Projet de Méthodes de Programmation Numérique Avancée "Algorithme de Lanczos "

Encadré par : **Mme Nahid Amad**

Réalisé par :

Aly Cissé
Selma Khadir
Lougani Faouzi

M2CHPS

18 décembre 2021

Table des matières

1 Problématique **3**
- 1.1 Notations et définitions . . . 3
- 1.2 Bases de Krylov . . . 3
- 1.3 L'algorithme de Lanczos . . . 3
  - 1.3.1 Etapes de l'algorithme : . . . 4
  - 1.3.2 Pseudo Code méthode de Lanczos : . . . 5

2 Cas séquentiel **6**
- 2.1 Description de l'algorithme proposé . . . 6
- 2.2 Étude de performance théorique . . . 6
- 2.3 Étude de performance pratique . . . 7

3 Version parallèle **7**
- 3.1 Parallélisme . . . 7
  - 3.1.1 Mémoire partagée : . . . 8
  - 3.1.2 Mémoire distribuée . . . 8
- 3.2 Problèmes de programmation parallèle . . . 10
- 3.3 Étude de performance théorique . . . 10
- 3.4 Étude de performance pratiques . . . 10
- 3.5 Architectures parallèles visées . . . 12
- 3.6 Machine parallèle RUCHE . . . 12

4 Conclusion **13**

Introduction

Un des problèmes majeurs rencontrés en analyse numérique est la capacité de résoudre un système d'équations linéaires de grande taille issu de la modélisation des problèmes rencontrés en physique, en mécanique, en chimie, en économie ou d'une façon générale les projets provenant de l'ingénierie. Il est connu que la discrétisation ou la linéarisation des équations non linéaires produit des systèmes d'équations linéaires de taille finie liée au nombre de points de discrétisation. La résolution d'un tel système lorsque sa taille est très grande est assez coûteuse en coût de temps, de calculs et en espace mémoire. La recherche actuelle en matière de résolution de systèmes linéaires consiste à réduire le nombre d'itérations, de calculs et du temps d'exécution des programmes. De nombreuses méthodes existent, et le choix d'une d'entre elles se fait selon la structure de la matrice (pleine, creuse, symétrique, définie positive,...).

les méthodes de résolution s'articulent autour de deux axes : Le premier occupe les méthodes directes, qui consistent à factoriser la matrice A en un produit de matrices faciles à inverser. En fait, les grands systèmes d'équations sont souvent creux, dans une telle situation une méthode directe effectue un grand nombre d'opérations redondantes. Les deux factorisations LU avec pivotage et QR de la matrice A sont les plus utilisées en raison de leur stabilité numérique. Le deuxième axe concerne Les méthodes itératives, qui consistent à construire une suite de solutions approximatives convergeantes vers la solution exacte, toute en partant d'une solution arbitraire.

Dernièrement des généralisations des méthodes classiques de type Krylov ont été proposées et développées pour résoudre les systèmes d'équations linéaires avec plusieurs secondes membres dont la méthode de Lanczos.

L'objectif principal de ce travail est la résolution numérique des systèmes linéaires de grand Le chapitre I sera consacré aux rappels de plusieurs définitions, théorèmes et résultats nécessaires qui seront utilisés lors de la réalisation de ce projet. Chapitre 2 sera consacré a l'algorithme associé et quelques unes de ses propriétés. chapitre 3 aura pour objectif de réaliser cette méthode en parallélisant l'algorithme séquentiel et récoltant les différentes mesures de performance. Et Enfin on conclura par la discussion sur les mesures et les différents résultats obtenus.

1 Problématique

1.1 Notations et définitions

Définition Soit A une matrice de $I_K n$ ($I_K = I_R$ ou C). La matrice A est dite :
— symmétriques si $A^T = A$,
— hermitiennes si $A^H = A$ et $\mathbb{K} = \mathbb{C}$,
— antisymétriques si $A^T = A$,
— antihermitienne si $A^H = A$,
— normale si $A^H A = AA^H$,
— unitaire si $A^H A = I$,
— diagonale si $a_i j = 0$ pour $j \neq i$,
— triangulaire supérieure si : $a_i j = 0$ pour $i > j$,
— triangulaire inférieure si $a_i j = 0$ pour $i < j$,
— tridiagonale : $a_i j = 0$ pour chaque couple i, j tel que $|ij| > 1$

1.2 Bases de Krylov

Les sous-espaces de Krylov sont utilisés dans de nombreux algorithmes numériques en algèbre linéaire pour trouver des solutions approchées à des problèmes de vecteurs propres avec des matrices de grande dimension.

