CALCUL HAUTE PERFORMANCE

POLYTECH SORBONNE

Rendu final de projet

BERBAGUI Amine
HACHANI Ghassen

MAIN 4

3 juin 2022

Table des matières

1 Exploration des performances du code séquentiel

On cherche à explorer les performances du fichier *lanczos_modp.c*, c'est-à-dire repérer les endroits du code qui prennent le plus de temps lors de l'exécution. Pour cela, on utilise les outils présent dans le cours, à savoir **GNU Profiler** ainsi que l'outil **Perf Tools**.
On a donc :

- ✔ GNU Profiler :
  - 69% du temps dans **sparse_matrix_vector_product**
  - 20% du temps dans **orthogonalize**
  - 11% du temps dans **block_dot_products**
- ✔ Perf Tool :
  - 81% du temps de **sparse_matrix_vector_product** à la ligne 283
  - 29% du temps de **orthogonalize** à la ligne 299
  - 44% du temps de **block_dot_products** à la ligne 312

➔ Ligne 283 : `y[i * n + l] = (a + v * b) % prime;` ⟶ fonction sparse_matrix_vector_product
➔ Ligne 299 : `C[i * n + j] = (x + y * z) % prime;` ⟶ fonction matmul_CpAB
➔ Ligne 312 : `C[i * n + j] = (x + y * z) % prime;` ⟶ fonction matmul_CpAtB

Ainsi on a identifié les hots-spots du programme et l'on peut maintenant nos différentes méthodes de parallélisation sur ces 3 fonctions de sorte à réduire le temps d'exécution avec une matrice donnée.

Avec la matrice **TF16** par exemple, on a séquentiellement un temps d'exécution d'environ 1m30s. On va donc essayer par la suite d'optimiser au maximum ce temps. On passera ensuite aux matrices de benchmarks (easy,medium,hard) pour évaluer nos résultats.

2 Optimisation du code séquentiel

On cherche maintenant à optimiser le code séquentiel avant de commencer le travail de parallélisation. Comme indiqué sur le sujet, le code peut être optimisé de plusieurs manières. L'une d'entre elle concerne le modulo **prime** qui est répeté dans les 3 lignes de commandes citées plus haut. Après les éclaircissements que vous nous avez gentillement fournies à la fin d'une séance de TD, on a quelque peu modifié les fonctions `matmul_CpAB` et `matmul_CpAtB`. En effet, ces deux fonctions effectuent des multiplications de matrices que l'on peut représenter comme suit :

$$C[i,j] = (C[i,j] + A[i,k].B[k,j]) \ \% \ \textbf{prime}, \text{ or on sait que}$$
$$A[i,k] < \textbf{prime} \text{ et } B[k,j] < \textbf{prime}$$
$$\iff C[i,j] < n.(prime-1)^2 \text{ or } n.(prime-1)^2 < 2^{64} \text{ qui est la taille maximale d'un entier 64 bits.}$$

Ainsi, on peut se passer du % **prime** dans notre 3ème boucle for,définir dans la 2ème boucle for la variable x car elle ne dépend pas de k et rajouter $C[i,j]$ = x % **prime** en dehors de la 3ème boucle pour ne plus avoir à faire n opérations % **prime**.

Ainsi, en terme de performance avec la matrice TF16, on passe de 1m30 à 1mn10s ce qui n'est tout de même pas négligeable. Et cela avant la parallélisation du programme.

3 Parallélisation OpenMP

Passons maintenant à la parallélisation avec OpenMP seulement. Nous allons décrire brièvement les commandes ajoutées dans les fonctions concernées ainsi que les performances constatées.

