Rapport de projet HPC/ Algorithme de block-Lanczos modulo p

Projet HPC / Algorithme de block-Lanczos

Professeurs référents : Charles Bouillaguet et Vincent Neiger

SORBONNE UNIVERSITÉ
Master 1 Sécurité Fiabilité et Performance du Numerique (SFPN)

Le 20/05/2022

Amadou BAYO et River MOREIRA FERREIRA

Chapitre 1

Profiling

Pour commencer , nous avons d'abord analysé le code séquentiel pour savoir quelle partie devrait être optimisée. Grâce au profiling nous avons repéré les parties du code qui prenaient beaucoup plus de temps à s'exécuter, vous pouvez voir le résultat sur le tableau ci-dessous. Donc nous avons décidé de paralléliser ces parties du code. En séquentiel le code prend environ **659.8s** sur grid5000 sur le site de nancy avec la matrice TF17.

Fonction	**Pourcentage**
sparse_matrix_vecteur_product	51,22 %
orthogonalize	28,05 %
block_dot_products	11,92 %

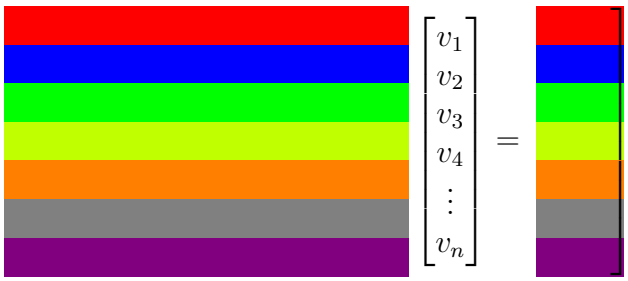
Chapitre 2

Parallélisation et résultats obtenus

2.1 Choix des algorithmes

Nous avons opté pour une stratégie où tous les processus exécutent l'algorithme mais seul le processus principal c'est à dire le processus 0 détient toutes les informations, à chaque étape de l'algorithme il fait une répartition **1D** des données et récupère les résultats une fois le calcul terminé. Pour la lecture de la matrice , seul le processus principal lit le fichier et partage la matrice aux autres processus, ceci évite toute inondation du système de fichier.

En prenant l'exemple sur le produit matrice vecteur, on repartit les n lignes de matrices entre les différents processus de manière contiguë, donc **n/p** (où p est le nombre de processus) blocs par processus. Quant au vecteur , il est partagé entièrement à tous les processus. Si **n%p != 0** , on désigne un processus pour calculer le reste du bloc.



NB : Dans notre implémentation le partage de la matrice en bloc se fait dès le début de la lecture du fichier contenant la matrice avant même le début de l'algorithme, ceci à pour avantage de corriger de nombreux bugs.

En terme de Communication, nous avons plutôt choisi des communications collectives qui semblaient être les mieux adaptées. Au début de l'algorithme

chaque processeur détient déjà son bloc de matrice, donc pour la première partie de la parallélisation c'est à dire la fonction "produit matrice vecteur", le processus 0 **broadcaste** le vecteur V à tout le monde et une fois le produit terminé, il fait un **Allreduce** pour mettre les valeurs à jour. Pour la deuxième étape de la parallélisation (la fonction "block_dot_products"), le processus principal partage les données en faisant un **scatter** et fait un **reduce** une fois le calcul terminé. Et pour la dernière étape de la parallélisation c'est à dire la fonction "orthogonalize" , le processus 0 broadcaste toutes les données nécessaires pour la parallélisation de cette fonction , ensuite fait un **gather** afin de rassembler toutes les données du calcul. on a pas oublié de broadcaster la variable stop pour que les processus sachent quand s'arrêter, et le calcul reprend.

