation **-Ofast**.

Dans ce qui suit une étude comparative de temps d'exécution de la version parallèle et séquentielle pour différent taille de matrice :

	temps d'exécution version séquentielle	temps d'exécution version parallèle
1000	0.013 S	0.01S
5000	0.40 S	0.35 S
10000	1.53 S	1.41S

TABLE 1 – Table de comparaison de temps d'exécution

On remarque une différence légère de temps d'exécution entre les deux versions(temps d'exécution : séquentielle-parallèle), en augmentant la taille de la matrice, le code parallèle améliore les performances mais avec une différence assez petite.

8 Scalabilité

Dans cette section on s'intéresse aux différentes méthodes d'évaluation d'un algorithme(performance d'un programme), parmi eux on distingue :

8.0.1 Scalabilité forte

Un programme est dite fortement scalable, si il permet de résoudre, un problème autant vite , que l'on rajoute proportionnellement des processeurs.

8.0.2 Scalabilité faible

On dit qu'un programme est faiblement scalable, si il permet de résoudre un problème avec des tailles différentes en temps comparable, autrement dit est d'augmenter la taille autant que le nombre de processus.

8.0.3 Calcul de scalabilité forte

L'étude de la scalabilité forte est faite pour une matrice de taille 5000, en gardons les même valeurs pour les autres paramètres.

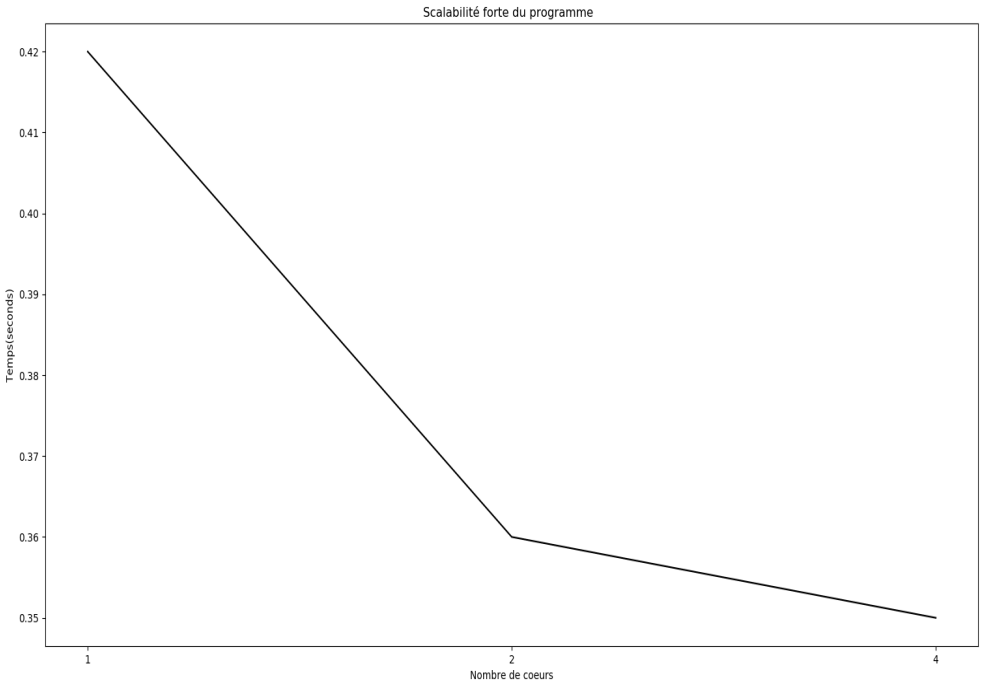

FIGURE 7 – scalabilité forte

On remarque que le programme n'est pas fortement scalable , donc pas linéaire comme on s'attendait idéalement(cela est principalement au temps de communication entre processus), mais on constate que l'augmentation de nombre de processus diminue le temps d'exécution, même si avec une différence légère.

8.0.4 Calcul de scalabilité faible

L'étude de la scalabilité faible est faite pour une matrice de taille 1000 pour 1 coeurs, en gardons les même valeurs pour les autres paramètres.

