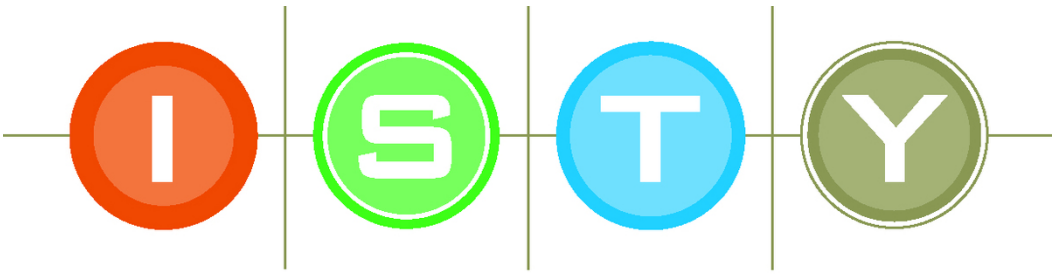
Rapport de projet

Programmation parrallèle et distribuée

Méthodes de la puissance itérée



INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

IATIC 4 - 2021/2022

Lucas DESHAYES & Clément ROBIN

Tables des matières

I - Introduction

1) Présentation du sujet

L'objectif du projet est d'implémenter la méthode de la puissance itérée pour calculer la plus grande valeur propre λ1 d'une matrice A et son vecteur propres associé u1.

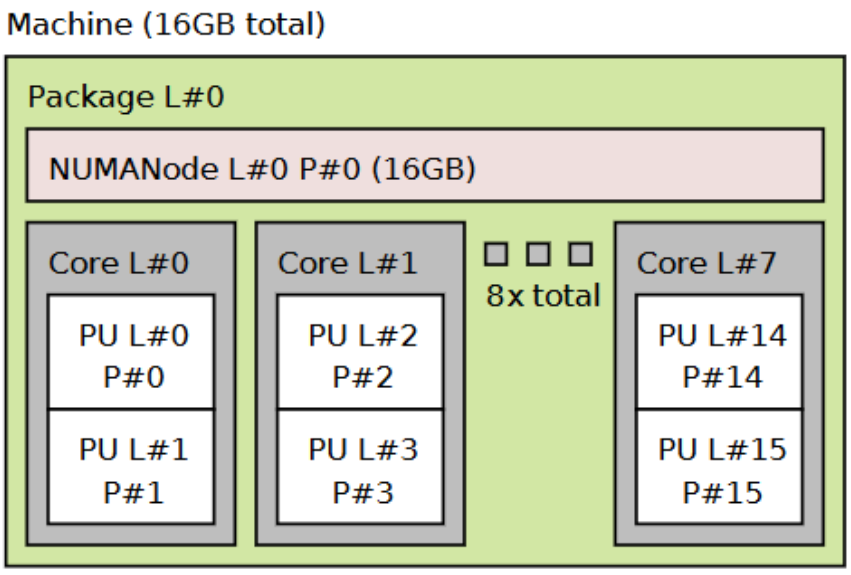
Plus particulièrement, il s'agit d'implémenter un algorithme en c, qui utilise la méthode de la programmation parallèle afin de rendre l'algorithme plus efficace en termes de temps d'exécution.
L'architecture parallèle visée est à mémoire partagée et l'interface de programmation pour le calcul parallèle sur ce type d'architecture est OpenMP (Open Multi-Processing)

2) Présentation de notre machine de test

Nous avons réalisé tous les tests et mesure sur un ordinateur dont dont voici la configuration du cpu :

Architecture	x86_64
Os	Windows Subsystem for Linux (WSL virtualisation de linux)
Processeur	AMD Ryzen 7 2700 Eight-Core Processor
Core	8
Thread(s) per core	2
CPU MHz	3200.000
Flags	ht (hyperthreading)

Voici également le résultat de la commande : lstopo



De plus le résultat de la commande : cat /proc/cpuinfo | grep -E "processor|core id"

core id : 0
processor : 0
processor : 1

core id : 1
processor : 2
processor : 3

core id : 2
processor : 4
processor : 5

core id : 3
processor : 6
processor : 7

core id : 4
processor : 8
processor : 9

core id : 5
processor : 10
processor : 11

core id : 6
processor : 12
processor : 13

core id : 7
processor : 14
processor : 15

Comme on peut le voir avec les résultats des cmd ci-dessus, notre machine de test possède 8 cœurs physiques {0,2,4,6,8,10,12,14} et 8 cœurs logiques {1,3,5,7,9,11,13,15}.
Lors de l'exécution nous pourrons donc assigner les threads en priorité sur les cœurs physiques afin d'optimiser notre programme et réduire le temps de l'exécution.

