HPC :
Algorithme de Lanczos par bloc

UE : Calcul haute performance : notions de base

Étudiants :
Tahar AMAIRI
Hamza RAIS

Enseignant :
Charles Bouillaguet

3 juin 2022

Table des matières

1 Introduction **2**

2 Exploration du code séquentiel **2**

3 MPI **3**
3.1 Phase 1 3
3.2 Phase 2 & 3 4

4 OpenMP **5**
4.1 Phase 1 5
4.2 Phase 2 & 3 5

5 Implémentations supplémentaires **6**
5.1 Hybride 6
5.2 Optimisation du code séquentiel 6
5.3 Checkpointing 6
5.4 Modification de la taille de la stack 7

6 Benchmarks **7**
6.1 MPI vs OpenMP 7
6.2 MPI vs Hybride 9

7 Conclusion **10**

8 Annexe **10**

1 Introduction

Le but du projet est de paralléliser l'algorithme de **Lanczos par bloc** permettant de résoudre les systèmes linéaires de la forme suivante : $Ax = 0 \mod p$[1]. Pour cela, on implémentera différentes versions :

— MPI
— OpenMP
— hybride, i.e, MPI + OpenMP

Pour chacune d'entre elles, on implémentera aussi un *checkpointing*. Ce rapport a pour objet de décrire brièvement les stratégies de parallélisation mises en place. Ainsi, pour plus de détails et d'informations, il faut se référer au code contenant des commentaires. Finalement, on finira par discuter des performances obtenues.

2 Exploration du code séquentiel

Avant de débuter la parallélisation, il est nécessaire d'obtenir les *hotspots* du code séquentiel. De cette manière, il est possible de connaître les parties de l'algorithme nécessitant une réelle parallélisation. Pour cela, nous allons utiliser le **GNU profiler** :

— 62 % @ `sparse_matrix_vector_product` (phase 1)
— 24 % @ `orthogonalize` (phase 3)
— 14 % @ `block_dot_products` (phase 2)

En utilisant maintenant la commande **perf**, on peut aussi obtenir les lignes dans chacune de ces fonctions :

— phase 1 : 75 % @ `y[i * n + l] = (a + v * b) % prime`
— phase 2 : 42 % @ `C[i * n + j] = (x + y * z) % prime` @ `matmul_CpAtB`
— phase 3 : 24 % @ `C[i * n + j] = (x + y * z) % prime` @ `matmul_CpAB`

On observe que les hotspots correspondent aux différentes phases de l'algorithme. Avec perf, on remarque que deux autres fonctions sont aussi appelées souvent : `matmul_CpAtB` et `matmul_CpAB` qui effectuent le produit entre deux matrices de taille $n \times n$. Finalement, et sans surprise, les lignes qui prennent le plus de temps d'exécution sont celles impliquant l'opération modulaire, connue pour être très lente.

1. Ici, x peut-être un vecteur si $n = 1$ ou bien une matrice si $n > 1$. Par conséquent, par soucis de simplicité, on utilisera toujours le terme *vecteur* pour s'y référer

3 MPI

3.1 Phase 1

Pour paralléliser le produit matrice-vecteur, il est tout d'abord nécessaire de distribuer la matrice et on peut faire cela de deux manières : 1D ou 2D. Étant donné que la distribution 2D engendre beaucoup moins de temps de communication [2], alors notre choix s'est naturellement porté sur celle-ci. Ainsi, si chaque processus détient une sous-matrice, on peut représenter la topologie du réseau comme une grille 2D : en effet, on peut voir la matrice principale comme un plan 2D constitué de sous-matrices représentant chacune un processus la détenant. Cette manière de superposer la distribution de la matrice sur la topologie du réseau permet de faciliter énormément les communications au sein des processus car on peut tirer profit de deux notions importantes chez MPI : les **communicateurs** et les **opérations collectives**.

