SORBONNE UNIVERSITÉ

POLYTECH SORBONNE

# Projet

## Introduction au HPC

---

# Parallélisation de la résolution de systèmes linéaires creux par la méthode du gradient conjugué

---

Samy Asma
Homer Durand

*Encadrant :* Charles Bouillaguet

Juin 2020

# Contents

**1 Introduction** **2**

**2 Parallélisation OpenMP** **2**

2.1 Parallélisation du produit matrice vecteur . . . . . 2

2.1.1 Implémentation OpenMP . . . . . 2

2.1.2 Résultats . . . . . 2

**3 Parallélisation MPI** **3**

3.1 Mise en place des communication et explication du code . . . . . 3

3.1.1 Résultats . . . . . 4

3.2 Interpretation des résultats et ouverture . . . . . 5

**4 Parallélisation OpenMP et MPI** **5**

4.1 Description du code . . . . . 5

4.1.1 Résultats . . . . . 5

**5 Comparaison des différentes parallélisation** **6**

**6 Amélioration du code par partitionnement 2D** **7**

**7 Conclusion** **8**

# 1 Introduction

L'objectif de ce projet de paralléliser un programme séquentiel qui effectue la résolution de systèmes linéaires du type Ax = b (où A est une matrice creuse réelle symétrique définie positive) par la méthode du gradient conjugué.

On utilisera, pour comparer les performances de nos algorithmes, trois matrices relativement grandes possédant des taux de valeurs non-nulles diférents.

- La matrice **hood** posséde 0.02214% de valeurs non-nulles et est de taille 220 542.
- La matrice **Serena** posséde 0.00331% de valeurs non-nulles et est taiile 1 391 349. C'est une matrice "très creuse".
- La matrice **nd24k** posséde 0.5539% de valeurs non-nulles et est taiile 72 000. C'est une matrice de petite taille mais "très" dense (comparativement aux autres).

# 2 Parallélisation OpenMP

## 2.1 Parallélisation du produit matrice vecteur

### 2.1.1 Implémentation OpenMP

La partie la plus consommatrice en temps de calcul dans la méthode du gradient conjugué - quand il s'agit de matrice de grande taille - est le produit matrice-vecteur (fonction *sp_gemv*) qui est effectué à chaque itération.

Afin d'accélérer ce produit, on répartie la charge de calcul en distribuant les lignes de la matrice à l'ensemble des threads disponibles avec la directive **#pragma omp parallel for**.

Nous avons tester la parallélisation des autres parties de l'algorithme afin de vérifier si cela apportait un gain de performance mais il semblerait que cela sature l'accès à la mémoire et ralentirait donc légérement le code plutôt que de l'accélérer.

### 2.1.2 Résultats

Afin d'évaluer le gain de performance de cette parallélisation on exécute l'algorithme sur les matrices **hood**, **Serena** et **nd24k** depuis la machine *ppti-14-305-14* composé de *8 processeurs Intel Core i7-6700 de fréquence de base 3.40GHz).* On obtient les temps d'exécution suivant :

**Serena :**

| Threads | Temps d'exécution |
|---|---|
| 1 | 94.7s |
| 2 | 58.8s |
| 3 | 53.6s |
| 4 | 52.6s |
| 5 | 54.6s |
| 8 | 55.1s |

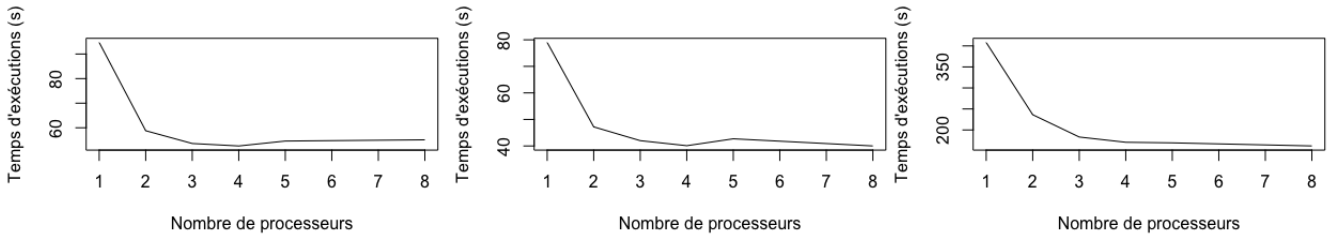
**Hood :**

| Threads | Temps d'exécution |
|---|---|
| 1 | 79.0s |
| 2 | 47.2s |
| 3 | 42.0s |
| 4 | 40.1s |
| 5 | 42.7s |
| 8 | 40.0s |