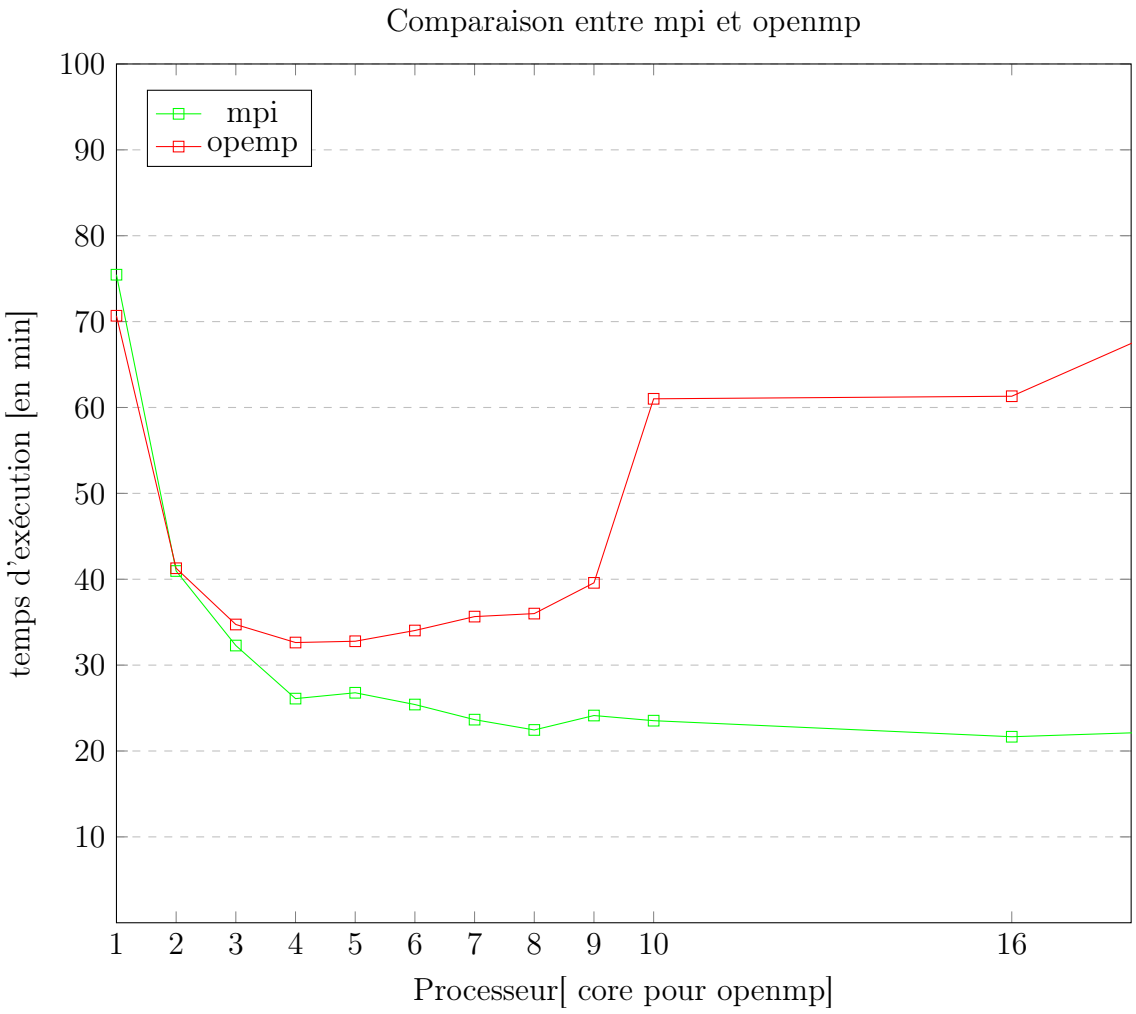
2.2 Résultats obtenus

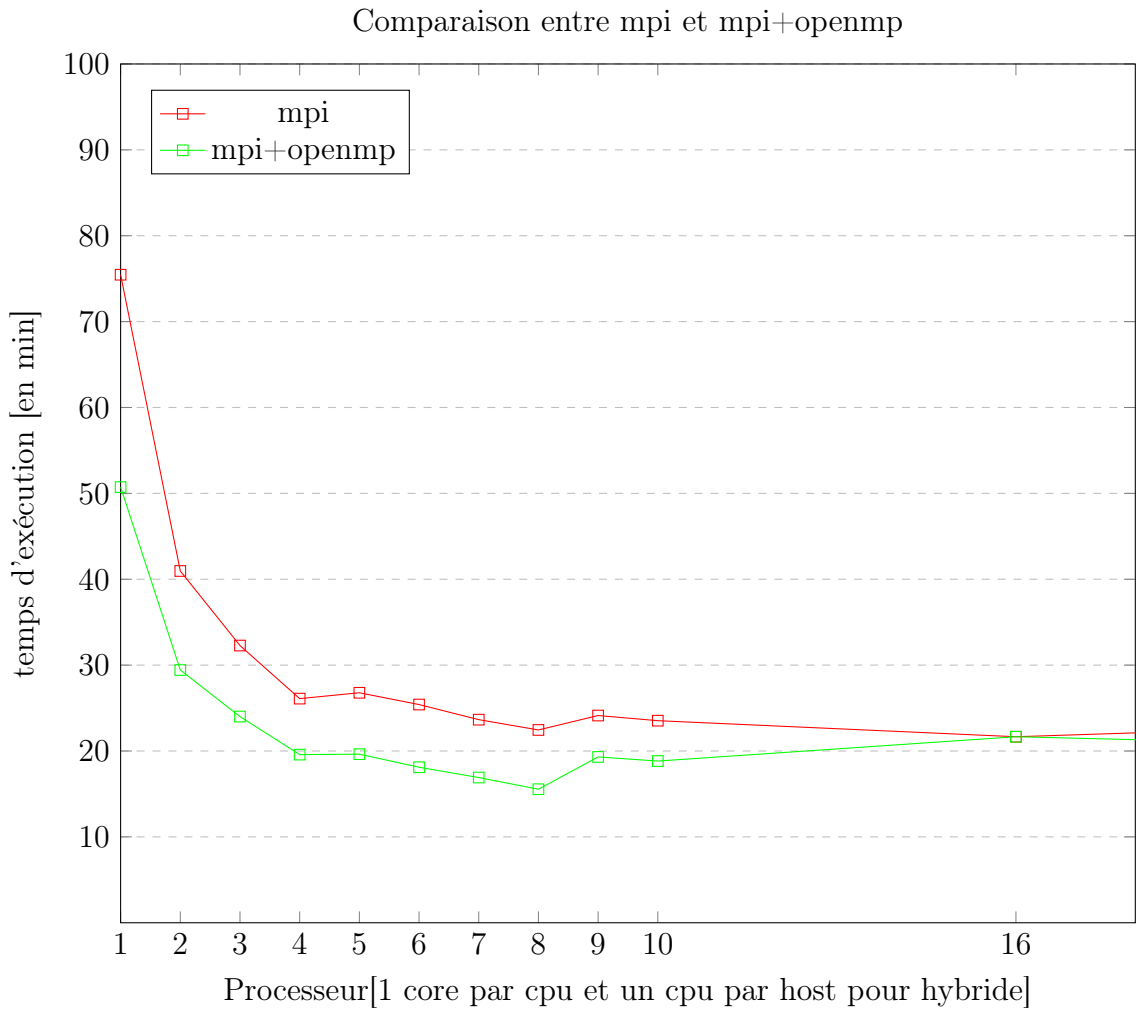
Nous avons réalisé les tests sur grid5000 sur le site de Nancy et voici les résultats que nous avons obtenu sur la matrice **TF18** avec les paramètres suivants pour l'exécution " –prime 65537 –right –n 2 –output kernel.mtx " :

MPI				**OpenMP**			
processor	time(min)	speedup	efficiency	core	time(min)	speedup	efficiency
1	75.46	0.97	0.97				
2	40.96	1.80	0.90	2	41.28	1.78	0.89
3	32.28	2.28	0.16	3	34.73	2.12	0.70
4	26.10	2.82	0.70	4	32.63	2.26	0.56
5	26.78	2.75	0.55	5	32.78	2.25	0.45
6	25.41	2.90	0.48	6	34.03	2.16	0.36
7	23.65	3.11	0.44	7	35.65	2.06	0.29
8	22.45	3.28	0.41	8	36.00	2.04	0.25
9	24.13	3.05	0.33	9	39.58	1.86	0.20
10	23.53	3.13	0.31	10	61.01	1.20	0.12
16	21.66	3.40	0.21	16	61.31	1.20	0.07
32	25.28	2.91	0.09				
64	30.28	2.43	0.03				

Hybride				
processor	core	time(min)	speedup	efficiency
3	1	24.01	3.07	1.02
4	1	19.58	3.76	0.94
5	1	19.63	3.75	0.75
6	1	18.11	4.07	0.67
7	1	16.91	4.36	0.62
8	1	15.55	4.74	0.54
8	2	17.75	4.15	0.51
8	3	18.08	4.07	0.50
9	1	19.31	3.81	0.19
10	1	18.83	3.91	0.39
16	1	21.66	3.4	0.21

processor : le nombre de processeurs sur lequel se passe l'exécution
core : nombre de coeurs sur lequel se passe l'exécution
time : le temps d'exécution est exprimé en minute
speedup : le rapport entre le temps séquentiel et le temps parallèle
efficiency : le rapport entre le speedup et la quantité de ressource
Pour rappel, le code séquentiel s'exécute sur la matrice TF18 en **73.76 minutes**