Prenons le cas d'une distribution 2D avec $np = 4$ (le nombre de processus) du produit matrice-vecteur suivant :

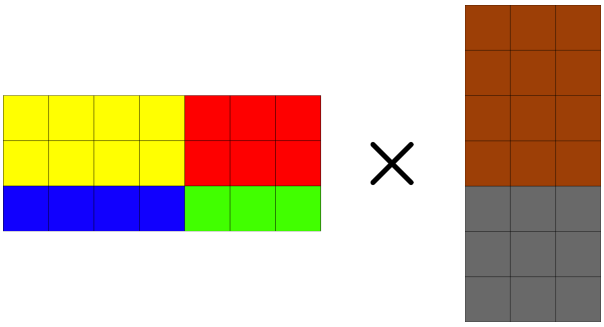


FIGURE 1 – $M \times V$ avec M une matrice de taille 3×7 et V un vecteur de taille 7×3

Ici, les processus p_0 et p_2 ont besoin de la partie en marron de V tandis que les processus p_1 et p_3 ont besoin de la partie en gris. Ainsi, les processus sur la même colonne sur la grille 2D de la topologie ont besoin de la même sous-partie du vecteur V. De même, pour obtenir le résultat du produit scalaire entre la première ligne de M et la première colonne de V, il suffit de sommer les résultats obtenus par p_0 et p_1. Par conséquent, pour obtenir le résultat de MV, nous avons besoin d'effectuer des communications seulement entre chaque ligne et colonne.

Sous MPI, cette topologie de grille en 2D peut être facilement obtenue via la fonction `MPI_Cart_create` : elle permet en plus d'obtenir pour chaque ligne et colonne de la grille, un communicateur (en utilisant la fonction `MPI_Comm_split`). Ainsi, pour la distribution du vecteur V, le processus *root* de chaque communicateur colonne va distribuer à ses pairs (à l'aide de `MPI_Bcast`) la sous-partie de V qu'il a obtenu. Concernant la somme par ligne, nous n'allons pas utiliser `MPI_Reduction` vu que nous sommes dans le corps $\mathbb{Z}/p\mathbb{Z}$ et que MPI n'implémente pas la réduction modulaire.

2. en effet, avec une distribution 2D, on divise le temps de communication par un facteur $\sqrt{p}$, avec p le nombre de processus

À la place, chaque processus enverra ses résultats au processus root de son communicateur ligne et celui-ci fera la réduction mod p. Finalement, pour obtenir le résultat de ce produit, chaque processus root de chaque communicateur ligne enverra ses résultats au processus root de la topologie à l'aide d'un `MPI_Gatherv`. Détaillons maintenant la procédure dans son entièreté :

1. **Initialisation** : On initialise l'environnement MPI et la grille 2D tout en attribuant à chaque processus ses coordonnées qui vont déterminer s'il est root d'un communicateur ligne/colonne. Le processus root de la topologie charge la matrice M et diffuse sa taille. Chaque processus détermine aussi la taille de sa sous-matrice selon ses coordonnées. On notera d'ailleurs que le processus root prendra l'excès de ligne/colonne si np ne divise pas la taille de M.
2. **Distribution des sous-matrices** : Le processus root ayant chargé la matrice distribue à chaque processus de la grille sa sous-matrice m.
3. **Distribution des sous-blocs de vecteur** : Étant donné que l'algorithme de Lanczos est itératif et que le processus root doit avoir à chaque itération le vecteur V, il est donc responsable de la distribution en amont de chaque sous-blocs de V. Pour cela, il diffuse à tous les processus root d'un communicateur colonne leurs sous-blocs v. Par la suite, ils diffusent dans leur communicateur colonne respectif le sous-bloc qu'ils ont obtenu. Par ailleurs, la distribution de V se fera en 1D et donc par ligne.
4. **Produit** $m \times v$: Chaque processus effectue le sous-produit qu'il lui a été assigné. Il envoie par la suite le résultat au processus root de son communicateur ligne qui effectuera la réduction modulaire.
5. **Rassemblement des résultats** : Finalement, les processus root de chaque communicateur ligne envoient leurs résultats au processus root de la topologie.

On remarque que les communications par colonne sont associées à la distribution des sous-blocs v tandis que celles par ligne sont associées au rassemblement des résultats (sauf pour l'étape 5, où le rassemblement se fait via la première colonne). Ainsi, dans le cas d'une transposition, cette observation s'inverse.

3.2 Phase 2 & 3

Les deux dernières phases correspondent, en majorité, à des produits de petites matrices de taille $n \times n$: chaque produit consiste à multiplier des sous-blocs de taille $n \times n$ de vecteurs avec d'autre sous-blocs ou matrice de même taille. Comme les vecteurs sont de taille très grandes, ces opérations sont effectuées énormément de fois. Par conséquent, la stratégie mise en place pour ces deux phases est très simple : le processus root distribue, de manière équitable (celui-ci prendra l'excès s'il y a lieu), à chaque processus sa liste de sous-blocs dont il fera le produit. Par la suite, tous les processus renvoient leurs résultats et selon la phase le processus root :

— **Phase 2** : effectue une réduction modulaire.
— **Phase 3** : rassemble dans le bon ordre les résultats.