**Nd24k :**

| Threads | Temps d'exécution |
|---|---|
| 1 | 408.2s |
| 2 | 236.3s |
| 3 | 183.2s |
| 4 | 170.7s |
| 5 | 169.3s |
| 8 | 161.9s |

**Serena :** **Hood :** **Nd24k :**




On voit qu'à partir de 4 threads, le gain d'efficacité se stabilise et que l'ajout de nouveaux threads ne semble plus intéressant. Nous pouvons faire l'hypothèse que cela est dû au fait que l'accès aux cases mémoires proches les unes des autres est saturé sur la machine à partir d'un certain nombre de threads s'exécutant "en même temps". De plus la performance de cette parallélisation est limité par la partie séquentielle du code qui ne peut être accéléré et donc l'accélération diminue à mesure que l'on s'approche de la plus grande optimisation du code parallélisable.

# 3 Parallélisation MPI

## 3.1 Mise en place des communication et explication du code

Le temps de communication entre les différents processeurs étant très consommateur en temps, particulièrement pour la communication de la matrice (qui se fait en O(nnz)), nous avons décidé de travailler avec une granularité basse. On donne la Matrice a tout les processus et on crée des vecteurs locaux.

Tout d'abord, on parallélise l'initialisation des vecteurs locaux. Pour ne pas faire trop de communications, on calcule le produit scalaire de $r\_local$ et $z\_local$ puis on utilise $MPI\_Allreduce$ pour avoir rz globale.

**Initialisation**
Ensuite, pour calculer l'erreur, on procède de la meme manière que précédement, on fais le produit scalaire de r_local avec lui même, qu'on somme avec un MPI_Allreduce puis on applique la fonction *sqrt* pour en avoir la norme qui sera l'erreur initiale.

**Dans la boucle While**:
On calcule le produit matrice vecteur : puisque la matrice appartient à tout les processus, chaque processeur choisit la partie de la matrice dont il doit s'occuper. Dans notre cas, on a fait un partitionnement par ligne(row partitionning) : chaque processus a $n/nbr\_processus$ lignes de la matrice. Pour ce calcul, on a besoin du vecteur p entièrement. On stocke le résultat dans $q\_local$ qui sera utilisé par la suite. Chaque processeur calcule le produit scalaire de p et q qu'il possède en local et on utilise $All\_reduce$ pour trouver le produit p*q.

On fais les calculs d'incrémentation de chaque vecteur local ($x\_local$,$r\_local$,$z\_local$). Puisque nous ne les utilisons jamais dans leur globalité, chaque processeur va calculer ses vecteurs locaux.

Le calcul de r*z commence dans l'initialisation (utilisation de *All_reduce*). On calcul la nouvelle valeur de p localement mais comme on a besoin de p pour le produit matrice-vecteur, on utilise un *All_gatherv*(*p_local*,*taille_loc*, *MPI_DOUBLE*,p,*taille_local*,*deplac_local*, MPI_DOUBLE, *MPI_COMM_WORLD*) avec :

- *p_local* : les vecteurs locaux
- p : vecteur globale
- *taille_local* : vecteur contenant la taille de chaque vecteur local
- *deplac_local* : vecteurs qui possède chaque indice de la position de *p_local* sur le vecteur globale avec i correspondant aux rang du processus.

On calcule à nouveau l'erreur (comme précédemment).

**Fin** : Lorsque l'erreur est plus petite que le seuil, on fait un *All_gatherv* pour rassembler tout les *x_local* dans le x globale et ainsi obtenir la solution.