Les méthodes itératives modernes, telles que l' algorithme d'Arnoldi, peuvent être utilisées pour trouver une (ou plusieurs) valeurs propres de grandes matrices creuses ou pour résoudre de grands systèmes d'équations linéaires. Ils essaient d'éviter les opérations matrice-matrice, mais utilisent plutôt les produits matrice-vecteur et travaillent avec les vecteurs résultants. Étant donné le vecteur b, on calcule Ab, puis on multiplie ce vecteur par A pour trouver $A^2 b$ etc. Tous les algorithmes qui fonctionnent de cette manière sont appelés méthodes de sous-espace de Krylov ; ce sont actuellement les méthodes les plus efficaces disponibles en algèbre linéaire numérique.

On cherche à résoudre le système d'équations linéaires suivant :

$Ax = b$

La matrice A est supposée inversible et de taille (m x m). De plus, on suppose que b est normé, (i.e., $b = 1$ Où, représente la norme euclidienne).

Le n-ième espace de Krylov pour ce problème est défini ainsi :

$\mathbb{K}_n = \text{Vect}\{b, Ab, A^2 b, \ldots, A^{n-1} b\}$

où Vect signifie le sous-espace vectoriel engendré par les vecteurs.

1.3 L'algorithme de Lanczos

En algèbre linéaire, l'algorithme de Lanczos est un algorithme itératif pour déterminer les valeurs et vecteurs propres d'une matrice carrée, ou la décomposition en valeurs singulières d'une matrice rectangulaire. il consiste à réduire une matrice A en une matrice tridiagonale T semblable à A et de construire les bases des sous-espaces de Krylov souhaitées.

L'intérêt principale des processus de Lanczos est qu'ils permettent de conserver une récurrence courte pour la construction des bases des sous espaces de Krylov.

En raison de leurs propriétés numériques efficaces, les processus de Lanczos sont maintenant introduits dans de nombreux algorithmes et utilisés dans plusieurs domaines, ils peuvent être également utilisés pour résoudre un système d'équations linéaires, ils aboutissent à des

méthodes itératives qui donnent la solution exacte en un nombre fini d'itérations inférieur ou égal à la dimension du système. La mise en œuvre de ces méthodes est effectuée par différents algorithmes.

algorithme de lanczos

Le processus de Lanczos symétrique consiste à construire une suite de vecteurs v_j par une formule de récurrence à trois termes. En effet, il suffit de disposer seulement des vecteurs v_{J-1} et $v_j$pour pouvoir calculer les v_{J+1} et le coût de calcul est indépendant du numéro de l'itération. Ceci explique pourquoi une relation seulement tri-récursive est effectuée dans le cas symétrique.

Soit A une matrice hermitienne, symétrique réelle de taille $n * m$, et si V est unitaire alors on obtient en sortie une matrice tridiagonale de taille $n * m$ tel que :
$AV_m = V_m T_m + \beta_{m+1} v_{m+1} e_m^T$
$T_m = V_m T A V_m$.

1.3.1 Etapes de l'algorithme :

— Soit V un vecteur arbitraire de norme euclidienne .$v_1 \in \mathbb{R}^n$
initialisation :

Posons $w_1' = Av_1$
Posons $\alpha_1 = w_1'^* v_1$
Posons $w_1 = w_1' - \alpha_1 v_1$

— soit :$j = 2, \ldots, m$
Si (aussi norme euclidienne).$\beta_j = \|w_{j-1}\|$ implique que ,$\beta_j \neq 0 v_j = w_{j-1}/\beta_j$
sinon,on choisi un vecteur arbitraire avec une norme euclidienne orthogonale à tout $v_1, \ldots, v_{j-1}$

— Nous obtenons :
$w_j' = Av_j$
$\alpha_j = w_j'^* v_j$
ainsi que la matrice tridiagonale

$$ {}_1T = \begin{pmatrix} \alpha_1 & \beta_2 & & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \beta_{m-1} & \\ & & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ 0 & & & & \beta_m & \alpha_m \end{pmatrix} $$

1.3.2 Pseudo Code méthode de Lanczos :

Algorithme 2.2 Processus de Lanczos symétrique

1. Choisir un vecteur initial v_1 tel que $\|v_1\|$=1,
2. Poser β_1=0, $v_0 \equiv 0$,
3. Pour $j = 1, 2, \cdots, m$ faire
4. $\quad w_j = Av_j - \beta_j v_{j-1}$,
5. $\quad \alpha_j = (w_j, v_j)$,
6. $\quad w_j = w_j - \alpha_j v_j$,
7. $\quad \beta_{j+1} = \|w_j\|_2$. si $\beta_{j+1} = 0$ arrêter.
8. $\quad v_{j+1} = \frac{w_j}{\beta_{j+1}}$,
9. Fin j.