3.1 Description du code

1. **block_dot_products** : cette fonction contient initialement 4 boucles for, 2 d'initialisation à 0 des matrices vtAv et vtAAv et 2 autres de calcul. Dans un premier temps, on a regroupé au sein d'une même boucle `for` l'initialisation des tableaux `vtAv` et `vtAAv`. Ensuite on garde les 2 boucles `for` appelant la fonction `matmul_CpAtB` au dessus desquelles on rajoute `omp parallel for` sans oublier la réduction sur le tableau correspondant de taille n*n. Enfin, on rajoute une dernière boucle effectuant la réduction modulaire des deux tableaux sans laquelle on aurait une erreur.
2. **orthogonalize** : pour cette fonction, on réserve une équipe de threads sur les 5 dernières boucles for qui s'occupent de calculer la prochaine valeur de v et de p et qui peuvent donc être faites en parallèle. Donc on place tout bêtement un `#pragma omp for` avant chacune de ces boucles.
3. **SMVP** : Dans cette fonction, le but est de paralléliser cette boucle `for` imbriquée. En regardant le fonctionnement de cette boucle, on voit que l'on a besoin d'une réduction sur le tableau y car chaque thread traite une portion du tableau. Comme l'on a des assignations de variables à l'intérieur de la boucle, on les passe en `private` dans notre directive openmp. De plus, comme chaque itération a un temps d'exécution relativement proche, on rajoute un `schedule(static)` même si OpenMP le fait par défaut. A ce stade, la tableau y contient des lignes non modulo `prime`, c'est pourquoi on a une boucle faisant la réduction modulaire sur chaque ligne de y.

4. **block_lanczos** : dans cette fonction, on peut paralléliser quelques boucles comme la boucle for d'initialisation de Av,v,p et tmp. La boucle qui suit également bien que vous nous avez dit qu'elle dissimulait un conflit car `random64` lit/écrit des variables globales. Toutefois, on en tient pas rigueur ici, et enfin la dernière boucle de la fonction où l'on copie les valeurs de tmp dans v, donc là aussi une boucle totalement parallélisable.

3.2 Cas pour p = 1073741789

Comme vous l'aviez spécifié dans le rapport de mi-parcours, si l'on utilise un modulo très grand comme celui-ci, le programme ne fonctionne plus. En particulier car dans la fonction **SMVP**, lors de la réduction, on somme les résultats obtenus de chaque thread ce qui donne un overflow car on dépasse le seuil de $2^{32} - 1$. Pareil pour **block_dot_product**. C'est pour quoi on crée un tableau (tab) d'entier 64 bits nous permettant de ne pas déborder et ainsi de travailler avec cette valeur. Cependant, comme les temps d'exécutions étaient légèrement plus élevés, on a préféré mettre cette solution en commentaire et garder notre modulo de base.

3.3 Performances obtenues

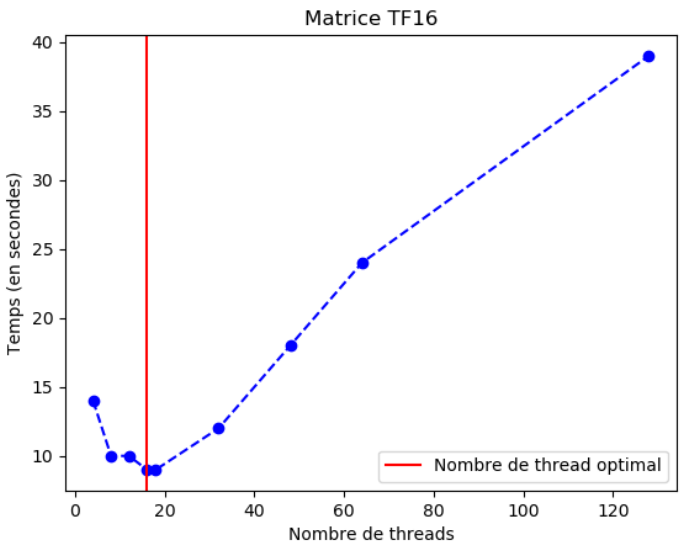
Regardons maintenant les résultats obtenus avec cette parallélisation OpenMP. Pour cela, on réserve un cluster assez puissant comme **grouille** à Nancy avec 64 coeurs et 128 threads On peut regrouper ces résultats sous la forme d'un joli tableau.
Rappelons que séquentiellement, on a :