Il est aussi important de noter que certaines communications supplémentaires sont effectuées : en effet, le processus root doit envoyer par exemple lors de la deuxième phase deux matrices de taille $n \times n$ à tous les processus pour qu'ils puissent effectuer le produit avec leurs sous-blocs.

4 OpenMP

4.1 Phase 1

En utilisant OpenMP, on n'effectue aucune distribution 2D directe de la matrice M. En effet, la stratégie est beaucoup plus simple car l'unique contrainte impliquant la parallélisation est le conflit d'écriture entre threads. Par conséquent, l'idée est de créer un tableau *cache* permettant à chaque thread de stocker ses résultats. La stratégie peut-être décrite comme ceci :

1. **Initialisation** : On initialise un tableau *cache* d'`u64` à 0. Chaque thread aura sa propre copie de celui-ci.
2. **Produit** : On parallélise la boucle `for` du produit matriciel à l'aide d'OpenMP et chaque thread peut stocker sans risque son résultat dans son tableau privé cache. Par ailleurs, toutes les opérations sont effectuées sans modulo.
3. **Réduction** : On effectue la somme de toutes les versions obtenues du tableau cache à l'aide de la clause `reduction`. Pour de très grands nombres premiers, on aurait eu un overflow si cache était un tableau d'`u32`, c'est pour cela qu'il a été initialisé pour des entiers de 64 bits.
4. **Réduction modulaire** : Finalement, on enregistre le résultat dans le tableau cible en effectuant une réduction modulaire.

4.2 Phase 2 & 3

Tout comme pour la version MPI, la stratégie pour ces deux phases est de distribuer les différents produits :

— **Phase 2** : On rencontre la même contrainte que pour la première phase, i.e, un conflit d'écriture. En effet, lors de la deuxième phase, on effectue un produit de matrices de taille $n * n$ dont le résultat est toujours stocker dans une même matrice. Ainsi, on va utiliser la même astuce de tableau cache intermédiaire mais ici on en aura deux car on effectue deux produits en simultané. Malheureusement, on ne pourra pas faire de réduction avec la clause d'OpenMP directement car les fonctions effectuant le produit matriciel n'accepte que des tableaux `u32`[3]. Pour remédier à ce soucis, on utilisera la clause `no wait` pour la boucle `for` effectuant les produits permettant ainsi aux threads les plus rapides d'être libérés plus tôt afin de sommer les résultats dans les tableaux cibles. Finalement, afin d'éviter un conflit d'écriture, on utilisera la clause `critical` lors de la réduction modulaire.

3. il est vrai qu'on pouvait changer le type de la fonction, mais il n'y avait aucune différence de performance entre les deux versions

Phase 3 : La parallélisation de cette phase est très directe car il n'y a aucun conflit d'écriture. Par conséquent, il suffit juste de paralléliser les boucles `for` effectuant le produit matriciel.

5 Implémentations supplémentaires

5.1 Hybride

La version hybride permet à l'aide de MPI de diviser le problème en sous-problèmes plus petits[4], i.e, que chaque processus est affecté à une sous-matrice (phase 1) et à une liste de sous-blocs de vecteurs (phase 2 et 3). Quant à l'utilisation d'OpenMP, elle permet à chaque processus de paralléliser les tâches qu'il doit effectuer.

L'implémentation de cette version hybride est très directe car la version MPI ne change en aucun cas comment les sous-problèmes (produit matrice-vecteur, matrice-matrice etc...) sont résolus : en effet, elle ajoute uniquement une couche de communication et de distribution. Ainsi, il suffit juste d'implémenter la version d'OpenMP directement sur celle de MPI.

5.2 Optimisation du code séquentiel

Le produit entre matrices de taille $n \times n$ peut-être amélioré de deux manières :

1. **Réduction modulaire** : On peut limiter le nombre de réduction modulaire effectuée. En effet, lors du produit scalaire entre une ligne et une colonne, il est possible de stocker le résultat dans une variable intermédiaire en `u64` et d'effectuer le modulo après la fin de la troisième boucle `for`.
2. **Localité des données** : En transposant les matrices, on peut les lire de manière contiguë dans la mémoire ce qui permet d'utiliser les caches du CPU. Bien plus, cela permet de *vectoriser* l'opération à l'aide d'instruction **SIMD** en utilisant OpenMP.