Remarques

- Ces algorithmes risquent d'échouer et de s'arrêter sans converger, une division par zéro peut se produire. Dans ce cas, une division par zéro (appellée breakdown) peut se produire. En réalité, l'orthogonalité ou la biorthogonalité exacte de ces vecteurs n'est observée qu'au début des processus, à un certain point les vecteurs commencent à perdre rapidement leur orthogonalité global.
- Il existe en principe quatre manières d'écrire la procédure d'itération. Paige et d'autres travaux montrent que l'ordre des opérations ci-dessus est le plus stable numériquement.
- En pratique, le vecteur initial peut être considéré comme un autre argument de la procédure, avec et des indicateurs d'imprécision numérique étant inclus comme conditions supplémentaires de terminaison de boucle.$\mathrm{v}_1\beta_j = 0$.

- Sans compter la multiplication matrice-vecteur, chaque itération effectue des opérations arithmétiques. La multiplication matrice-vecteur peut être effectuée dans des opérations arithmétiques où est le nombre moyen d'éléments non nuls dans une ligne. La complexité totale est donc , ou si ; l'algorithme de Lanczos peut être très rapide pour les matrices creuses. Les schémas d'amélioration de la stabilité numérique sont généralement évalués en fonction de cette haute performance.
- Les vecteurs sont appelés vecteurs de Lanczos. le vecteur n'est pas utilisé après le calcul. Par conséquent, on peut utiliser le même stockage pour les trois. De même, si seule la matrice tridiagonale est recherchée, alors l'itération brute n'a pas besoin d'être calculé , bien que certains schémas pour améliorer la stabilité numérique en aurait besoin plus tard. Parfois, les vecteurs Lanczos suivants sont recalculés à partir du moment où cela est nécessaire.

2 Cas séquentiel

2.1 Description de l'algorithme proposé

L'algorithme de *lanczos* que nous avons implémenté prend en argument une matrice carrée $A \in R^{n*n}$, un vecteur initial aléatoire de taille n normalisé : c-a-d v tel que $||v|| = 1$, et un nombre m de valeurs propres qu'on souhaite approché. Puis en sortie, l'algorithme calcul la matrice de Ritz T_m de taille maximale $m \times m$ avec les coefficients α_i, $beta_j$ et la matrice V_m qui contient m vecteurs orthogonaux de taille n. Si au cours d'une itération m_0 inférieure à m on trouve un $\beta_j = 0$, alors l'algorithme s'arrête automatiquement.
Pour déterminer maintenant l'approximation des m valeurs propres, on a utilisé la bibliothèque $gsl/blas$ sur la matrice creuse Tm. En comparant ainsi les valeurs propres de la matrice A à celles de T_m, on remarque qu'on a une bonne approximation de la plus grande valeur propre de la matrice A.
Voici un exemple d'application de notre algorithme sur une petite matrice dense, symétrique :

```
Matrice A
9.000000  1.000000  -2.000000  1.000000
1.000000  8.000000  -3.000000  -2.000000
-2.000000  -3.000000  7.000000  -1.000000
1.000000  -2.000000  -1.000000  6.000000
---Version Sequentielle dans lanczos--
Matrice Tm 2,
3.579912  2.011318
2.011318  10.826790
Valeur propre Pour A : 12  9  6  3
Valeur propre Pour Tm =   11.3476  3.05911
Performance results:
  Time: 0.000056 s
```

Nous pouvons voir avec ces résultats qu'on a une approximation des valeurs propres extrêmes.