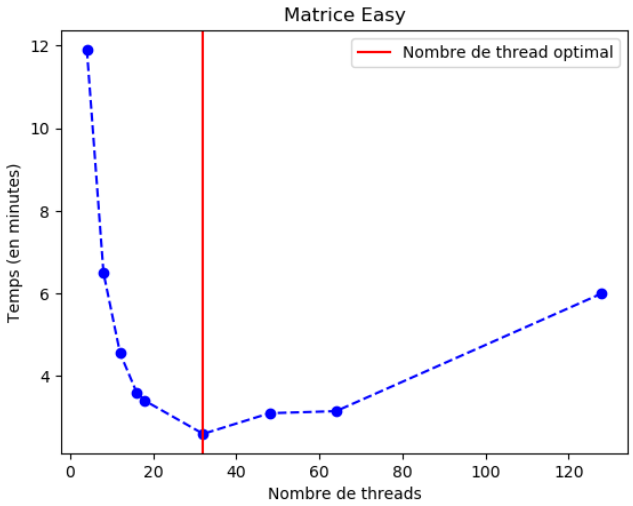
Matrices	**Temps d'exécution**
TF16	1mn 30s
Easy	1h10mn
Medium	≈ 1j
Hard	≈ 1 mois
HPC	≈ 5 ans

Avec la parallélisation sous ces ressources, on obtient :

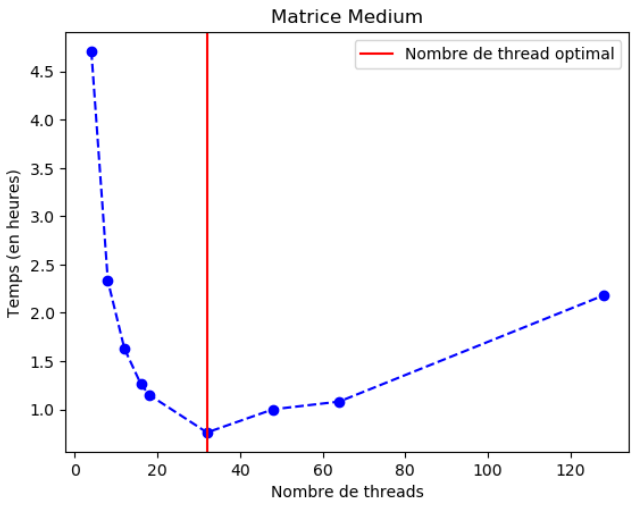
Matrices	**Temps d'exécution**
TF16	32s
Easy	4mn 42s
Medium	2h 15mn
Hard	1j 5h
HPC	32j 22h

On remarque une très nette optimisation du temps d'exécution. La différence se voit d'autant plus que la taille augmente. Sur la matrice HPC, on passe de 5 ans à 38j soit une réduction d'environ 98% du temps d'exécution.
On cherche à présent a analyser le temps d'exécution de chaque matrice en fonction du nombre de threads et ainsi trouver le nombre n_opti de threads optimal pour chaque matrice, c'est-à-dire le nombre de threads minimum et suffisant à réduire au maximum le temps d'exécution.