### 3.1.1 Résultats

Comme dans la section précédente, afin d'évaluer le gain de performance de cette parallélisation on exécute l'algorithme sur les matrices **Hood**, **Serena** et **nd24k** depuis les machines *ppti-14-305-14*, *ppti-14-305-15*, *ppti-14-305-16* composées chacune de *8 processeurs Intel Core i7-6700 de fréquence de base 3.40GHz)*. On obtient les temps d'exécution suivant :

**Serena :**

| Processus | Temps d'exécution |
|---|---|
| 2 | 317,9s |
| 3 | 209,5s |
| 4 | 411,7s |
| 8 | 972,6s |

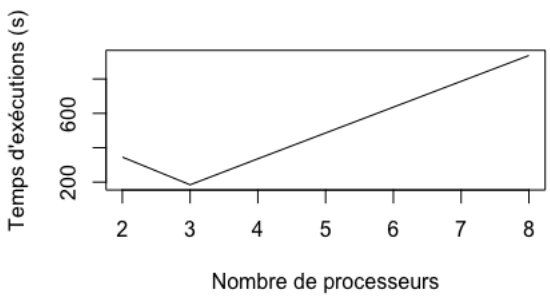
**Hood :**

| Prcessus | Temps d'exécution |
|---|---|
| 2 | 345,5s |
| 3 | 184,2s |
| 4 | 335,6s |
| 8 | 936,6s |

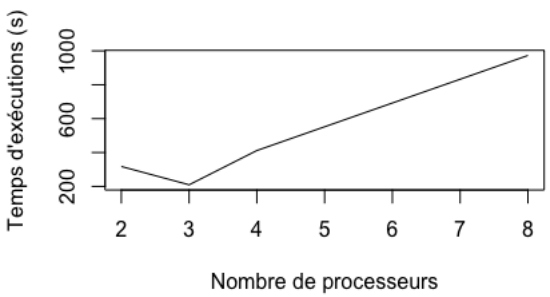
**ndk24 :**

| Processus | Temps d'exécution |
|---|---|
| 2 | 847,3s |
| 3 | 319,8s |
| 4 | 354,8s |
| 8 | 879,2s |

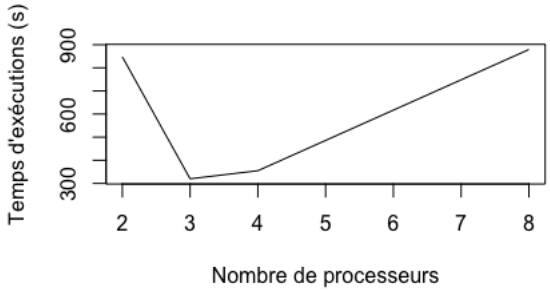
**Hood :**




**Serana :**



**Nd24k :**




## 3.2 Interpretation des résultats et ouverture

Sachant que **nd24k** est la plus dense, on comprend bien pourquoi - pour celle-ci - le temps de calcul avec 2 processus est très lent. Le plus petit temps d'exécution est pour 3 processus. Notre hypothèse est que notre partitionnement se faisant par ligne, nous sommes obligé d'utiliser un *All_gatherv* dans la boucle, se qui crée énormément de communication entre les différents processus et qui est donc très coûteux en temps. Cela expliquerai pourquoi MPI est plus lent que le code séquentiel (Voir partie comparaison) et aussi pourquoi le nombre optimal de processus est de trois. Avec trois processus, on a un juste compromis entre le coût de lancement de MPI et le surcoût des communications qui arrivent à partir de 4 à 5 processus. Afin d'améliorer la parallélisation MPI il serait intéressant d'utiliser un partitionnement par bloc (2D) qui permettrait de faire le produit matrice vecteur avec en divisant la mtrice et le vecteur entre tout les processus. Ceci permettrai de ne pas utiliser un *All_gatherv* dans le *while* et réduirai ainsi les coûts de communication. Nous détaillerons cette partie dans la section **Amélioration du code**.

# 4 Parallélisation OpenMP et MPI

## 4.1 Description du code

Suite aux deux partie précédente, nous avons tester un code hybride entre MPI et OpenMP. L'objectif est d'améliorer le code MPI en parallélisant les boucles for à l'intérieur de chaque machine en se propageant dans l'ensemble de ses coeurs. Pour la fonction *sp_gemmv*, on utilise un pragma omp parallel for afin de paralléliser les lignes à l'intérieur de chaque matrice. On a vu dans la partie OpenMP qu'il faut juste paralléliser cette fonction pour avoir une amélioration du temps de calcul.

