

**MATIERE :
METHODE ET PROGRAMMATION NUMERIQUES AVANCEES**

METHODE DES PUISSANCES ITEREES

Responsable de formation
Mme. Nahid Imad

Etudiante
AICHA LAMMAMRA

2020/2021

Table des matières

I.	Introduction	3
II.	Problématique.....	3
III.	Méthode de la puissance.....	3
1.	Principe.....	3
2.	Algorithme.....	3
IV.	Méthode de la déflation	4
V.	Cas séquentiel.....	5
1.	Description de l'algorithme proposée.....	5
2.	Validation du code séquentiel.....	6
3.	EDP théorique	7
A.	Complexité en temps :	7
B.	Complexité en mémoire :	7
4.	EDP pratique	7
A.	Comparaison et interprétation	8
5.	Conclusion « local »	9
VI.	Cas parallèle	9
1.	Description	9
2.	EDP pratique	10
	Accélération.....	11
VII.	Conclusion générale	12

Table des figures

Figure 1.	Implémentation séquentielle	5
Figure 2.	résultats obtenus à l'aide du site internet	6
Figure 3.	Résultats d'exécution	7
Figure 4.	Temps d'exécution de la méthode de puissance itérée en fonction de la taille de la matrice.	8
Figure 5.	Temps d'exécution de la méthode de déflation en fonction de la taille de la matrice	8
Figure 6.	Temps d'exécution en fonction de taille de la matrice	10
Figure 7.	Temps d'exécution en fonction du nombre de threads	10
Figure 8.	Accélération d'un programme parallèle	11

I. Introduction

Dans plusieurs problèmes physiques ou mathématiques, la résolution des problèmes telle que la résolution de grands systèmes linéaires $Ax = b$, où A est une matrice réelle carrée, se souvent nécessaire à trouver les valeurs propres d'une matrice et les vecteurs propres associés.

Si on voulait définir les vecteurs propres d'une application linéaire, on dirait que ce sont les vecteurs correspondants aux axes privilégiés selon lesquels l'application se comporte comme une dilatation, multipliant les vecteurs par une même constante. Ce rapport de dilatation est appelé valeur propre.

II. Problématique

Le calcul des valeurs propres et des vecteurs associés par la méthode connue nécessite le calcul du déterminant de la matrice, soit un calcul très lourd en opération, lorsque la matrice est de dimension élevée. C'est pourquoi, on s'intéresse aux méthodes itératives qui consistent à calculer à chaque itération une correction de la solution approchée à l'aide d'un vecteur résidu.

L'utilisation de ses méthodes pour des matrices de très grandes tailles nécessite le passage à une exécution sur des machines parallèles et par conséquent l'adaptation des algorithmes à l'architecture de calcul.

Dans le cadre du module MPNA, je m'intéresse à la méthode des puissances itérée, qui permet d'obtenir le vecteur propre associé à plus grande valeur propre en module d'une matrice carrée.

III. Méthode de la puissance

1. Principe

La méthode de la puissance itérée est une méthode itérative qu'approche la plus grande valeur propre d'une matrice $A \in M_n(\mathbb{R})$ et de son vecteur propre associée. Sous certaines conditions :

- La matrice A doit être symétriques définie positive.
- La matrice A doit posséder n valeurs propres distinctes $\lambda \in \mathbb{R}$ telles que :

$$\forall i \in \{1, \dots, n\}, |\lambda_1| > \dots > |\lambda_n|$$

Soient $u_1 \dots u_n$ est vecteurs propres associés.

2. Algorithme

On part d'un vecteur $x^{(0)}$ normalisé $x^{(0)} = \frac{x^{(0)}}{\|x^{(0)}\|}$.

Tant que le nombre d'itération maximal n'est pas atteint et que la différence des deux dernières valeurs approchées de λ_n est supérieure à la précision souhaitée, on calcule :

$$x^{(k+1)} = A \frac{x^{(k+1)}}{\|x^{(k+1)}\|}$$

$$\lambda = \langle x^{(k+1)}, x^{(k)} \rangle$$

IV. Méthode de la déflation

Considérons la matrice A utilisée dans la partie précédente. On applique une première fois la méthode de la puissance itérée sur cette matrice. Ainsi on obtient la valeur propre la plus grande en module.