II - Méthode de la puissance itérée

1) Présentation de l'algorithme

Notre problème étant de calculer la plus grande valeur propre λ1 d'une matrice A et son vecteur propres associé u1. Pour le résoudre nous utilisons la méthode de la puissance itérée et l'algorithme suivant :

Algorithme de la méthode des puissances

Input: A, v, n
Result: λ_1
initialization : $v_1 = \frac{v}{\|v\|_\infty}$
for $k = 1, 2, \cdots$ *jusqu'à la convergence* **do**
  $v_k \leftarrow \frac{1}{\alpha_k} A v_{k-1}$
  $\alpha_k \leftarrow argmax_{i=1,\cdots,n} |(A v_{k-1})_i|$
end
$\lambda_1 \leftarrow \alpha_k$

Cet algorithme prend en entrée une matrice A, un vecteur v et un entier n. La valeur de retour de l'algorithme correspond à la plus grande valeur propre de la matrice A.
Nous avons décidé de couper l'algorithme en 3 parties. La première correspond à la création du vecteur Avk-1.
Dans la deuxième partie nous calculons la valeur de vk avec le calcul : 1/ak * Avk-1. Et dans la troisième partie nous calculons la valeur propre avec le calcul : argmax xi(|Avk-1|).
De plus, dans la 3ème partie nous calculons la convergence du programme afin de savoir quand il doit s'arrêter.

2) Implémentation et parallélisation

Notre objectif a été d'implémenter l'algorithme ci dessus en c et de le paralléliser avec les directives OpenMP. Pour l'implémenter nous avons tout d'abord créé une structure matrice afin de manipuler aisément les matrices dans notre programme.

De plus, nous avons parallélisé les 3 parties de l'algorithme. En effet sur la première partie, nous créons Avk-1 en faisant le produit : Matrice A * Vecteur Vk-1 dans une double boucle for qui parcourt notre matrice. Dans la deuxième partie nous calculons le vecteur vk dans une boucle for et dans la dernière partie nous calculons la valeur propre également dans une boucle for. Nous avons donc 3 boucles à paralléliser.

De plus nous ne pouvons pas paralléliser la boucle de convergence car en effet les actions inclus dedans ne sont pas indépendantes d'une itération à l'autre.

Nous avons donc choisis de paralléliser les boucles for de la manière suivante :

```
#pragma omp parallel for schedule(static,n/num_th) num_threads(num_th)
```

omp parallel for : nous permet de créer une nouvelle région parallèle et de paralléliser la prochaine boucle for.

schedule(static,n/num_th) : nous permet de séparer la boucle for pour les threads qui vont être créés dans la région parallèle. Nous avons choisi d'utiliser schedule static et la taille de paquet de n/num_th pour découper le partage entre les threads. Car en effet ici, notre matrice comporte des données de même taille où que l'on se trouve. Donc nous avons choisi cette méthode pour découper la boucle for.

num_threads(num_th) : nous permet de déclarer le nombre de threads souhaité lors de notre région parallèle.

Ici dans cet algorithme, dans les boucles for les opérations utilisées sont des affections et des calculs mais avec aucune dépendance entre eux, donc nous avons pu paralléliser les boucles simplement.

Par ailleurs, nous avons utilisé la directive critical lors d'un accès de concurrence lorsque l'on affecte la valeur propre.

III - Analyse des performances du programme

1) Calcul prévisionnel

A) Complexité temps

O(convergence) * (O(n*n) + O(n) + O(n)) = O(convergence) * $O(n^2)$

1 2 3

B) Complexité mémoire

La valeur propre est le résultat de l'algorithme, donc la complexité en mémoire est O(1)

C) Facteur d'accélération

g = (k * n) / (k - 1 + n)

D) Débit

D(p) = Charge de travail pour une taille n / Temps d'exécution

E) Débit asymptotique

Pour obtenir le débit asymptotique, on détermine la limite du débit lorsqu'on tend vers l'infini.

F) N max

A l'aide du logiciel SiSoftware Sandra nous avons pu calculer le Rpeak du cpu qui est de 160,36 GFlops.

2) Tests et résultats

Lors de nos tests réalisés sur le programme nous avons utilisé les export des variables d'environnement suivant afin d'optimiser notre code en parallèle:

OMP_PLACES="{0,2,4,6,8,10,12,14,1,3,5,7,9,11,13,15}", pour assigner en priorité les threads sur les cœurs physiques.
OMP_PROC_BIND=true, pour empêcher que les threads se déplace sur les cœurs lors de l'exécution.