Comparaison entre mpi et openmp
mpi
opemp
temps d'exécution [en min]
100
90
80
70
60
50
40
30
20
10
1
2
3
4
5
6
7
8
9
10
16
Processeur[core pour openmp]



2.3 Commentaires

- **mpi :** Comme vous pouvez voir sur le tableau, nous obtenons des gains de performance avec mpi jusqu'à 16 processeurs ,où nous avons eu la meilleure performance en temps et accélération avec 21.66min de temps d'exécution (version séquentielle 73.76min) , et nous atteignons une accélération de 3.40 . En terme d'efficacité avec au plus un processeur nous atteignons sur 2 cpus avec une valeur de 0.97 . Au delà de 16 processeurs avec mpi nous commençons à perdre en performance comme vous pouvez voir dans le tableau ci-dessus. Donc notre limite de parallélisme est fixée à 16 processeurs, et il y a aucun intérêt d'utiliser plus de 16 processeurs pour paralléliser ce projet avec mpi uniquement.

- **OpenMP :** Quant à openmp , nous avons enregistré la meilleure performance avec 4 coeurs où nous atteignons 32.63 min de temps d'exécution et 2.26 d'accélération. Nous perdrons énormément quand on essaye d'aller au-delà de 4 coeurs, Donc notre limite de parallélisme avec openmp uniquement est seulement 4 coeurs.
- **mpi+OpenMP :** Pour mieux profiter de la parallélisation hybride , nous avons utilisé un cpu par host et en faisant du multithreading à l'intérieur. Nous avons testé plusieurs scénarios possibles, et nous avons eu le meilleur gain avec 8 processeurs et 1 coeur par cpu. Comme vous pouvez remarquer sur le tableau ci-dessus , nous perdrons en performance lorsqu'on essaye d'augmenter le nombre de coeurs ou qu'on essaye d'aller au delà de 8 processeurs. Nous avons eu de très bon résultat avec la programmation hybride car nous atteignons des accélérations jusqu'à 4.74 et un temps d'exécution d'une quinzaine de minutes à peine.

2.4 Comparaison

Comme on peut voir sur le graphique, nous avons eu de meilleure performance avec mpi qu'openmp. Cette différence devient très énorme lorsque qu'on essaye d'aller au delà de 4 threads.

Avec la programmation hybride , nous avons eu de meilleur résultat que mpi uniquement . Même en augmentant le nombre de processeur , cette différence se fait sentir comme on peut voir sur le graphique.

Nous pouvons retenir que programmer avec mpi ou openmp c'est bien, mais faire de la programmation hybride c'est encore meilleur.

Chapitre 3

Checkpointing

En cas de panne réseau, plantage, coupure électrique ... , nous avons mis en place un système capable de continuer l'exécution où le programme s'est arrêté lors de l'interruption sans pourtant avant besoin de tout recommencer à zéro.

Pour mettre en oeuvre ce système, nous avons analysé d'abord le programme pour savoir quelles variables sauvegardées pour conserver l'état du programme à une certaine itération. Brièvement, nous avons observé quelles variables réinitialisées à chaque itération et quelles sont celles qui sont utiles d'une itération à l'autre. Du coup,au final il suffit de stocker les vecteurs v et p. On stocke chaque vecteur dans un fichier où on sauvegarde aussi sa taille et le nombre d'itérations déjà effectué. On conserve le nombre d'itérations pour pouvoir prédire le nombre d'itérations restant à la prochaine exécution. Le manuel d'utilisation du checkpoint est expliqué dans le fichier "README.txt".

NB : On fait le checkpoint toutes les 800 itérations pour mpi et à chaque itération pour openMP. Quand on relance à partir du checkpoint le nombre d'itérations déjà effectué est déduit du nombre itérations restantes.

Chapitre 4

Benchmark sur les grosses matrices

Nous savons maintenant qu'avec 8 processeurs et 1 coeur par cpu , nous atteignons la meilleure performance de notre code parallélisé. En disposant de 8 processeurs et 1 coeur par cpu, nous allons réaliser les tests sur de très grandes matrices.

Matrice	**code sequentiel**	**code parallelisé hybride**
TF17	13 min 55 s	2 min 23s
TF18	1 h 30 min 22 s	15 min 48s
TF19	10 h 14 min 20 s	1 h 48 min 24 s

Sorbonne Université
Campus Pierre et Marie Curie
4 Place Jussieu, 75005 Paris