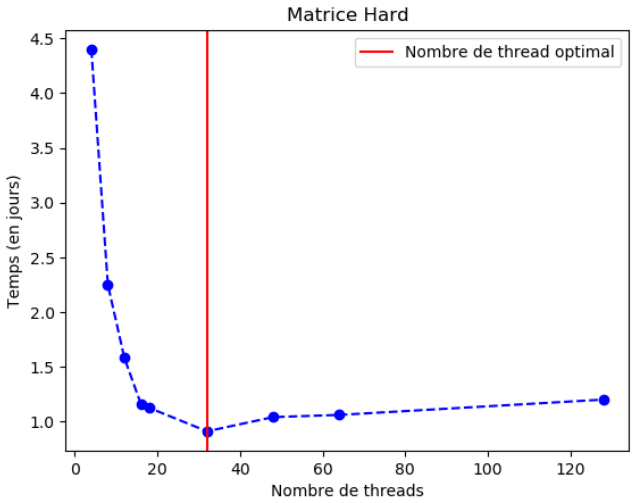




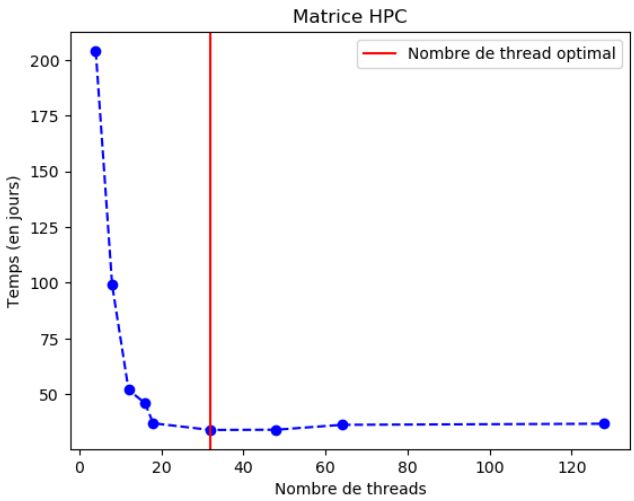
(a) Matrice Easy



(b) Matrice Medium



(a) Matrice Hard



(b) Matrice HPC

Matrices	n_opti	Temps d'exécution	Accélération
TF16	18	9s	$\approx \times 10$
Easy	32	2mn 40s	$\approx \times 20$
Medium	32	45mn	$\approx \times 32$
Hard	32	21h 40mn	$\approx \times 33$
HPC	32	32j	$\approx \times 57$

On voit finalement qu'avec le bon nombre de threads pour chaque matrice, on obtient un temps encore plus petit. Pour nous et sur ce serveur de calcul, le nombre optimal était toujours $n\_opti = 32$ sauf pour TF16 car elle est petite. On remarque également qu'à partir de ce n_opti, le temps d'exécution augmente à nouveau rapidement pour les petites matrices et assez lentement pour hard et HPC

4 Parallélisation MPI

On passe maintenant à la parallélisation MPI. Cette méthode se base sur une communication entre des noeuds exécutant des programmes parallèles sur des systèmes à mémoire distribuées. Nous avons utilisé une décomposition 1D de la matrice entre les différents processus. Les différents `MPI_Barrier` sont la par sécurité, on ne les détaillera pas plus bas.

4.1 Description du code

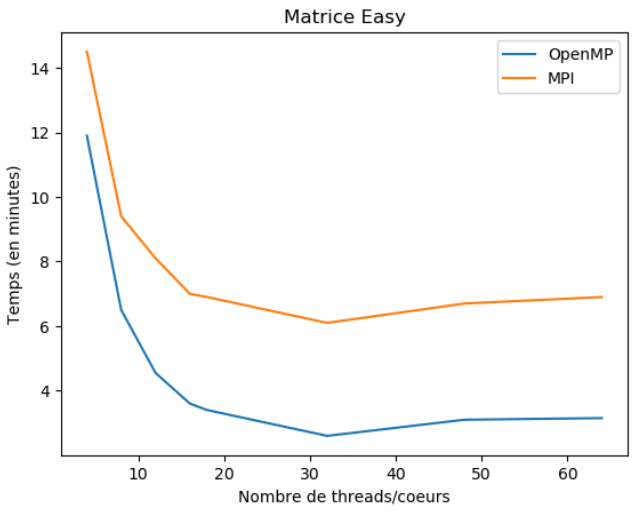
- **main** : On initialise le nombre de processus et son rang puis on charge la matrice et son nombre de coefficient non-nuls `nn`. On envoie `nn` à chaque processus puis on détermine le reste des éléments après avoir divisé à part égale le nombre d'éléments pour chaque noeud (rm). On calcul le nombre d'élément par noeud (`nn_local`) ainsi que l'indice de début de la matrice initial qu'aura chaque processus selon si le rang du processus est plus petit que le reste `rm`.
  On crée deux tableaux statique :`displs` et `sendcnts` représentant les paramètres de `MPI_Scatterv`. Chaque valeur i de `displs` représente le déplacement (par rapport à sendbuf) à partir duquel prendre les données sortantes pour traiter i. `sendcnts` spécifie la taille des blocs à envoyer. Ces tableaux vont nous servir à distribuer la matrice. On alloue la mémoire pour la matrice local **M1** et on diffuse le nombre de lignes, de colonnes, et de coefficients non-nuls de **M1** à tous les autres processus.
  On distribue maintenant la matrice à l'aide de la fonction `MPI_Scatterv` qui permet contrairement à `MPI_Scatter` d'envoyer un nombre différent d'élément à chaque processus mais surtout d'ignorer certaine portions du tableau (*overlap*). On se sert de nos deux tableaux en paramètre. Enfin, chaque processus appelle la fonction `block_lanczos` avec son `nn_local` en paramètre correspondant. Finalement, si le rang du processus vaut 0, on sauvegarde le résultat final.