On suppose qu'il s'agit de la valeur propre λ_n et du vecteur propre u_n .

A présent on veut calculer la deuxième plus grande valeur propre et son vecteur propre associée, idée consiste donc à créer une matrice B tel que :

$$B = A - \lambda_n u_n u_n^t$$

Puis on applique la méthode de la puissance itérée à cette nouvelle matrice, on obtient la deuxième plus grande valeur propre et son vecteur propre associé. Ensuite, pour obtenir toutes les autres valeurs propres il suffit de réitérer ce procédé en recréant à chaque fois la matrice B.

V. Cas séquentiel

1. Description de l'algorithme proposée

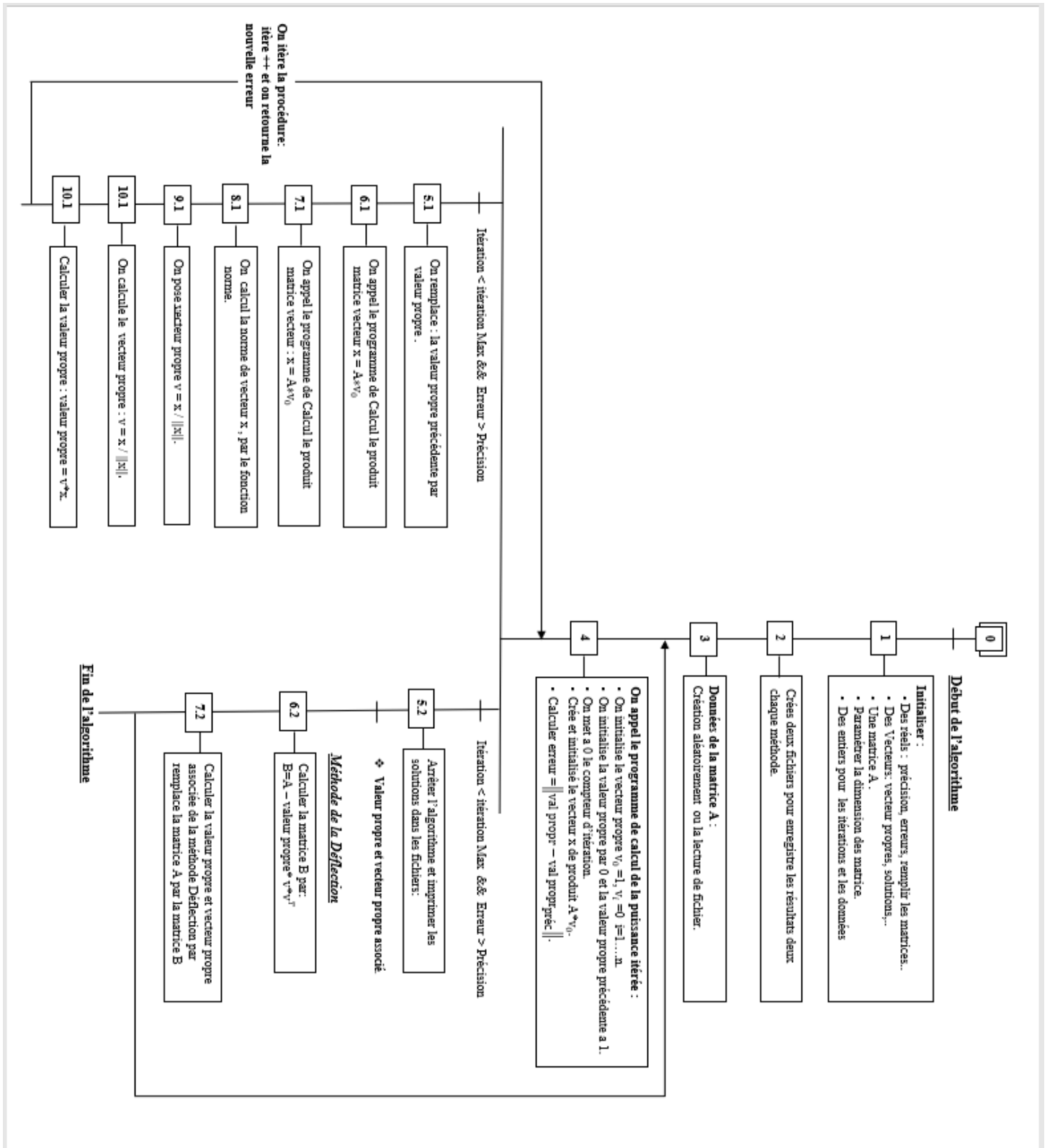


Figure 1. Implémentation séquentielle

À partir du diagramme précédent, on a implémenté la version séquentielle suivant 4 fonctions principales :

❖ **powrIter power_method (double **A, int n):**

Permet de trouver le vecteur propre associé à la plus grande valeur propre de la matrice A de taille (n*n) en appliquant la méthode de la puissance itérée.

