Sorbonne Université

Projet HPC
Rapport

---

# Résolution de systèmes linéaires creux par la méthode du gradient conjugué

---


***Étudiants :***
Boubacar Diallo
Jeffrey-Chris Kebey

***Enseignants :***
Charles Bouillaguet
Lilia Ziane Khodja
Gilles Moreau


June 5, 2020

# Sommaire

## 0.1 Introduction

Le Gradient Conjugué Préconditionné (GCP) est l'une des méthodes itératives les plus importantes pour la résolution systèmes linéaires du type Ax = b, où A est une matrice creuse réelle symétrique définie positive. Le but de ce projet nous est de paralléliser le programme séquentiel du solveur GCP qui nous a été fourni . Dans la suite, nous présenterons dans un premier temps nos algorithmes en MPI pur puis le hybride MPI + OpenMP utilisés et nos choix d'implémentions, dans un second temps nous montrerons les résultats que nous avons obtenus, leur analyse et la comparaison des différentes versions par rapport au programme séquentiel.

| Étape | Opération | Fonction |
|---|---|---|
| | $i = 0\,;\;\; x_0 = 0 \quad r_0 = b$ | |
| S1 | $z_0 = M^{-1} r_0$ | extract_diagonal |
| | $p_0 = z_0$ | |
| S2 | $\tau_0 =< r_0, r_0 >$ | dot |
| | **repeat** | |
| S3 | $\omega_i = A d_i$ | sp_gmv |
| S4 | $\beta_i = \tau_i \;\; \rho_i = \beta_i / < d_i, \omega_i >$ | dot |
| S5 | $x_{i+1} = x_i + \rho_i d_i$ | AXPY |
| S6 | $r_{i+1} = r_i - \rho_i d_i$ | AXPY |
| S7 | $z_{i+1} = M^{-1} r_{l+1}$ | |
| S8 | $\beta_{i+1} =< z_{i+1}, r_{i+1} >$ | dot |
| S9 | $d_{i+1} = (\beta_{i+1}/\beta_i) d_i + z_{i+1}$ | AXPY |
| S10 | $\tau_{i+1} =< r_{i+1}, r_{i+1} >$ | dot |
| | $i = i + 1$ | |
| | **while** | |

TABLE 1 – Pseudo code du GCP

Nous présentons rapidement la description algorithmique du GCP itératif classique à la Table 1.Dans l'implantation séquentielle les opérations suivantes sont exécutées : la fonction **extract_diagonal**(S1), un produit matrice creuse et d'un vecteur **sp_gmv** (S3), quatres produits vectoriels **dot**(S2,S4,S7 et S10), trois autres opérations que nous avons nommées **AXPY**(S5, S6 et S8)(similaires), et quelques opérations.

## 0.2 Méthodologie et Implémentation

Différentes méthodes de parallélisation sont possibles. Dans chaque section, nous détaillerons le choix d'implantations et nous monterons un algorithme de fonctionnement du programme. En MPI, le code est compilé avec le compilateur mpicc. Nous avons également parallélisé le code totalement en OpenMP (code disponible dans l'archive) avant de réaliser le code hybride MPI + OpenMP.