5.3 Checkpointing

Pour activer le checkpointing, il faut ajouter à l'exécution l'option `-checkpoint cp`, avec `cp` l'intervalle de sauvegarde en seconde (si aucun intervalle est fourni, `cp` est fixé à 1 minute). À l'inverse, si on veut charger un checkpoint, on utilisera l'option `-load-checkpoint` qui ne prend aucun argument. En effet, les fichiers de checkpoint au format `txt` sont nommés et reconnus directement. Par ailleurs, pour les implémentations MPI et hybride, seul le processus root effectue le checkpoint car il détient tous les tableaux nécessaires à la reprise d'une itération. Finalement, des modifications ont été apportées à l'affichage lors de l'exécution du programme à partir d'un checkpoint : affichage du bon nombre d'itérations, estimation correcte du temps d'exécution restant etc...

4. cf. le paradigme "**divide and conquer**"

5.4 Modification de la taille de la stack

Il est nécessaire d'augmenter la taille de la pile afin d'éviter des **segmentation faults** lors de l'utilisation d'un `n` élevé ou d'une matrice *challenge*. Cette modification doit aussi s'appliquer sur tous les noeuds sur lesquels on veut lancer la parallélisation dans le cas d'une implémentation MPI/hybride. Par conséquent, il a été nécessaire d'effectuer des ajouts pour chaque librairie :

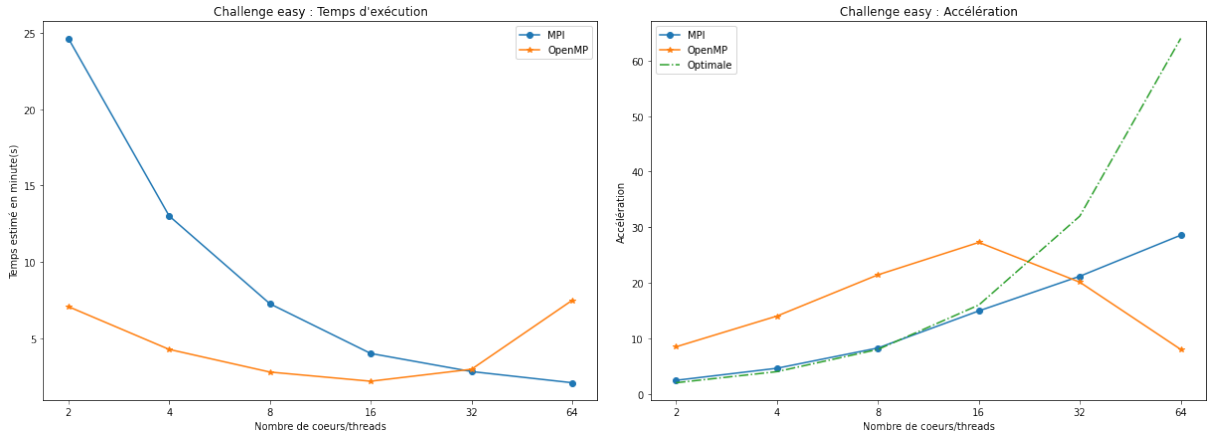
— **MPI** : On utilisera la fonction `setrlimit` qui permet de fixer la taille de la pile pour le thread **courant uniquement**. On fixera la taille de la pile à **1 Go**.
— **OpenMP** : On utilisera la commande `export OMP_STACKSIZE="1G"` avant l'exécution du programme. Par conséquent, il est nécessaire de fournir cette variable d'environnement aussi à `mpiexec` ou `mpirun` pour la version hybride (voir l'annexe pour plus de détails). Finalement, il est important de noter que `setrlimit` complète cette solution car elle n'est valable que pour les threads **fils**.

6 Benchmarks

Tous les benchmarks ont été effectués en utilisant les options suivantes : `-prime 1073741789 -n 4`. Le temps obtenu est celui indiqué par l'estimation, il y a donc quelques fluctuations lorsqu'une implémentation stagne et n'arrive plus à obtenir d'amélioration. Tous les fichiers ont été compilés avec les options `-O3` et `-mfma`[5]. Finalement, tous les résultats se trouvent dans le dossier `benchmarks`.