- **SMVP** : On rajoute `nnl` en paramètre correspondant au `n_local` du main. On crée la matrice **y1** pour chaque processus. On récupère le rang de chaque processus comme on l'a pas en paramètre puis on laisse la boucle `for` imbriquée comme elle était. Une fois finie, on utilise la fonction `MPI_AllReduce` pour regrouper les résultats de chaque processus entre eux (`y`) et les distribuer à tous les processus (`y1`). On effectue la réduction modulaire de y1 que l'on stock dans y.

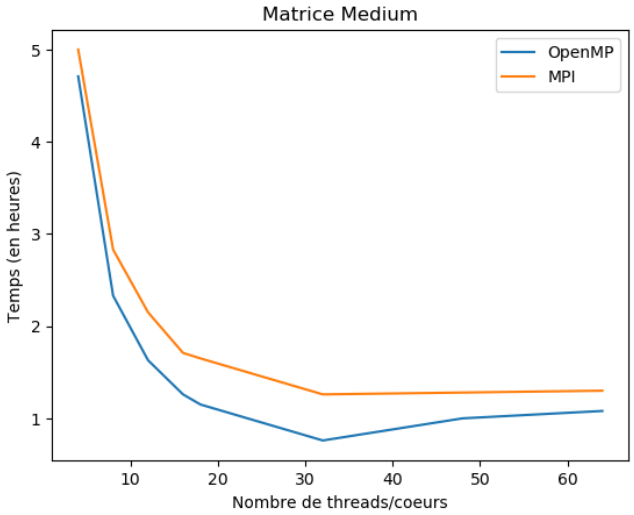
- **block_lanczos** : A partir de la boucle `while`, chaque processus calcule les fonctions **SMVP** avec leur `nn_local` respectif. Le processus de rang 0 calcule les fonctions nécessaire à l'algorithme ainsi que la fonction de correction puis envoie aux autres processus l'état de la variable stop, si elle vaut `true`, on sort de la boucle, sinon on envoie le tableau **v** à jour aux autres processus puis on réitère. Une fois sorti de la boucle, le processus de rang 0 appelle la fonction `final_check` pour vérifier les assertions puis on libère la mémoire des tableaux.

4.2 Performances obtenues

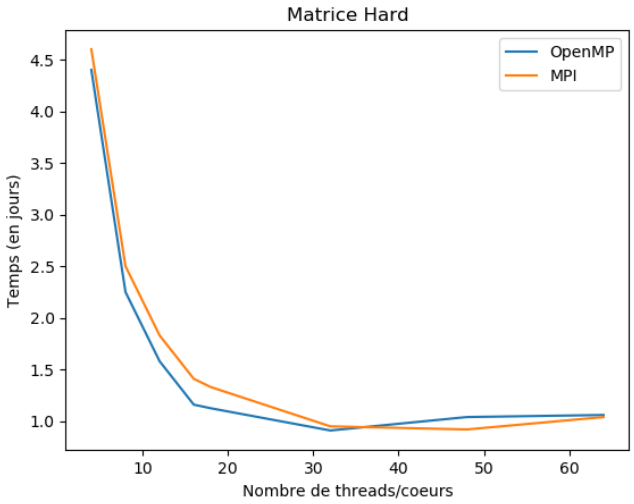
Similairement à OpenMP, sur le même noeud **grouille** (AMD EPYC 7452, 32 cores/CPU, 128GB RAM, 1788GB SSD, 1 x 25Gb Ethernet), on fait varier le nombre de coeurs avec `-np` pour trouver le temps optimal et ainsi comparer avec OpenMP. Graphiquement, on a :