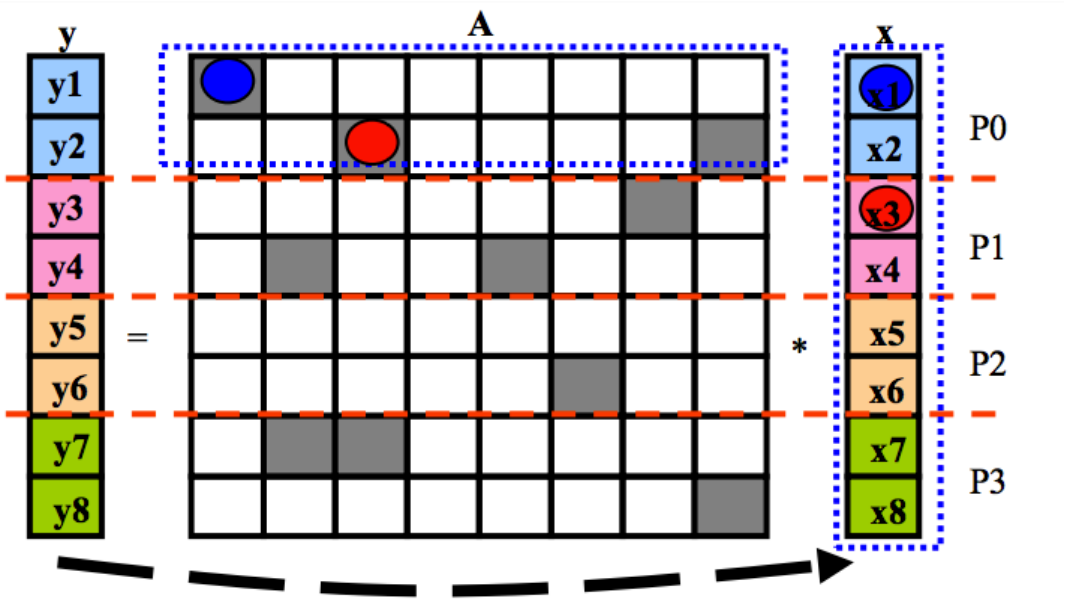



FIGURE 1 – Partition de la matrice et du vecteur entre processus

### 0.2.1 Parallélisation MPI pur

Dans cette section, nous analysons les modèles de communication de l'implémentions de MPI PUR sur une plate-forme à mémoire distribuée pour résoudre le GCP.
Les considérations suivantes sont prises en compte dans l'implémentions :

1. Chaque nœud détient un composant du système de taille n_local obtenue à partir d'une division entière supérieure de n(taille du système) par nproc(nombre de processus),et le reste est rajouté à la portion du dernier processus.
   Cette répartition est illustrée dans la Figure 1.
2. Afin d'améliorer efficacement l'envoi des données effectué par le noeud 0 que nous appelons processus maître , nous envoyons simplement les sous-parties de la matrice et vecteur plutôt que de les envoyer entièrement.
3. Reconstitution de la solution globale,chaque solution partielle est mise à la ligne I dont elle a été extraite.

#### Parallélisation de l'algorithme GCP

Tel qu'illustré dans le pseudo code précédent, l'algorithme du GCP comprend quatre types de calculs à chaque itération que nous pouvons paralléliser comme suit :

**sp_gemv(S3) :** Celle-ci est l'étape de calcul qui requiert le plus de temps ses arguments sont la matrice de coefficients A, qui est distribuée par blocs de lignes, et le vecteur d, qui est partitionné et distribué selon A. Une étape de communication est nécessaire avant d'exécuter cette fonction afin d'assembler les parties distribuées du vecteur d en un seul vecteur p, qui est répliqué dans tous les processus.Nous désignons cette communication comme d→ p, qui peut être effectuée dans MPI via MPI Allgatherv. Notez que le vecteur p est le seul tableau qui est répliqué dans tous les processus Après cela, le calcul peut se dérouler en parallèle et chaque processus calcule sa tranche locale du vecteur de sortie

**dot(S2,S4,S7,S10)** Dans cette fonction, chaque processus peut calculer simultanément un résultat partiel (S2,S4). Ensuite, ces valeurs intermédiaires valeurs sont réduites en une valeur puis répliquée sur les processus.En appliquant cette idée à tous les produits dot, il existe quatre synchronisations parce que $\rho, \beta, \tau$ sont répliqués sur chaque processus.Nous implémentons cette réduction de MPI en utilisant MPI Allreduce.

**Opérations SAXPY :** :Les opérations S5,S6 et S9 sont connues comme étant de simples ax plus y (SAXPY), où a est un scalaire et x et y sont des vecteurs.Chacune de ces opérations se fait de façon partielle sur chaque processus,car tous les vecteurs sont distribués identiquement parmi les processus, ces étapes ne requièrent pas de communication.

**Operations S1 et S7** :L'opération S1 (incluant la fonction extract_diagonal) et S2 sont les instructions de pré-conditionnement du programme GCP équivaut à diviser chaque élément de r par le coefficient de la rangée correspondante dans A. Aucune communication n'est requise car chaque processus stocke les éléments de la diagonale de A qui lui correspond et un vecteur local r donc les calculs peuvent s'effectuer parallèlement.