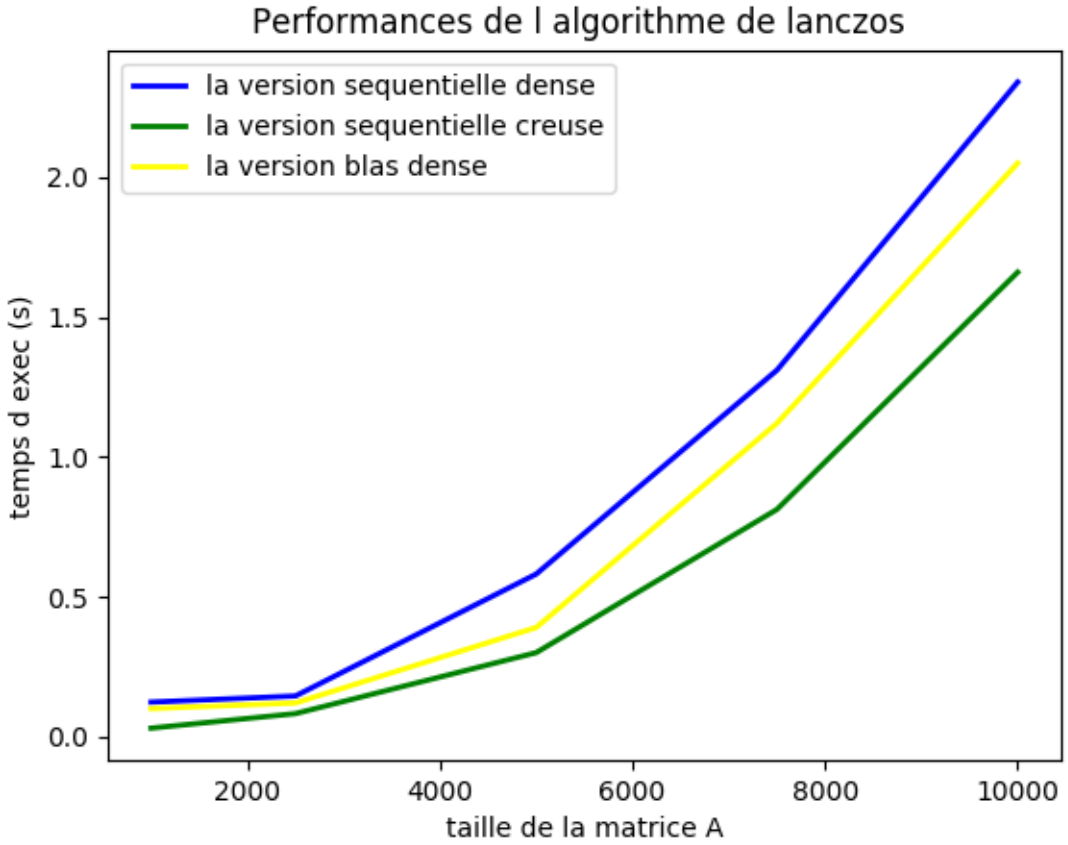
2.2 Étude de performance théorique

On voit que la globalité de calcul se fait dans la partie multiplication matrice-vecteur, alors si on considère pas cette phase, le reste de l'algorithme peut se faire en O(n) opérations arithmétique,et la partie multiplication matrice vecteur se fait en O($d \times n$) avec d le nombre moyen des éléments non nuls dans une ligne de la matrice, ainsi la complexité totale peut être estimée à O($d \times n \times m$) , cet algorithme peut donc être adapté pour les matrices creuses.
Pour l'algorithme de l'anczos, la plus grande performace qu'on peut avoir se trouve dans le produit matrice-vecteur. Donc plus on optimise le produit matrice-vecteur, plus on gagne en

performance. C'est pour cette raison l'utilisation de la bibliothèque *blas* nous donne des performances incroyables, ainsi que l'utilisation de la matrice creuse. Car pour la matrice creuse la complexité est de $O(nnz)$, où nnz est le nombre d'éléments non nuls dans la matrice.

2.3 Étude de performance pratique

Afin d'évaluer les performances, nous avons mesurer les performances dans un premier temps de l'algorithme Lanczos version séquentielle en utilisant la fonction clock_t() de la bibliothèque time.h en variant taille de la matrice A. Par la suite on a mesuré les performances de l'algorithme avec la version utilisant les routines blas précisément la bibliothèque gsl pour la deuxième phase, les résultats sont représentés par la figure suivante.



3 Version parallèle

3.1 Parallélisme

Dans le sens le plus simple, le calcul parallèle est l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème de calcul :

— Un problème est divisé en parties distinctes qui peuvent être résolues simultanément
— Chaque partie est ensuite décomposée en une série d'instructions
— Les instructions de chaque partie s'exécutent simultanément sur différents processeurs
— Un mécanisme global de contrôle / coordination est utilisé

La programmation parallèle fournit

— Plus de ressources CPU

— Plus de ressources mémoire
— La capacité de résoudre des problèmes qui n'étaient pas possibles avec le programme série
— La capacité de résoudre les problèmes plus rapidement

Deux approches sont envisageables :

3.1.1 Mémoire partagée :

— Utilisé par la plupart des machines
— Plusieurs cœurs (processeurs)
— Partager un espace mémoire global
— Les cœurs peuvent échanger / partager efficacement des données

Étant donné la nature de l'algorithme de *lanczos* qui est itérative (c'est à dire la valeur suivante obtenue en fonction de la valeur précédente), nous ne pouvons pas directement paralléliser la boucle à chaque étape car la boucle est fortement séquentielle. Par ailleurs le but de l'algorithme de *lanczos* est de déterminer une certaine quantité de valeur et vecteur propres, ainsi cette boucle n'est pas forcement destinée à être très grande, sauf dans le cas où l'on cherche à raffiner les résultats en visant une certaine précision, alors on itérera l'algorithme de *lanczos* jusqu'à l'obtention de la précision souhaitée.
Afin d'obtenir les performances, nous avons décider de cibler les fonctions définissant les opérations de matrices-vecteurs en appliquant le parallélisme des données. Pour cela nous avons utiliser deux outils : Les instructions SIMD (simple instruction multiple data) et la programmation mémoire partagée avec la bibliothèque OpenMP.