6.1 MPI vs OpenMP

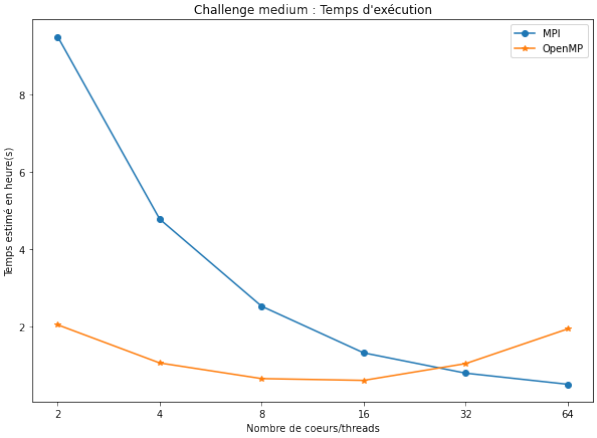
⇒ **CPU** : `(AMD EPYC 7452 32-Core Processor)` × 2



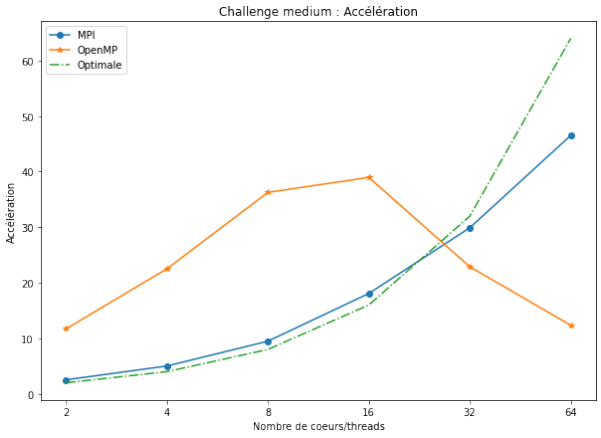
(a) Temps d'exécution (b) Accélération

FIGURE 2 – Challenge **easy** (60 minutes en séquentiel)

5. on obtenait des résultats légèrement meilleurs avec ce flag qu'avec `-mavx2`. De même, on a évité l'utilisation du flag `-march=native` pour la même raison (sachant que la compilation se fait sur le serveur frontend et non sur les noeuds réservés, ce flag n'a plus d'intérêt donc)

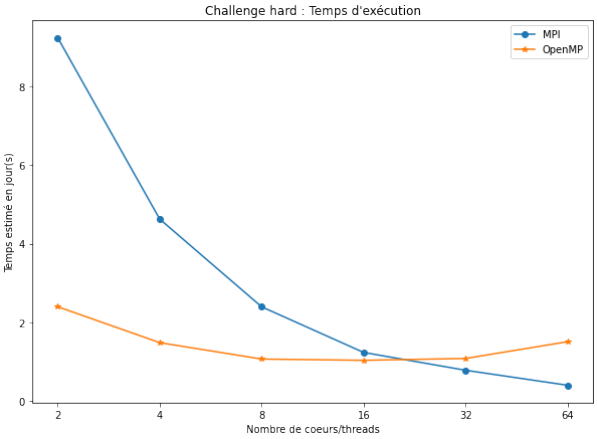


(a) Temps d'exécution

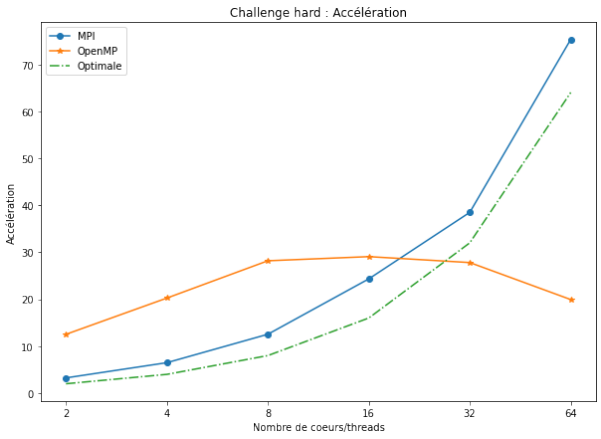


(b) Accélération

FIGURE 3 – Challenge **medium** (24 heures en séquentiel)