En tenant compte des directives de parallélisation des lignes précédentes la version parallèle de l'algorithme du gradient conjugué pré-conditionné par blocs peut être décrit à l'aide du pseudo code suivant :

**Processus maître**

- charger la matrice et le vecteur depuis le système de fichiers en mémoire
- Distribuer les données entre les processus

**Processus maître et esclaves**

- Réception des partitions de données
- Initialisation des vecteurs locaux.
- envoie et réception de la portion du vecteur à multiplier
- Calcul partiel du produit matrice-vecteur
- Effectuer le produits scalaires
- Reconstitution du produit entre les noeuds
- Les opérations SAXPY
- sortir de la boucle si convergence
- Envoi au processeur maître de la solution partielle

**Processus maître**

- Reconstitution de la solution globale

### 0.2.2 Parallélisation MPI + OpenMP

Dans cette sous-section, nous détaillons comment exploiter deux niveaux de parallélisme via une combinaison de deux interfaces de programmation parallèles : MPI et OpenMPI.
Nous avons principalement créer des équipes de threads avec la directive #pragma omp parallel dans les fonctions : **extract_diagonal**(S1),**sp_gmv** (S3),**dot**(S2,S4,S7 et S10),**AXPY**,

dans les opérations(partition de la matrice et du vecteur a multiplier ) exécuter uniquement par le processus master,et d'autre sections parallélisables, en y ajoutant la directive #pragma om for pour partager le travail entre l'équipe de thread créée. A cet effet chaque processus détient une équipe de thread qui s'exécutent simultanément sur chacune des ses opérations.

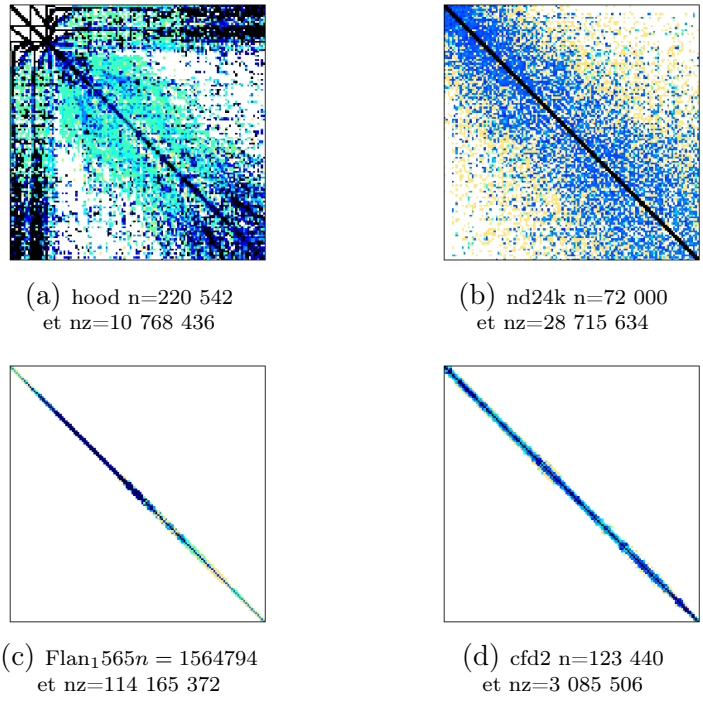
## 0.3 Résultats Expérimentaux

Dans cette section, nous présentons les résultats des nos expérimentations effectuées sur des matrices des caractéristiques différentes .Notez que tous nos tests ont été effectués sur ppti-305 et ppti-gpu-1.ppti-305 chaque pc est équipé de 4 coeurs Intel Core i7-6700,fonctionnant à 3.4GHz, et avec 16Go de RAM .Le ppti-gpu-1 est de 44 coeurs, 384Go de RAM
Comme nos deux plates-formes sont différentes l'une avec une mémoire distribuée(ppti-14-305) et l'autre avec une mémoire partagée(ppti-gpu1) nous avons procède comme suit pour les tests :

— **ppti-14-305** Chaque noeud correspond à une machine au sein de laquelle est exécuté 4 processus .

— **ppti-gpu1** un noeud équivaut à 4 processus ;

Nous avons choisi quatre matrices pour l'expérimentation de notre implémentions qui sont représentées dans les figures suivantes :



(a) hood n=220 542 et nz=10 768 436

(b) nd24k n=72 000 et nz=28 715 634

(c) $\text{Flan}_1565n = 1564794$ et nz=114 165 372

(d) cfd2 n=123 440 et nz=3 085 506

FIGURE 2 – Matrices utilisées pour les tests de l'implimentation.

| Matrice | ppti-14-305-01 | ppti-gpu-1 |
|---|---|---|
| cfd2 | 35.5 | 44.6 |
| hood | 82.9 | 111.9 |
| nd24k | 418.9 | 539.0 |
| Flan_1565 | 2033.8 | 2571.9s |