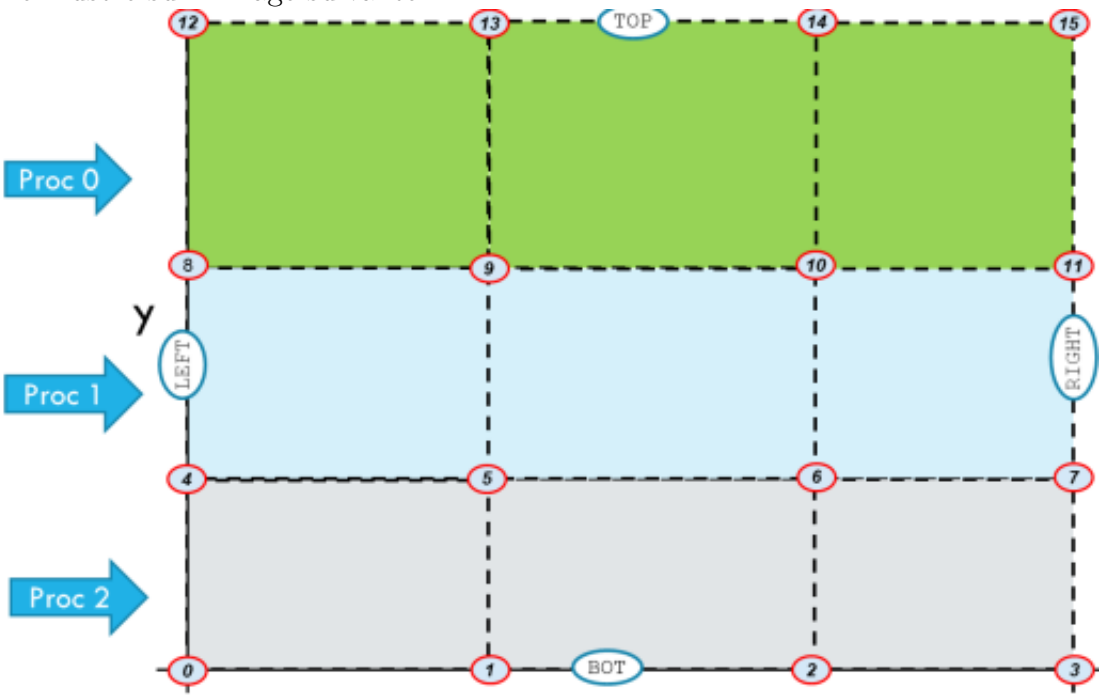
— **SIMD :** Les instruction simd, sont des instructions qui cibles les différents registres 128 bytes, 256 bytes ou 512 bytes selon si la machine est compatible SSE2,AVX/AVX2 ou AVX512. Avec, ces instructions, nous pouvons effectuer la même opération sur plusieurs données et bénéficier non seulement de la vitesse des registres, mais aussi de la capacité qu'a le processeurs de pré charger les données et de prédire les opérations suivantes. Il existe différentes manière d'utiliser les instructions SIMD : on peut soit utiliser la bibliothèque "intrinsic.h" et utiliser les fonctions prédéfinies pour la vectorisation, ou utiliser les flags/pragma pour le compilateur GCC afin qu'il puisse générer pour nous les différentes instructions SIMD pour notre machine. Nous avons finalement opté pour la deuxième méthode, en utilisant les flags suivants : *-O3*(Pour les optimisation du compilateur), *unroll-loops* (pour dérouler au maximum possible les boucles, *inline* (pour insérer les petites fonctions à l'endroit où elles sont appelées), *-march=native* (Pour dire au compilateur de générer des instructions qui cible exactement notre machine). Cependant pour que le compilateur génère au mieux ces instructions, nous avons veillez à mieux stocker notre matrice afin que l'accès aux données soit consécutif.
— **Pragma openmp :** Il s'agit d'utiliser les threads de la machine et d'exécuter différentes instructions en instantanées. Chaque thread va ainsi effectuer une partie de la boucle avec un minimum de dépendance entre les itérations. Pour ce faire, on a utilisé des pragma *#pragma omp parallel for* , sur les différentes boucles en faisant attention aux sections critiques comme dans les cas de réductions ou on utilise *#pragma omp parallel for reduction(signe :variable).*