❖ **powrIter deflation_method(double **A, double value, double* vector, int n):**

Permet de trouver le vecteur propre associer à la deuxième plus grande valeur propre de B en appliquant Méthode de la déflation.

❖ **double *prod_mtrix_vector(double **matrix,double *vector,int n):**

Permet faire le produit de la matrice A de taille (n*n) et le vecteur propre de taille n et stocker le résultat dans le vecteur x.

❖ **double prod_scalaire (double *vector1,double *vector2,int n):**

Permet faire le produit deux vecteur de taille n et retournera le résultat scalaire.

2. Validation du code séquentiel

Pour valider les résultats, j'ai utilisé une matrice de taille (4 *4) et j'ai fixé une tolérance à $1.e^{-7}$ et t j'ai comparé nos résultats avec les résultats obtenus à l'aide du site internet <https://www.dcode.fr> (figure 2). Je constate que j'ai obtenu les mêmes résultats ce qui valide mon algorithme (figure.3).

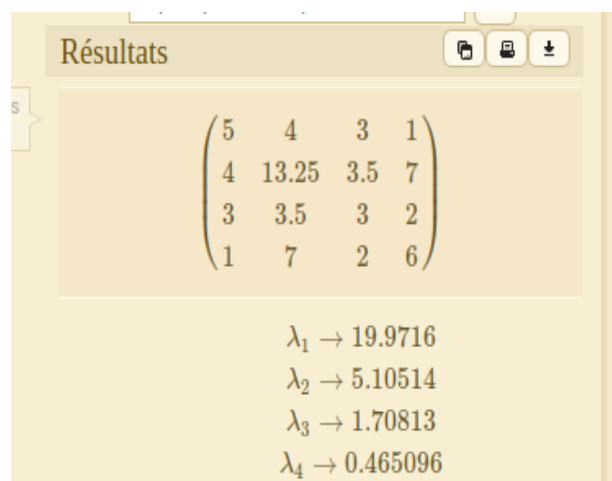
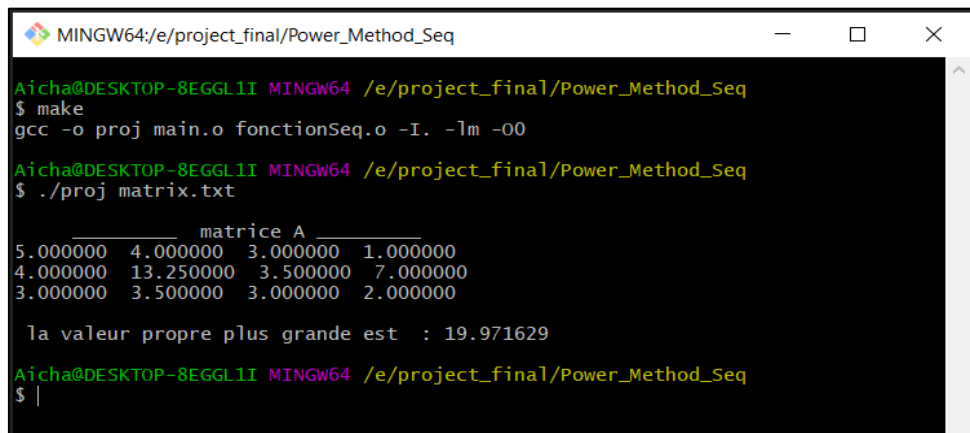