TABLE 2 – Temps d'exécution en séquentiel sur ppti-14-305-01 et ppti-gpu1

### 0.3.1 MPI Pur

Les TABLES 3 et 4 indiquent respectivement le temps d'exécution de MPI pur pour les matrices répertoriées à la TABLE 2 sur les deux plate-formes(ppti-14-305-01 et ppti-gpu1) en variant le nombre de noeud .

| Nombre de noeuds | 2 | 4 | 8 | 12 |
|---|---|---|---|---|
| cfd2 | 24.2 | 13.1 | 10.1 | 9.5 |
| hood | 66.6 | 37.5 | 22.6 | 18.7 |
| nd24k | 302.3 | 150.5 | 88.6 | 62. |
| Flan_1565 | 1191.9 | 771.6 | 480.0 | 362.1 |

TABLE 3 – Temps d'exécution des matrices de la TABLE 2 sur ppti-gpu-1

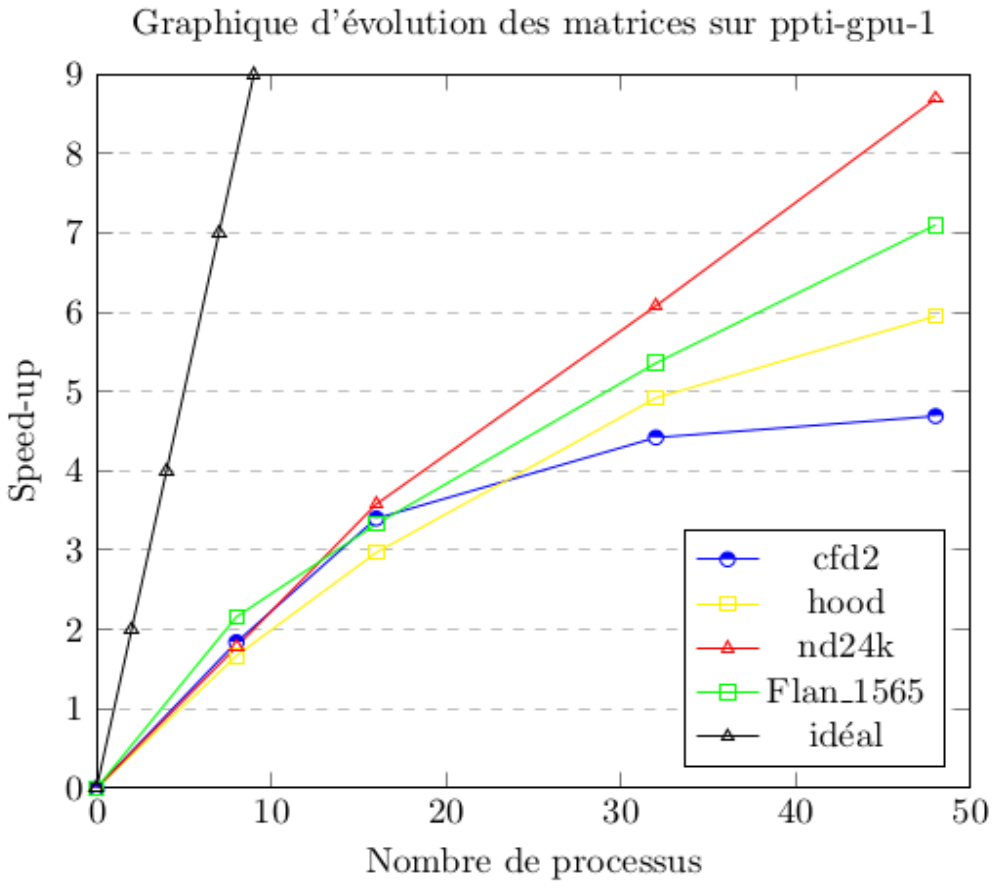



FIGURE 3 – Évolutions d'accélérations des matrices de TABLE1 sur ppt-gpu-1

| Nombre de noeuds | 2 | 4 | 8 | 12 |
|---|---|---|---|---|
| cfd2 | 25.3 | 23.8 | 114.0 | 256.5 |
| hood | 69.0 | 52.0 | 128.1 | 396.3 |
| nd24k | 243.6 | 181.6 | 156.0 | 255.4 |