3.1.2 Mémoire distribuée

— Collection de nœuds qui ont plusieurs cœurs
— Chaque nœud utilise sa propre mémoire locale
— Travaillez ensemble pour résoudre un problème
— Communiquez entre les nœuds et les cœurs via des messages

— Les nœuds sont mis en réseau ensemble

Dans notre cas le parallélisme en mémoire distribué est moins envisageable pour la version normale de l'algorithme de *lanczos*. Il y'a cependant une alternative qui consiste à subdiviser la matrice en plusieurs sous domaine et de repartir les tâches mpi sur chaque sous-domaine. Comme illustré sur l'image suivante :



Avec cette configuration, chaque processus charge la zone concernée de la matrice, mais le désavantage de cette décomposition avec l'algorithme de lanczos, est au niveau du calcul des coefficients de la matrice de Ritz où on aura besoin de chaque résultat partiel des différents sous-domaines. Ce qui implique la synchronisation des différents processus et augmente les transferts entre processus. Ce qui au finale dégrade les performances. On peut néanmoins utiliser la mémoire distribuer pour trouver les valeurs propres à une précision donnée. En suivant l'algorithme dans le document suivant [4] :

Iterative Lanczos algorithm

Step 1. Choice of m.
Step 2. Choice of initial vector x.
Step 3. Normalization of x : $y_0 = x/\parallel x \parallel$ and $\beta_{-1} = 0$.
Step 4. Computation of T_m matrix elements.

- For $j = 0, m-2$, do

$$\alpha_j = (Ay_j, y_j)$$
$$y'_{j+1} = Ay_j - \alpha_j y_j - \beta_{j-1} y_{j-1}$$
$$\beta_j = \parallel y'_{j+1} \parallel$$
$$y_{j+1} = y'_{j+1}/\beta_j$$

- End for j
- $\alpha_{m-1} = (Ay_{m-1}, y_{m-1})$

Step 5. Computation of the eigenvalues of T_m matrix.
Step 6. Computation of the eigenvectors $w_i^{(m)}$ of T_m matrix.
Step 7. Computation of the Ritz vectors $v_i^{(m)}$ by (73).
Step 8. If ($\max_{1 \le i \le r} \parallel (A - \lambda_i^{(m)} I) v_i^{(m)} \parallel \le$ *requested precision*) then stop, otherwise with a new initial vector go to step 3.

Avec cet algorithme, chaque processus exécute le code avec des points de démarrage différents. On arrête l'exécution dès lors qu'un processus trouve la bonne précision.

Il existe d'autres versions de l'algorithme de *lanczos*, mais qui demande l'utilisation d'autre méthodes comme la factorisation.
Nous avons pris la version classique de l'algorithme de *lanczos*.

3.2 Problèmes de programmation parallèle

L'objectif est de réduire le temps d'exécution
- Temps de calcul
- Minimiser la communication
- Temps d'inactivité - attente des données d'autres processeurs
- Temps de communication - temps nécessaire aux processeurs pour envoyer et recevoir des messages
- L'équilibrage de charge
- Répartir le travail de manière égale entre les processeurs disponibles
- Réduire le nombre de messages passés
- Réduire la quantité de données transmises dans les messages