Figure 2. résultats obtenus à l'aide du site internet



```
MINGW64:/e/project_final/Power_Method_Seq
Aicha@DESKTOP-8EGGL1I MINGW64 /e/project_final/Power_Method_Seq
$ make
gcc -o proj main.o fonctionSeq.o -I. -lm -O0
Aicha@DESKTOP-8EGGL1I MINGW64 /e/project_final/Power_Method_Seq
$ ./proj matrix.txt

      matrice A
5.000000  4.000000  3.000000  1.000000
4.000000 13.250000  3.500000  7.000000
3.000000  3.500000  3.000000  2.000000

la valeur propre plus grande est : 19.971629
Aicha@DESKTOP-8EGGL1I MINGW64 /e/project_final/Power_Method_Seq
$ |
```

Figure 3. Résultats d'exécution

3. EDP théorique

A. Complexité en temps :

Pour chaque **itération** dans la méthode puissance itérée, je dois effectuer les opérations suivantes :

- Multiplication de la matrice A et le vecteur propre : $O(n^2)$.
- Multiplication scalaire : $O(n)$.
- Norme : $O(n)$.
- Division scalaire : $O(n)$

B. Complexité en mémoire :

- Matrice A : $O(n^2)$.
- Vecteur propre v : $O(n)$.
- Vecteur x ($x = A*v$) : $O(n)$.
- Division scalaire : $O(n)$.

4. EDP pratique

Evaluation de performance en terme du temps d'exécution de deux méthodes (puissance itérée, déflation) de calcul avec l'augmentation de la taille des données (matrice A), à l'aide de la bibliothèque « <time.h> ».

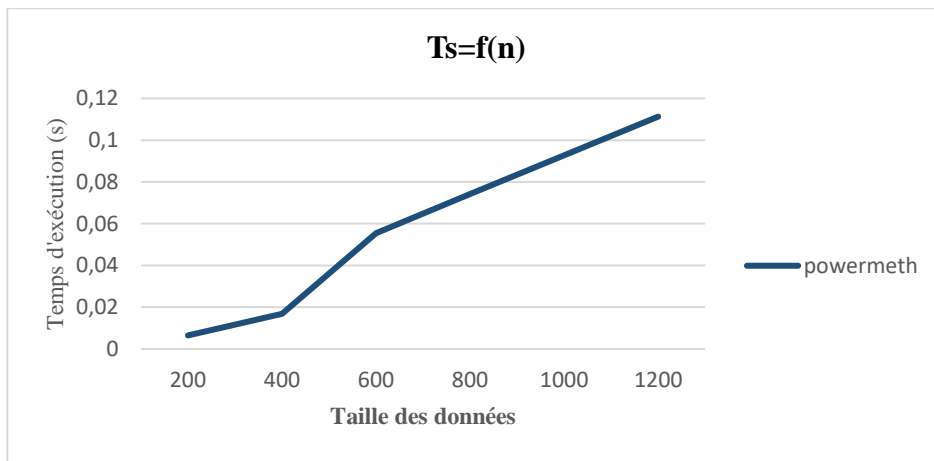


Figure 4. Temps d'exécution de la méthode de puissance itérée en fonction de la taille de la matrice.