TABLE 4 – Temps d'exécution des matrices sur ppti-305

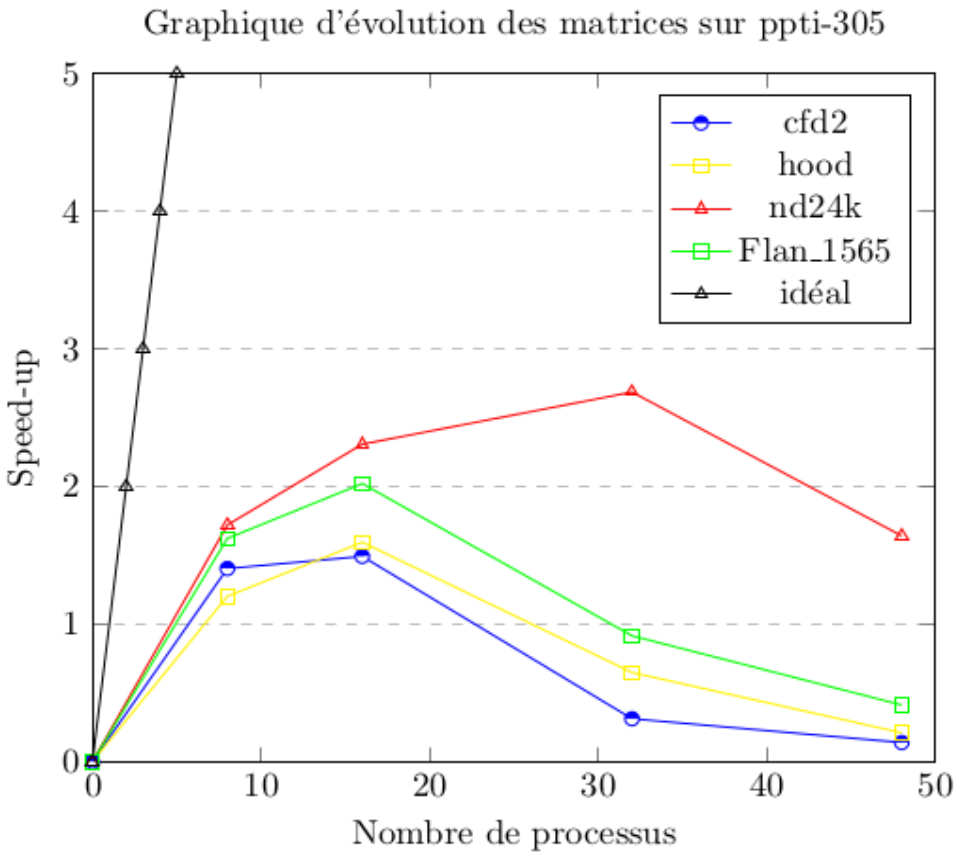



FIGURE 4 – Évolutions d'accélérations des matrices de TABLE1 sur ppti-305

Les Figures 3 et 4 montrent l'accélération en fonction du nombre de processus(quatre processus pour chaque machine de ppti) sur nos deux plateformes. Avant tout d'abord, on constate que le ppti-gpu a des meilleures performances par rapport au ppti-305 , et cela était prévisible car le cluster ppti-gpu les communications sont effectuées à travers une mémoire partagée,en revanche les noeud du ppti-305 communiquent à travers un réseau ce qui pénalisent fortement ses accélérations.

On peut voir que l'implémentions actuelle évolue pas très proche de l'accélération idéale souhaitée pour les deux clusters. Une tendance peut être observée, plus la matrice est dense plus l'accélération croit,cela est logique car la quantité de données à traiter augmente également et par conséquent, davantage de processus peuvent être utilisés avant que la quantité croissante de communications ne commence à dominer le temps d'exécution.

Notre méthode de distribution de la matrice entre les noeuds handicape aussi de façon générales les performances car les nœuds ont des charges de travail déséquilibrées. Les nœuds qui ont les sous-matrices les plus denses mettent plus de temps pour traiter leurs sous-matrices que les nœuds qui ont les sous-matrices les plus creuses.Ce problème

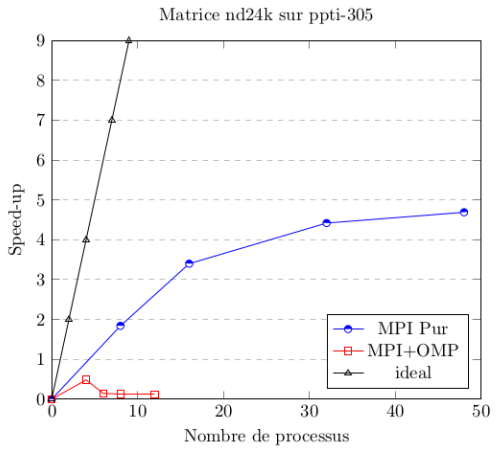
pourrait être résolu de manière efficace en appliquant des permutations des lignes et des colonnes, de telle sorte à ce que la distribution des coefficients non nuls pour toutes les sous-matrices soient proches .