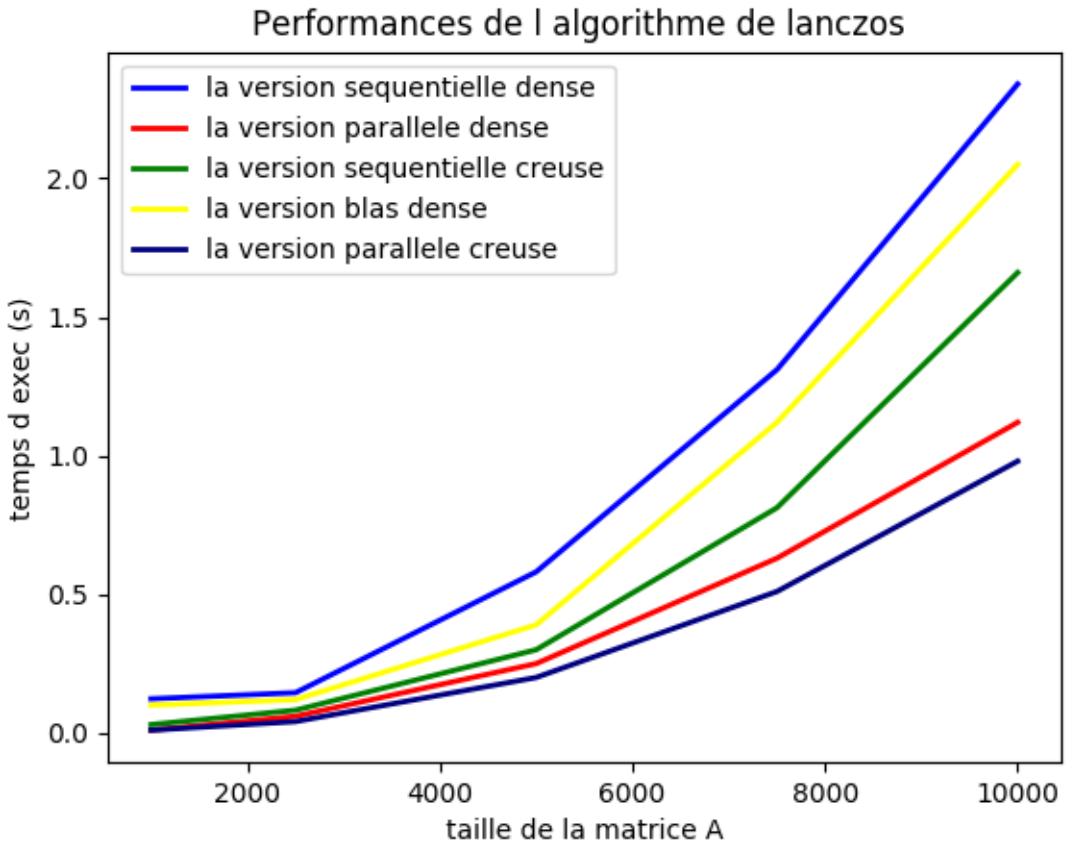
3.3 Étude de performance théorique

Afin d'évaluer les performances dans le cas parallèle, nous pouvons noter que la complexité maximale qu'on peut avoir dans la version classique est lorsque la taille de la matrice de Ritz $m = n$ (où n est la taille de la matrice initiale A) est $O(n^3)$. La plus grande partie de cette complexité est induite par le produit matrice vecteur. Ainsi dans un premier temps, avec les instructions SIMD (avx/avx2), la complexité du produit matrice vecteur passe de $O(n^2)$ à $O(n * n/4)$, puisque les instructions avx/avx2 peuvent charger 4 "*double*" et les additionner en un instant au lieu d'un seul "*double*" dans le cas séquentiel normal. En plus, lorsqu'on lance l'algorithme version parallèle *OpenMP*, la complexité serait équivalente à $O(n/k * n/4)$ avec k raisonnable, on verra plus tard qu'on se stabilise à partir de $k = 20$, comme le montre notre courbe de scalabilité ci-desous.

3.4 Étude de performance pratiques

Afin d'évaluer les performances du code parallèle, nous avons fixé la variable
OMP_NUM_THREAS=4. On obtient les résultats suivants pour les différentes tailles de la matrice. On a pris comme exemple la matrice tri-diagonale suivante : $a = 11111111$ et $b = 9090909$

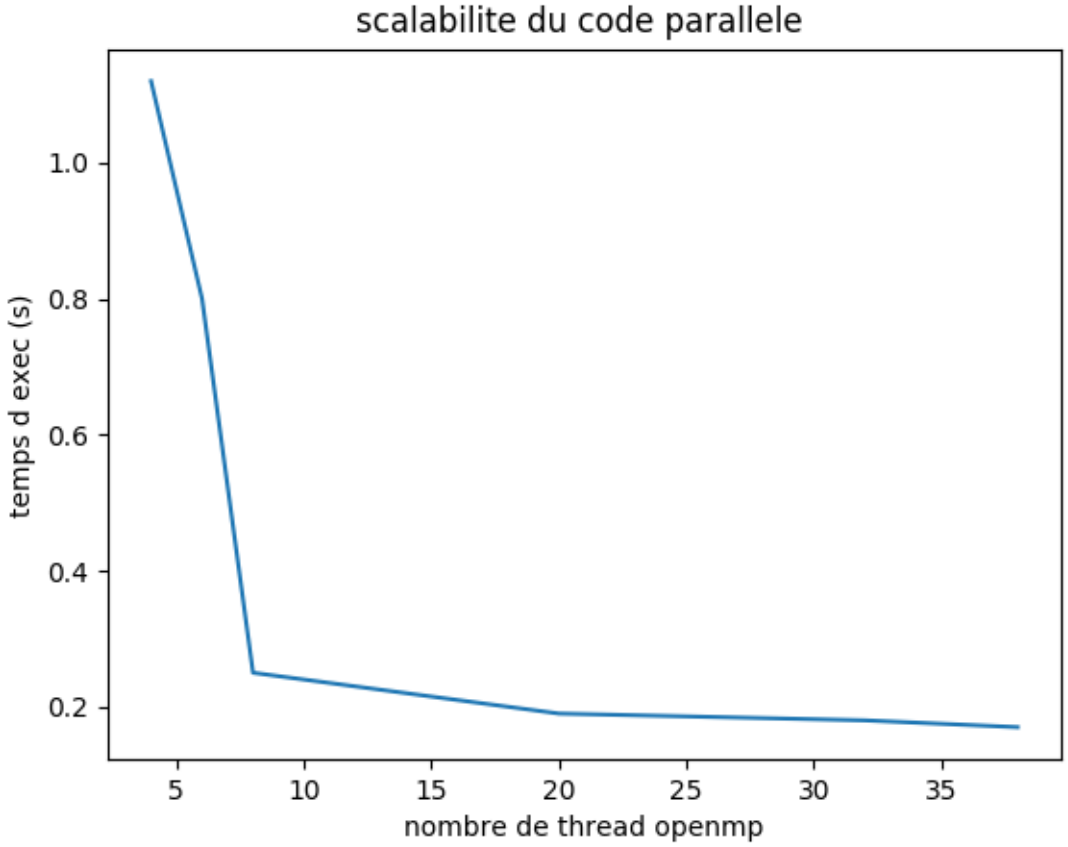
$$A_9 = \begin{pmatrix} a & b & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ b & a & b & 0 & \cdots & \cdots & \cdots & \cdots & \vdots \\ 0 & b & a & b & 0 & \cdots & \cdots & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & b & a & b & 0 & \cdots & 0 \\ \vdots & \cdots & \cdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \cdots & \cdots & \cdots & 0 & b & a & b & 0 \\ \vdots & \cdots & \cdots & \cdots & \cdots & 0 & b & a & b \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & b & a \end{pmatrix}$$