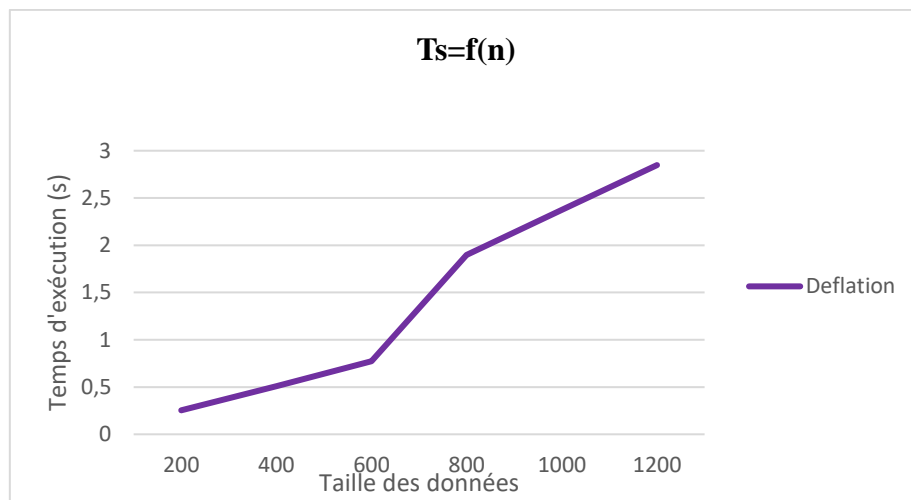


Figure 5. Temps d'exécution de la méthode de déflation en fonction de la taille de la matrice

A. Comparaison et interprétation

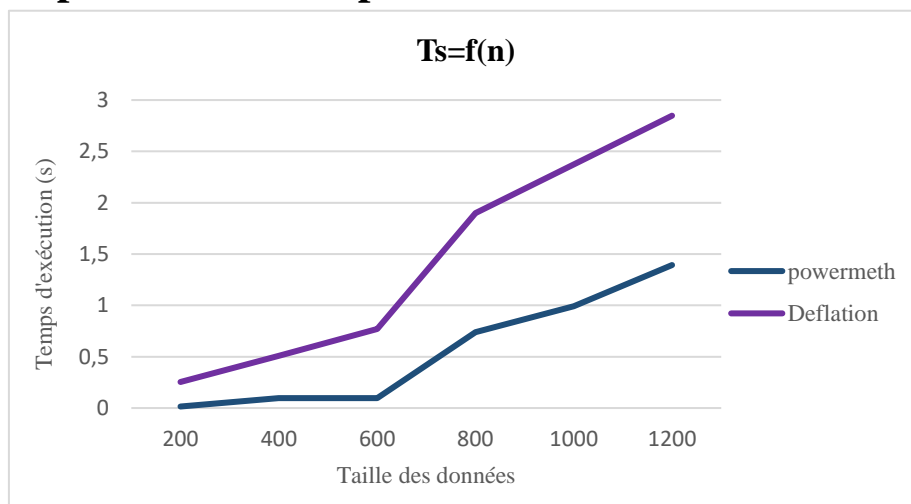


Figure 6. Comparaison des temps d'exécution pour les deux méthodes

Je remarque l'évolution du temps de calcul avec l'augmentation de la taille de la matrice A, Aussi je peux remarquer qu'il y a une forte augmentation du temps de calcul puisqu'à des tailles (1200*1200), je commence déjà à avoir des temps de calcul aux alentours de 20s pour la méthode déflation et plus d'une demi-minute pour la méthode puissance itérée.

5. Conclusion « local »

Je peux conclure que la taille de la matrice influence beaucoup sur les performances du programme.

VI. Cas parallèle

1. Description

Afin de passer à la version parallèle on doit tout d'abord connaître les fonctions les plus coûteuses en termes de calcul et cela par la mesure de temps d'exécution de ces dernières comme représenté dans le tableau ci-dessous.

Tableau 1. Temps d'exécution des différentes fonctions dans la fonction power_method

Taille des données (600*600)	Méthode de puissance itérée	Mrod_mtxrx_vector	Dev_vecteur_scal	Prod_scalaire
Nombre d'itération	80			
Temps d'exécution (ms)	0.41	0.38	0.02	0.008
Temps d'exécution total	0.41*80 = 33.49	92% du temps d'exécution de la fonction puissance itérée.	5% du temps d'exécution de la fonction puissance itérée	2% du temps d'exécution de la fonction puissance itérée

Comme vous pouvez le voir, ces trois fonctions représentent environ 99% de tous les travaux du programme. Dans ce qui suit, je me focaliserai sur la parallélisation de ces trois fonctions. Le profilage a été réalisé avec de taille 600*600.