### 4.1.1 Résultats

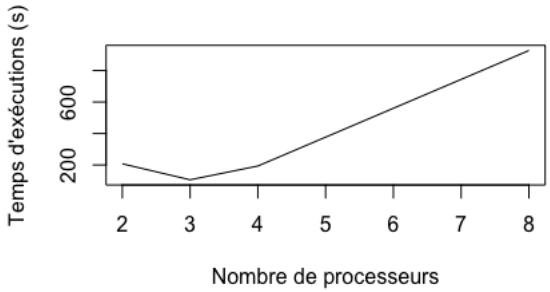
Comme dans la section précédente, afin d'évaluer le gain de performance de cette parallélisation on exécute l'algorithme sur les matrices **Hood**, **Serena** et **nd24k** depuis les machine *ppti-14-305-14*, *ppti-14-305-15*, *ppti-14-305-16* composé de *8 processeurs Intel Core i7-6700*

*de fréquence de base 3.40GHz).* On obtient les temps d'exécution suivant :

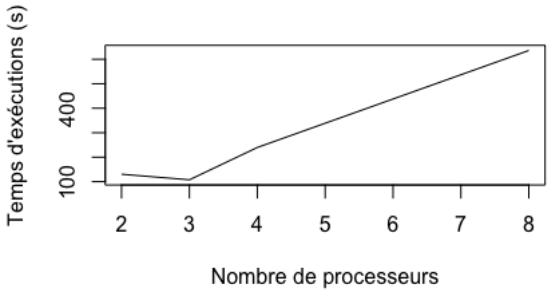
**Serena :** **Hood :** **ndk24 :**

| Processus | Temps d'exécution | Prcessus | Temps d'exécution | Processus | Temps d'execution |
|---|---|---|---|---|---|
| 2 | 130.4s | 2 | 207.3s | 2 | 521.1s |
| 3 | 107.6s | 3 | 105.6s | 3 | 119.1s |
| 4 | 239.6s | 4 | 193.2s | 4 | 186.3s |
| 8 | 635.8s | 8 | 927.4s | 8 | 1267.8s |

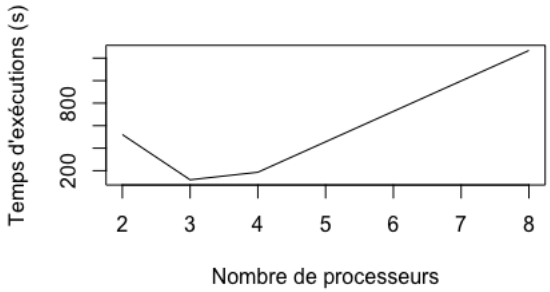
**Hood :**




**Serana :**




**Nd24k :**




On voit qu'on a une réduction globale du temps de calcul par rapport à la parallélisation MPI, ce qui semble logique étant donné qu'on parallélise le produit matrice vecteur à l'intérieur de chaque machine.

# 5 Comparaison des différentes parallélisation

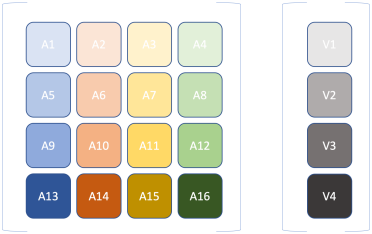
Pour chaque algorithme de parallélisation, on regarde la version la plus performante (qui dépend du nombre de threads, de processus, de machines utilisés) de celui-ci. Pour OpenMP, nous avons choisis 4 threads (la résolution n'étant pas significativement plus rapide au-delà.
Pour MPI, nous avons choisis 3 processus sur 3 machines. Pour la technique hybride, on choisit 4 thread et 3 processus sur 3 machines.

| Matrice | Séquentiel | OpenMP (4 Threads) | MPI | MPI + OpenMP |
|---|---|---|---|---|
| Hood | 76.2s | 40.1s | 184.2s | 107.6s |
| Serena | 95.4s | 52.6s | 209.5s | 105.6s |
| Nd24k | 413.5s | 170.7s | 319.8s | 119.1s |