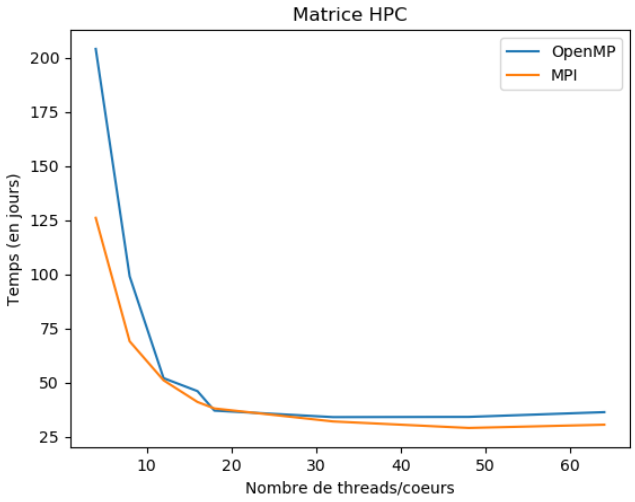
(a) Matrice Easy



(b) Matrice Medium



(a) Matrice Hard



(b) Matrice HPC

Matrices	valeur np optimale	Temps d'exécution	Accélération
Easy	48	6mn 40s	$\approx \times 10$
Medium	48	1h 15mn	$\approx \times 21$
Hard	48	22h 20mn	$\approx \times 32$
HPC	48	29 j	$\approx \times 63$

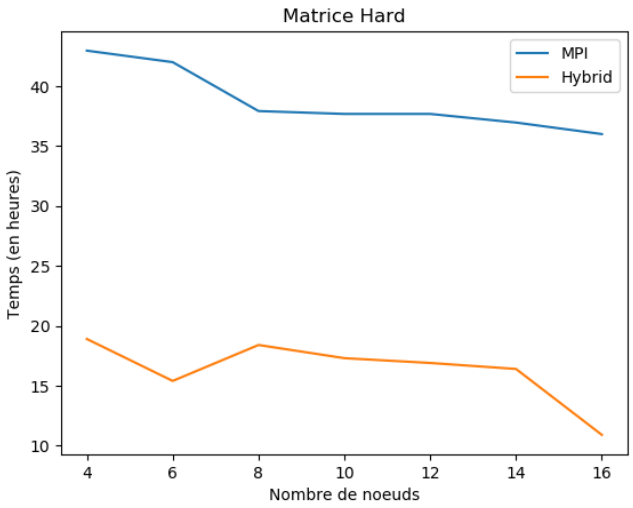
On remarque que plus la taille de la matrice augmente, plus la version MPI surpasse la version OpenMP. Ce dernier est beaucoup plus efficace pour les petites matrices mais lorsqu'on atteint une taille suffisamment grande et à partir de $np = 32$, MPI se rapproche de plus en plus d'OpenMP voire fait mieux que lui pour la matrice Hard/HPC.