Concernant la partie parallèle de la méthode, j'ai essayé d'implémenter une architecture de mémoire partagée, cette implémentation n'utilise que des threads (OpenMp), elle est basée principalement sur l'implémentation séquentielle que j'avais présentée précédemment. La seule différence c'est que les boucles principales ont été adaptées pour tirer bénéfice du multithreading, et augmenter ainsi la performance de mon programme.

2. EDP pratique

J'ai réalisé mes tests de performances sur une machine 1 Nœud de 16 cœurs.

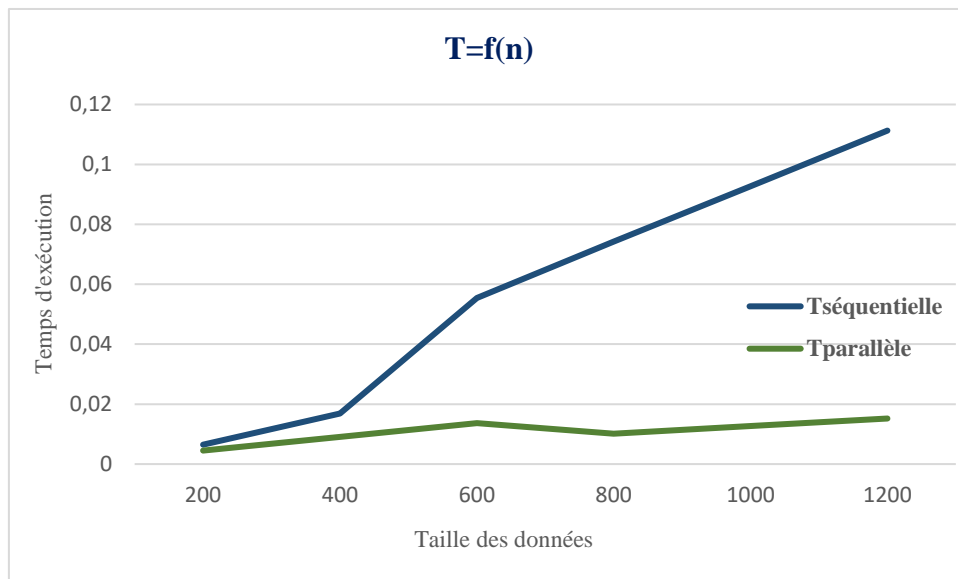


Figure 6. Temps d'exécution en fonction de taille de la matrice

La figure 6 représente les résultats du temps d'exécution de deux versions en variant la taille de la matrice (matrice a été remplie aléatoirement). Je remarque que la version parallèle est rapide par rapport à la version séquentielle qu'on a réalisé, et plus la taille de la matrice augmente plus elle va plus vite.