On remarque que sur la matrice la plus dense (*Nd24k*), le code séquentiel est très lent, et le code MPI est plus rapide. Le fait que la matrice soit dense influe sur la performance de MPI car le coût des communications ne sur-compense pas le gain de performance.
Puisque OpenMP et MPI sont plus rapides que le code séquentiel, le code hybride permet d'aller beaucoup plus vite et divise par 4 le temps d'exécution (par rapport au séquentiel). Pour les matrice moins dense, on voit que MPI n'est pas très efficace, en effet il est plus lent que le code séquentiel du fait du coup de communication trop important. Le code hybride est plus performant que MPI car la parallélisation multi-threads accélère les calculs.
On voit qu' OpenMP apporte toujours un gain substantiel de performance car il permet de paralléliser le produit matrice vecteur sans qu'il ne soit nécessaire de communiquer entre les threads puisque la mémoire est partagée.

# 6 Amélioration du code par partitionnement 2D

À la vue des performances de notre algorithme de parallélisation multi-machines et multi-threads nous avons réfléchi à une parallélisation qui pourrait être plus efficaces car celle-ci réduirait les coups de communication. Nous n'avons malheureusement pas eu le temps de la mettre en place.
L'idée serait ici de partitionner la matrice A en la divisant en np lignes et np colonnes formant ainsi $np^2$ bloques - soit $np^2$ sous-matrices $A_i$ - ($np^2$ étant le nombre de machines) et le vecteur $x$ en np bloques - donc np sous-vecteur $x_i$.



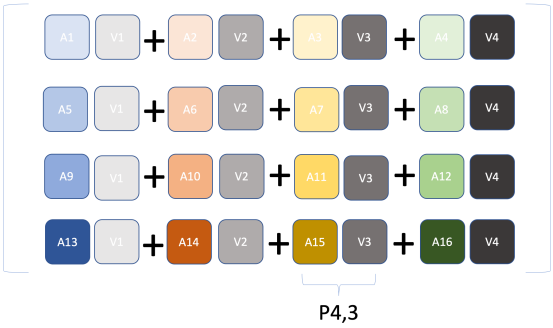

On peut ainsi distribuer chaque sous-matrice $A_i$ à un processus $P_{l,c}$ (avec $l = i//2$ et $c = i\%2$) auquel on affecte un vecteur $x_c$ ($MPI\_scatterv$). Une parallélisation OpenMP, sur le code séquentiel du produit matrice vecteur, nous permettra de calculer ,rapidement, chaque élément de la somme nécessaire au du vecteur $b_i$ (à la façon du code de parallélisation OpenMP).

Nous calculons cette somme avec la fonction $MPI\_reduce$ :

$$b_i = \sum_{u=O}^{np} A_{i//u+i\%u}.x_i$$

La parallélisation OpenMP se fait sur le produit $A_{i//u+i\%u}.x_i$.

Ainsi on pourra rassembler le vecteur $b$ en rassemblant l'ensemble des partitions de $b_i$.




On utilise ainsi le caractère creux de la matrice en calculant les produits matrices-vecteurs (à l'aide du code séquentiel optimisé pour les matrices creuses) parallélisés avec OpenMP (directive **#pragma omp parallel for** dans la fonction *sp_gemv*) tout en réduisant la quantité de communication puisque chaque processus ne communique qu'avec ses voisins de droite et de gauche.
On obtiendrait ainsi une version plus efficace du code parallélisé avec OpenMP et MPI.

# 7 Conclusion

On voit ici à travers la methode du gradient conjugué qu'il existe plusieurs technique de parallélisation d'un code donnée. Chacun avec ces avantages ainsi que ses inconvéniants. Nous n'avons pas réussi à avoir un code MPI efficace par rapport au code séquentielle. Cependant, il existe plusieurs voie d'amélioration comme un meilleur découpage de la matrice et des vecteurs. Nous voyons aussi que la technique hybrique MPI-OpenMP permet d'améliorer les performances de MPI. Il existe d'autre techniques de parallélisation comme la technique CUDA pour les GPU, qui peut être utilisé pour accélérer la méthode du gradient conjugué.