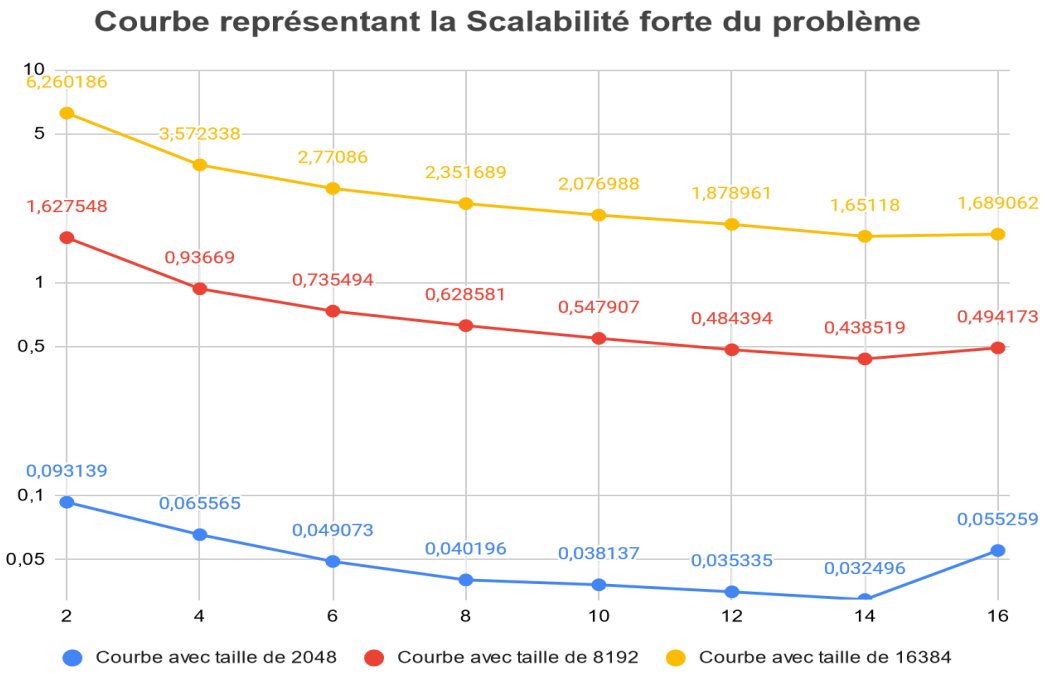
En programmation parallèle, on définit la scalabilité comme étant la capacité que possède un programme à utiliser des ressources de calcul additionnelles.

Si on veut effectuer plus rapidement les mêmes simulations, alors il est question de la scalabilité forte.
Et si on veut simuler des modèles plus grands ou plus détaillés sans augmenter la durée d'exécution, il s'agit de la scalabilité faible.

Voyons à présent comment se comporte notre programme en calculant la scalabilité forte et faible.

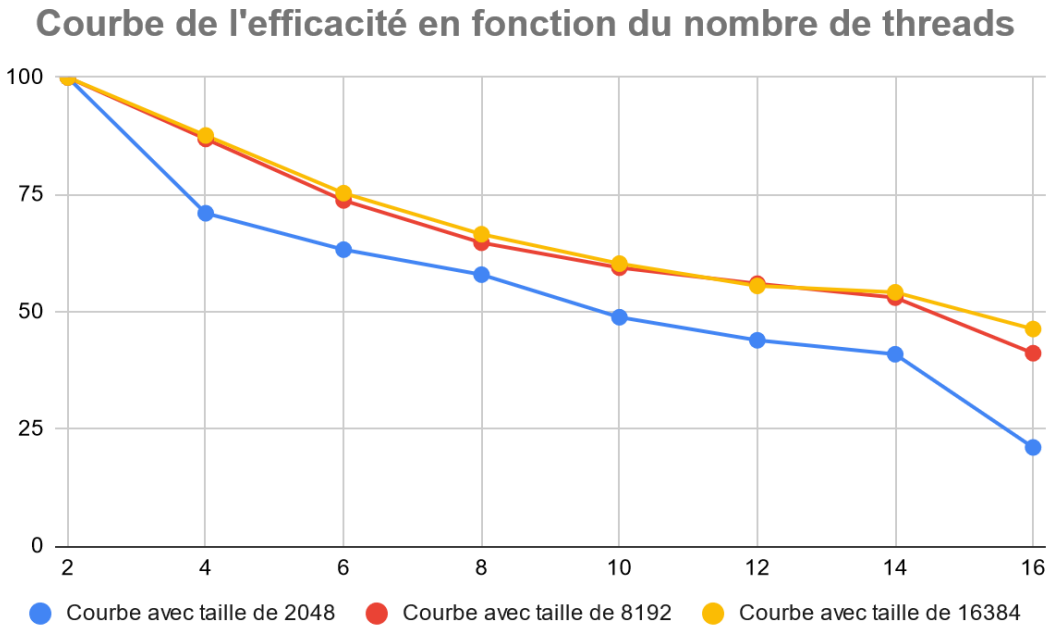
A) Calcul de la scalabilité forte

Dans ce cas, le problème reste fixe alors que le nombre de cœurs augmente. On s'attendrait idéalement à une scalabilité linéaire, c'est-à-dire que la diminution du temps d'exécution par rapport à la valeur de référence serait réciproque au nombre de cœurs ajoutés.