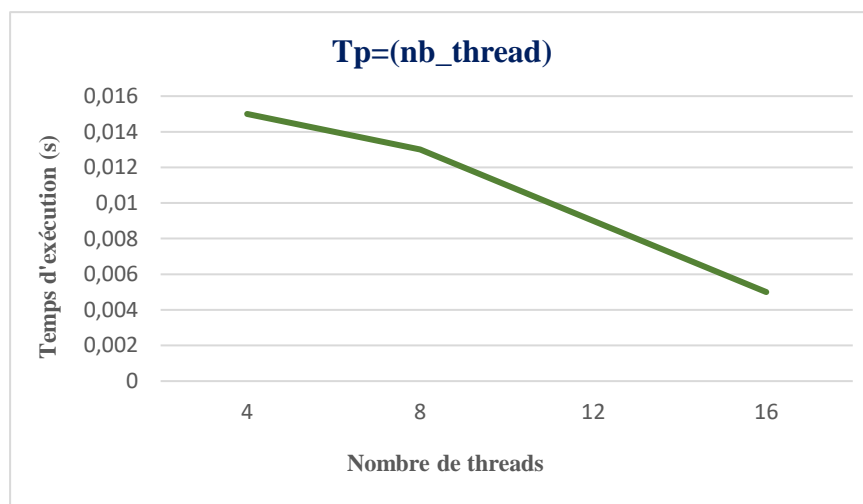


Figure 7. Temps d'exécution en fonction du nombre de threads

La figure 7 représenté les résultats du temps d'exécution d'une matrice de taille (600x600) en variant le nombre de Threads. On remarque un temps d'exécution qui décroît significativement ce qui explique que la version parallèle est rapide par rapport à la version séquentielle qu'on a réalisé.

Accélération

L'accélération d'un programme parallèle (ou speed up) représente le gain en rapidité d'exécution obtenu par son exécution sur plusieurs threads.

On la mesure par le rapport entre le temps d'exécution du programme séquentiel et le temps d'exécution sur p threads.

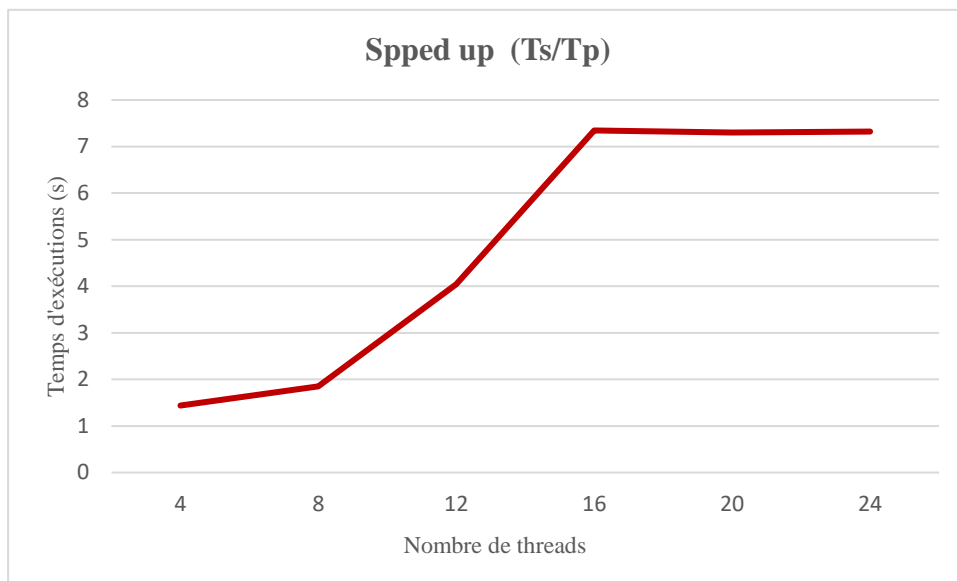


Figure 8. Accélération d'un programme parallèle

D'après la courbe du speedup j'ai remarqué qu'il y'a un gain très important au début, cette augmentation se stabilise à partir de 16 cœurs de calculs, un facteur d'accélération environ 7 est obtenu.

Ce graphique montre également que le parallélisme semble être scalable. Autrement dit, plus il y a de ressources parallèles disponibles, plus l'accélération est rapide.

D'après la loi d'Amdhal j'ai pu paralléliser jusqu'à 85% du code séquentiel.

Conclusion « local »

La version parallèle m'a permis d'avoir des temps d'exécutions réduit et très efficace pour des problèmes de petites tailles, j'ai alors décidé de continuer le travail par implémenté la deuxième version parallèle pour des problèmes de taille plus grande.

VII. Conclusion générale

Les expérimentations aboutissent au fait que la méthode converge pour seulement une partie des matrices ce qui était attendu compte-tenu de la structure de la méthode, le côté expérimental a donc corroboré le côté théorique. Par ailleurs, la méthode de départ est largement optimisable et il est paraît assez probable qu'une version modifiée de la méthode classique pourrait être particulièrement intéressante dans un contexte de Calcul Haute Performance d'autant que ce domaine se voit faire face à un nombre important de problèmes dont il est question ici.