— En appliquant le modèle Openmp on voit beaucoup de différences au niveau du temps d'exécution, commençant par le fait que la version parallèle est meilleure que la version blas , mais aussi la réduction importante entre la version séquentielle et la version parallèle ou on est a 3 fois mois de temps d'exécution.

— On a fais une étude de scalabilité afin de savoir si on a choisi le modèle de parallélisme adéquat ou on s'intéresse principalement à l'efficacité du programme,Le problème reste

fixe alors que le nombre de thread openmp augmente (ce qui veut dire que le nombre de coeurs augmente aussi).



— Pour notre cas on remarque que la durée d'exécution diminue fortement lorsqu'on utilise 8 threads, représentant le nombre référence de cœurs CPU pour notre programme afin qu'il soit efficace.
— En comparant ce nombre à l'intervalle de thread choisi, on déduit qu'on a une bonne scalabilité, justifiant une bonne utilisation du modèle de parallélisation utilisé.

3.5 Architectures parallèles visées

Dans notre implémentation parallèle, l'architecture ciblée est la mémoire partagée, en appliquant le parallélisme des données qui utilise différents coeurs de calcul afin d'améliorer les performances de notre code.

Le principe repose sur la parallélisation des boucles de multiplication matrice-matrice ou matrice-vecteur qui représentent la majorité de notre code.

3.6 Machine parallèle RUCHE

Nous avons exécuté notre code parallèle sur la machine RUCHE qui dispose des caractéristiques suivantes :

— 2 Threads par cœur.
— 80 unités de traitement disponibles pour le processus en cours.
— 2 noeuds numa.
— Architecture x86_64 à mémoire partagée.

Pour charger notre code et le lancer sur RUCHE on a suivi la documentation disponible sur https://mesocentre.pages.centralesupelec.fr/user_doc/

Afin d'exécuter les différentes version implémentées, un fichier README.md est inclus avec le code source.

4 Conclusion

Il existe plusieurs algorithmes Lanczos (itératif, utilisant la factorisation..etc) il faut en choisir la méthodes la plus adaptée pour chaque situation : pour les matrices larges et creuses, les matrices dont le nombre de lignes est très grandes par rapport aux nombre de colonnes ou l'inverse, etc.

Les mesures de performances nous montrent qu'il faut privilégier les fonctions des librairies standard de calcul numérique qui sont déjà optimisées au niveau mathématique et algorithmique.

Il faut choisir le modèle de programmation (soit parallélisme de données ou parallélisme de taches) en adéquation avec la ou les architectures parallèles visée.

Annexes

https://www.irisa.fr\T1\guilsinglrightMSI\T1\guilsinglrightbeyrouth-dea-0204

Bibliographie

[1] https://stringfixer.com/fr/Lanczos_algorithm
[2] https://fr.wikipedia.org/wiki/Sous-espace_de_Krylov
[3] https://thesesenafrique.imist.ma/handle/123456789/1714
[4] The Pad6-Rayleigh-Ritz method for solving large hermitian eigenproblems, NAHID EMAD