Cette courbe nous montre le temps d'exécution de notre algorithme en fonction du nombre de threads sur 3 séries. La première est sur une taille de matrice de 2048, la seconde est sur une taille de matrice de 8182 et la dernière série est sur une taille de 16384. La courbe nous montre clairement que le temps d'exécution diminue en fonction du nombre de thread utilisé de 2 à 14 threads. Cependant on voit que dans notre problème l'utilisation de 16 threads augmente grandement le temps d'exécution.

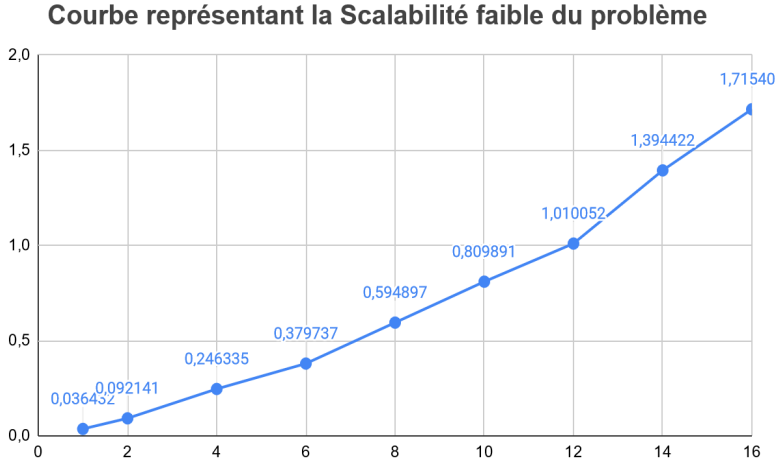
Maintenant intéressons nous à l'efficacité en fonction de l'ajout de ressource.
Formule pour calculer l'efficacité: T2/Tn * 2/n * 100
T2 : temps d'exécution sur 2 coeurs
Tn : temps d'exécution sur n coeurs
n : n coeur

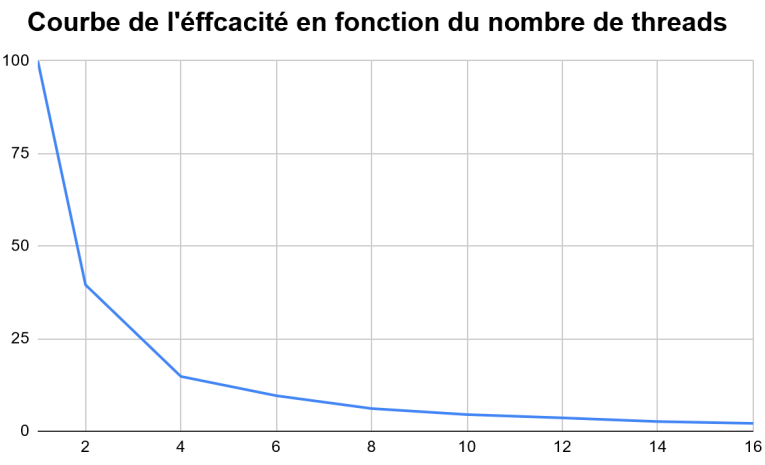


La courbe de l'efficacité en fonction du nombre de threads nous permet de mieux choisir les ressources pour les optimiser.
Une efficacité de 75 % ou plus est à privilégier. Avec nos résultats de mesure, nous recommandons donc de soumettre les tâches sur 6 cœurs pour les tailles de matrice de 8192 et 16384. Jusqu'à 16 cœurs, la durée d'exécution continue de décroître, mais l'amélioration obtenue avec plus de 6 cœurs serait une mauvaise utilisation des ressources.

B) Calcul de la scalabilité faible

Pour diminuer la scalabilité, la taille du problème est augmentée en proportion de l'ajout de cœurs pour obtenir idéalement une scalabilité linéaire et que la durée d'exécution demeure stable.





Après analyse notre algorithme possède une scalabilité faible non linéaire, en effet lorsque l'on augmente la taille de la matrice et le nombre de threads, l'efficacité de notre programme chute.

Bibliographie

Nous avons utilisé une base préexistante de l'algorithme sur le site :
https://www.codesansar.com/numerical-methods

Pour nous aider avec les formules des calculs prévisionnel nous avons utilisé le site :
https://hmf.enseeiht.fr

Pour mieux comprendre la scabilité nous avons utilisé :
https://docs.computecanada.ca

Pour voir notre jeu de données, allez sur le fichier PPD_DonnéesTests.pdf