(a) Temps d'exécution



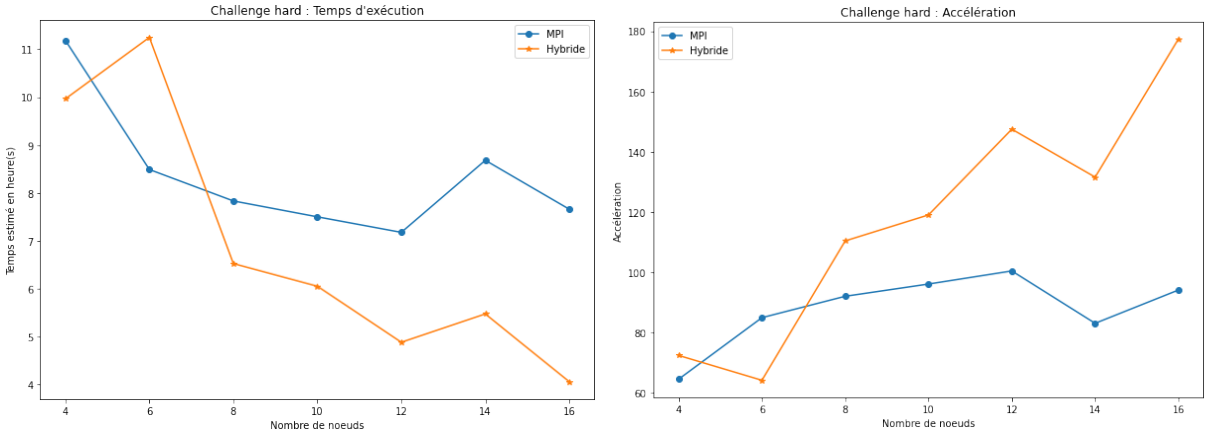
(b) Accélération

FIGURE 4 – Challenge **hard** (30 jours en séquentiel)

Pour chacune des matrices, on observe que la version d'OpenMP commence à stagner à partir d'un certain seuil de threads. Bien plus, on obtient une dégradation des performances : en effet, à partir de **16** threads, l'accélération obtenue diminue. Par ailleurs, face à MPI, OpenMP a une meilleure accélération lors de l'utilisation d'un faible nombre de threads (resp. coeurs pour MPI) et surpasse même l'optimale. Concernant MPI, on remarque que cette implémentation **scale** très bien avec le nombre de coeurs surtout avec les matrices medium et hard au point de surpasser OpenMP pour ces mêmes matrices. Finalement, on voit bien que ces tendances sont présentes pour toutes les matrices et que l'accélération obtenue croit avec la taille de la matrice.