### 0.3.2 Hybrid MPI-OpenMP

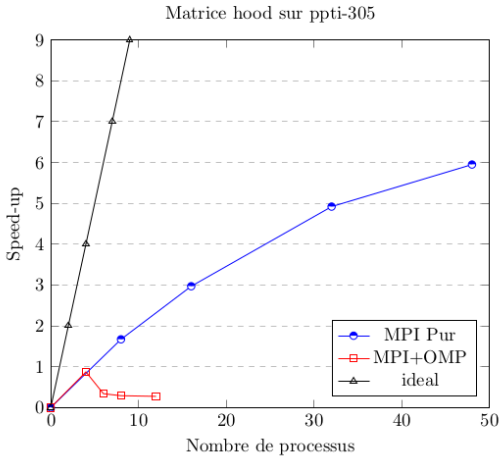
Dans cette section, nous mesurons les performances obtenues avec le hybrid( MPI et OpenMP )et les comparer avec MPI pur.
Nous avons fixé le nombre de noeud à deux(2 PC), et utilisons un processus par noeud en faisant varier le nombre de thread à l'intérieur de chaque noeud telle est la configurations que nous avons appliquée.(voir TABLE 5).

| Nombre de Noeud/ | 4 | 6 | 8 | 12 |
|---|---|---|---|---|
| cfd2 | 73.5 | 244.2 | 270.7 | 292.3 |
| hood | 95 | 245.6 | 286.4s | 302.8 |
| nd24k | 186.9 | 235.2 | 224.3 | 227.4 |
| Flan_1565 | | | | |

TABLE 5 – Exécution des matrices de la TABLE 1 sur gpu-1




(a) Exécution de matrice nd24k sur MPI pu et MPI+OMP




(b) Exécution de matrice hood sur MPI pu et MPI+OMP

FIGURE 5 – Comparaison de MPI pur et MPI+OpenMP

La Figure 5 indique l'évolution atteinte de la matrice nd24k et hood avec l'implémentions hybride sur la plateforme ppt-305.

La configuration un processus par noeud a des performances nettement inférieure celle de 4 processus par noeud de MPI pur . Avec cette configuration, la matrice et les vecteurs sont distribués sur les nœuds avec une partition par nœud, donc chaque fois que les informations doivent être transférées entre les processus, cela se fait sur le réseau. Ceci est bien sûr plus lent que vitesse de transfert locale pouvant être obtenue entre des processus sur les mêmes nœuds lors de l'utilisation de configurations avec plusieurs processus sur le même nœud. De plus, lorsqu'un seul processus est alloué par nœud, tous les threads de travail ne seront pas alloués au même socket que le processus.Par conséquent, certains

threads devront très probablement accéder à la mémoire allouée à l'autre socket du nœud, augmentant ainsi le temps nécessaire pour accéder à la mémoire.

## 0.4 Conclusion

A travers ce projet, nous avons pu voir les possibilités et l'importance de la programmation en parallèle.De ce travail, nous avons abordé la Résolution de systèmes linéaires creux par la méthode du gradient conjugué .Nous avons d'abord analysé les tâches MPI pur et hybride(MPI + OpenMP) puis comparer les deux versions.
D'après l'expérimentation de notre implémentation le MPI pur à des meilleurs performances par rapport au hybrid avec une configuration un processus par noeud et multithreading à l'intérieur.

# Bibliographie

[1] Roman Iakymchuk, Maria Barreda, Stef Graillat, José Aliaga, Enrique Quintana-Ortí. Reproducibility of Parallel Preconditioned Conjugate Gradient in Hybrid Programming Environments. 2020. hal- 02427795v2.

[2] Hao Zhuang *, and Jin Wang [*Sparse Matrix-Vector Multiplication for Circuit Simulation*]. CSE260 (2012 Fall) Course Project Report.

[3] Alexandersen, J., Lazarov, B. S., Dammann, B. (2012), [*Parallel Sparse Matrix - Vector Product : Pure MPI and hybrid MPI-OpenMP implementation. Technical University of Denmark. D T U Compute. Technical Report, No. 2012-10*].