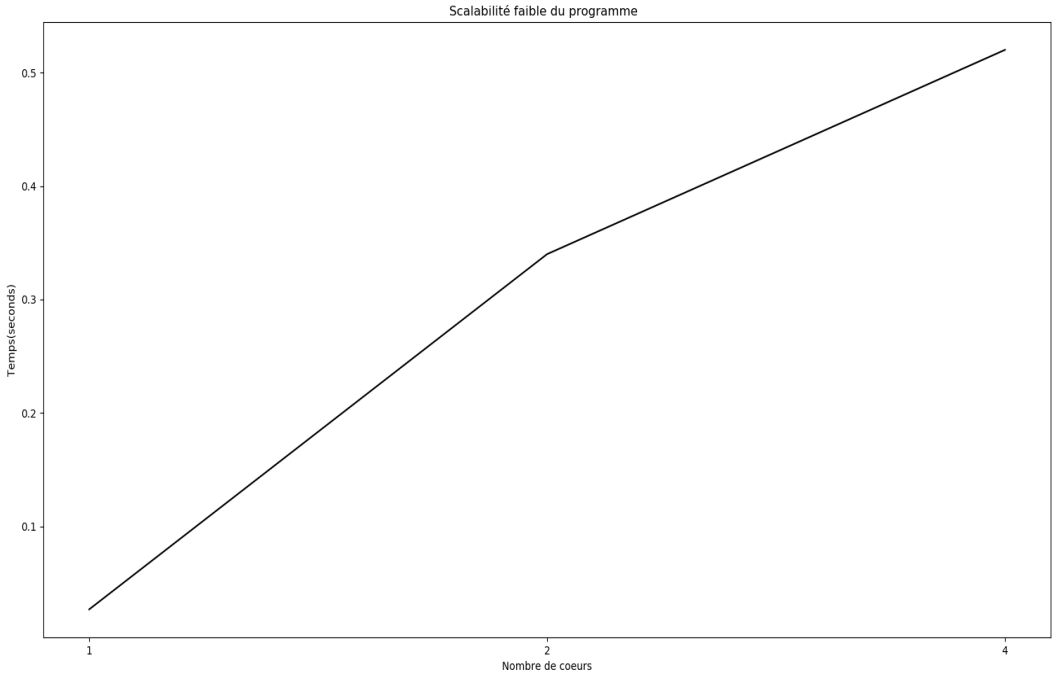

FIGURE 8 – scalabilité faible

Avec l'augmentation du taille du problème et le nombre de processus à la fois , on a eu un temps d'exécution qui est pas stable, mais comme on remarque l'augmentation du temps est assez raisonnable.

Conclusion

Durant ce modeste travail ,Nous avons vu en première partie une présentation de l'équation de la chaleur à une dimension , ensuite on a passé à l'explication des méthodes de krylov, particulièrement la méthode du gradient Conjugué. Pour bien illustrer et comprendre cette méthode de résolution numérique, on a commencé par une implémentation sous scilab avant de passer à la programmation en langage C. Après avoir implémenté la méthode d'une manière séquentielle, l'étape suivante était de passer à l'optimisation ,autrement dit :La parallélisation. Plusieurs solutions ont été proposées pour paralléliser le Gradient Conjugué, notre choix était de suivre une approche plus centralisée qui consiste à paralléliser le produit matrice-vecteur, pour ce faire on a fait appel à MPI(message passing interface), afin de distribuer les tâches à effectuer (le produit matrice-vecteur) sur plusieurs processus .
Enfin pour estimer la qualité de notre parallélisation, on a fait une mesure de performance en calculant la scalabilité forte et faible, afin de mettre en évidence la réduction en terme de temps d'exécution.
Certe on a pas pu avoir un gain énorme mais on a réussi a réduire considérablement le temps d'exécution.

Références

[1] Cours de l'université Compiègne, France

[2] Yousef Saad. Iterative Methods for Sparse Linear Systems

[3] SADEK EL MOSTAFA.Méthodes itératives pour la résolution d'équations matricielles

[4] Michel KERN .Analyse numérique avancée, Sup-Galilée – MACS 2

[5] D. DELESALLE,L. DESBAT,D. TRYSTRAM.Résolution de grands systèmes linéaires creux par méthodes itératives parallèle, M2AN - MODÉLISATION MATHÉMATIQUE ET ANALYSE NUMÉRIQUE.

[6] Bertrand Thierry.Maillage et Éléments Finis

[7] J. Erhel .Résolution de systèmes linéaires creux par des méthodes directes
par méthodes itératives parallèles

[8] David Dureau,Marc Pérache.Cours Programmation Parallèle et Distribuée