5 Parallélisation hybride : MPI & OpenMP

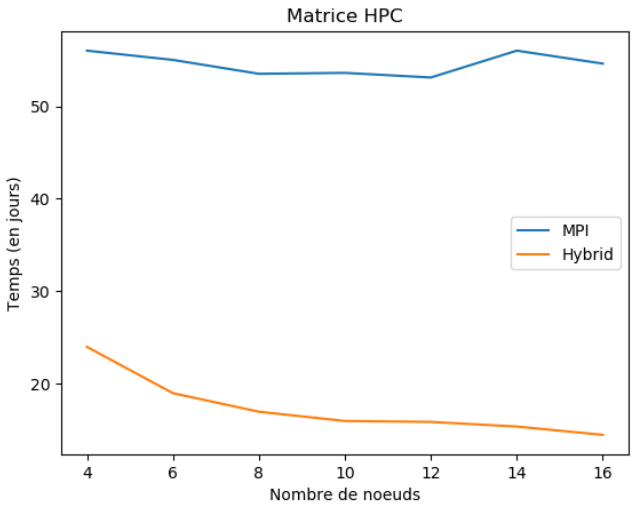
Finalement, dans cette dernière parallélisation, on cherche à combiner nos deux méthodes vues jusqu'à là dans le but d'avoir un seul processus MPI par noeud et du multi-threads au sein de chaque noeud. L'implémentation de cette version hybride a pour avantages de réduire la mémoire nécessaire et de réduire drastiquement le temps d'exécution comme l'on a plusieurs noeuds.
Au niveau du code, les explications ne sont pas nécessaires car la version hybride repose sur la version MPI avec les directives OpenMP.

5.1 Performances obtenues

Comme cette version hybride utilise plusieurs noeuds, on a beaucoup plus de communications qui se font. Ainsi, le débit de données est limité à 10Gb sur la plupart des noeuds grid5000 ce qui freine les performances. Pour palier à cela et pouvoir optimiser nos communications au maximum, on utilise le réseau de communication **Omni-path** à 100Gb ce qui va inévitablement accroître les performances. Sur grid5000, on fixe le nombre de threads à 32 (nombre optimal pour OpenMP) et on fait varier le nombre de noeuds de 4 à 16 par pas de 2.



(a) Matrice Hard



(b) Matrice HPC

Matrices	Nb noeud optimal	Temps d'exécution	Accélération
Hard	16	10h 50mn	$\approx \times 65$
HPC	16	14 j	$\approx \times 130$

Comme on pouvait le prévoir, la version hybride est beaucoup plus performante pour de très grandes matrices. On arrive à réduire le temps d'exécution avec HPC de 29j à 14j soit deux fois moins.

6 Conclusion

En conclusion de ce projet, on a pu appliquer différentes méthodes de parallélisation très connues, qui ont bien fonctionné puisqu'on arrive assez bien à réduire le temps d'exécution de nos matrices pour l'algorithme de block-Lanczos. Pour les petites matrices, on remarque que la parallélisation OpenMP est bien plus efficace et qu'à l'inverse, plus la matrice est grande, plus la version MPI est meilleure et l'utilisation de communicateurs est judicieuse.

Enfin on a pu implémenté la version hybride des deux qui logiquement est beaucoup plus performante surtout avec le réseau Omni-path. D'un calcul séquentiel de 5 ans avec la matrice HPC, on passe à un calcul parallèle de 14 jours.

Toutefois, la parallèlisation MPI du code se fait uniquement sur la fonction `SMVP`. En parallélisant également les fonction `block_dot_product` et `orthogonalize`, on peut penser avoir des temps encore plus faible.

7 Annexe

Commandes Grid5000 utilisés pour la réservation des serveurs :

- OpenMP & MPI : oarsub -t exotic -p grouille -l host=1 -I
- Hybrid avec Omni-path : oarsub -p dahu -l host=6 -p "opa_rate=100" -I

Commandes Grid5000 utilisés pour l'exécution du code :

- OpenMP : export OMP_NUM_THREADS= puis exécution classique ./lanczos_modp ...

- MPI : `mpiexec -max-vm-size 8 -x OMP_NUM_THREADS=32 -machinefile $OAR_NODEF ILE -mca mtl psm2 -mca pml ^ ucx,ofi -mca btl ^ ofi,openib ./lanczos_modp -matrix Hard -prime 65537 -n 2`

- Hybrid : `mpiexec -npernode 1 -max-vm-size 4 -x OMP_NUM_THREADS=32 -machinefile $OAR_NODEF ILE -mca mtl psm2 -mca pml ^ ucx,ofi -mca btl ^ ofi,openib ./lanczos_modp -matrix Hard -prime 65537 -n 2`