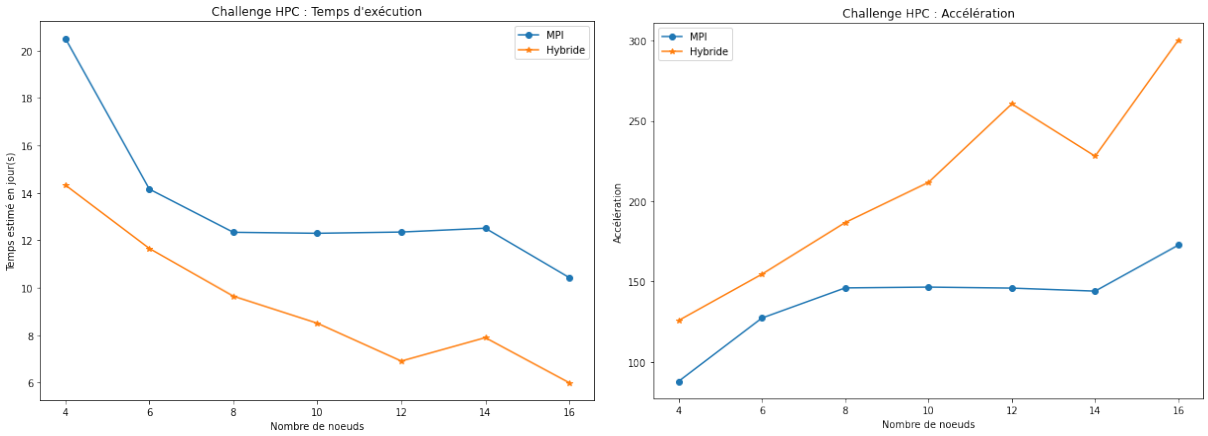
6.2 MPI vs Hybride

Étant donné que notre implémentation effectue énormément de communications, il est nécessaire d'utiliser un réseau performant pour obtenir de bons résultats [6]. Ainsi, nous allons utiliser un réseau **Omnipath** pour les benchmarks de cette partie. Par ailleurs, comme nous l'avons pu le remarquer dans la partie précédente, notre implémentation OpenMP livre de meilleures performances avec 16 threads. Par conséquent, on fixera le nombre de threads à cette valeur pour la version hybride. Finalement, nous allons utiliser uniquement les très grandes matrices, i.e, hard et HPC car il est très probable que pour les petites matrices, on obtienne très peu de différences par rapport à la partie précédente dû au temps de communication (voire une dégradation).
⇒ **CPU** : `(Intel Xeon Gold 6130 16-core Processor)` × 2



(a) Temps d'exécution (b) Accélération

FIGURE 5 – Challenge **hard** (30 jours en séquentiel)



(a) Temps d'exécution (b) Accélération

FIGURE 6 – Challenge **HPC** (5 ans en séquentiel)

6. en effet, avec un réseau Ethernet classique et 6 noeuds, on obtient un temps d'exécution de 3 jours pour la matrice medium sous MPI : les communications limitent donc l'implémentation

On observe que la version hybride performe beaucoup mieux que l'implémentation MPI pure. En effet, cela se traduit par une accélération **"linéaire"** pour la version hybride contre une **"demie-parabole"** (i.e, l'allure de la fonction racine carrée) pour MPI. Cela s'explique par la réduction des communications dans la version hybride, remplacées par une couche de parallélisme beaucoup plus rapide via OpenMP. En effet, sous MPI, chaque coeur de chaque CPU est considéré comme un noeud, ce qui mène à énormément de communication. Finalement, on remarque qu'il y a certaines fluctuations menant à des différences de l'ordre de quelques jours.

7 Conclusion

Matrice	**Meilleure(s) version(s)**	**Séquentiel**	**Parallèle**	**Accélération**
Easy	MPI & OpenMP	60 mins	$\approx$ 2 mins	$\approx$ 30
Medium	MPI	24 heures	$\approx$ 31 mins	$\approx$ 46.5
Hard	Hybride	30 jours	$\approx$ 4 heures	$\approx$ 180
HPC	Hybride	5 ans	$\approx$ 6 jours	$\approx$ 304

TABLE 1 – Tableau récapitulatif

En conclusion, on a réussi à implémenter différentes versions de parallélisation chacune adaptées à une taille de matrice. En effet, on remarque que pour des petites matrices, il est beaucoup plus intéressant d'utiliser OpenMP que MPI. On obtient des résultats très proches pour une version plus simple à implémenter. Bien plus, dans le cas d'une limitation de ressources, l'implémentation sous OpenMP est bien plus pertinente. Pour des matrices très grandes, il est recommandé d'utiliser la version hybride qui permet d'obtenir des accélérations allant jusqu'à 300 dans le cas de la matrice HPC. Finalement, on observe que la parallélisation est parfois primordiale pour résoudre des problèmes tels que celui présenté dans ce rapport : passer de 5 ans à 6 jours en terme de temps d'exécution permet de rendre la résolution cohérente et faisable.

8 Annexe

Dans cette annexe, on liste les commandes utilisées pour obtenir les benchmarks :

- MPI (1 noeud) : `mpiexec -np p -machinefile $OAR_NODEFILE -map-by ppr:1:core ./lanczos_modp [options]`
- OpenMP : `export OMP_STACKSIZE="1G" && export OMP_NUM_THREADS=p && ./lanczos_modp [options]`
- MPI (diff. noeuds) : `mpiexec -max-vm-size p -machinefile $OAR_NODEFILE -map-by ppr:1:core -mca mtl psm2 -mca pml ^ucx,ofi -mca btl ^ofi, openib ./lanczos_modp [options]`
- Hybride : `export OMP_STACKSIZE="1G" && export OMP_NUM_THREADS=16 && mpiexec -max-vm-size p -x OMP_STACKSIZE="1G" -x OMP_NUM_THREADS=16 -machinefile $OAR_NODEFILE -map-by ppr:1:node -mca mtl psm2 -mca pml ^ucx,ofi -mca btl ^ofi,openib ./lanczos_modp [options]`