Rapport de projet

Calcul haute performance

Rédigé par :

BENAISSA Rania

SHAO Xinan

Année universitaire 2021/2022

1 Matrices de Benchmark

Tout au long de ce présent rapport, les tests sont effectués sur:

- Des matrices de benchmark de la collection ***SuiteSparse Matrix***
- Des matrices de tailles variables que l'on a généré via le script de gestion de projet.

Les matrices considérées ont les caractéristiques suivantes :

Nom	Lignes	Colonnes	Valeurs non nulles	Type	Provenance
TF15	6334	7742	80057	combinatoire	collection SuiteSparse Matrix
TF16	15437	19321	216173	combinatoire	collection SuiteSparse Matrix
TF 17	38,132	48,630	586,218	combinatoire	collection SuiteSparse Matrix
TF 18	95,368	123,867	1,597,545	combinatoire	collection SuiteSparse Matrix
TF 19	241,029	317,955	4,370,721	combinatoire	collection SuiteSparse Matrix
IG5-18	47,894	41,550	1,790,490	combinatoire	collection SuiteSparse Matrix
medium	200,000	198,381	26,200,000	combinatoire	Générée

Table 1: Caractéristiques des matrices choisies

2 Performances du code séquentiel

Après avoir compris le fonctionnement du programme itératif du projet, notre premier réflexe fut de chercher les bouts de code où se concentrait l'essentiel du temps d'exécution pour avoir une approximation du nombre de blocs de code à optimiser.

En s'aidant de l'analyseur de performances **GNU Profiler**, nous avons testé les performances de nos fonctions sur quelques matrices avec des hyperparamètres p et n fixés respectivement à 65537 et 2 et avons constatés que le temps d'exécution se focalisait dans trois fonctions principales comme le montre le tableau suivant :

Matrice	TF15	TF16	TF17	TF18
sparse_matrix_vector_product	68.81	69.55	71.35	47.2
orthogonalize	19.53	19.39	18	32.86
block_dot_products	11.69	11.07	10.67	19.97

Table 2: Pourcentage de temps pris par chaque fonction du programme sur les matrices considérées avec n = 2 et p = 65537

3 Parallélisation avec MPI

3.1 Choix d'implementation

Compte tenu des multiplications matricielles plutôt coûteuses en temps en itératif, paralléliser l'algorithme de Lanczos par bloc nécessite que l'on découpe la matrice de départ ***M*** et le vecteur *x* avant de procéder aux calculs. En d'autres termes, nous maintenons l'utilisation telle quelle des fonctions «sparse_matrix_vector_product» , «orthogonalize» et «block_dot_products » mais sur des données de tailles réduites.

Découpage et tri de la Matrice *M*

La matrice M étant de taille $N * M$, nous avons opté dans un premier temps pour un découpage par colonnes en blocs de dimension 1 (de taille $N * m$ où m est le nombre de colonnes données à chaque processeur) avec en perspective de limiter au mieux le temps de communication entre les processeurs.

Pour se faire, nous proposons de préalablement trier la matrice M par colonnes en implémentant *une version parallèle de l'algorithme quick sort*. Nous avions par ailleurs testé d'autres algorithmes: odd even sort avec une exécution itérative du quick sort puis du bubble sort mais cela n'était pas très concluant en terme de performances en comparaison avec le quick sort parallèle qui est beaucoup plus rapide (il prenait par exemple un peu moins d'une seconde pour trier la matrice TF19).

Dans notre programme, la fonction qui se charge du découpage de M est ***subdiviseM*** et elle fonctionne comme suit:

- Le processeur désigné pour charger la matrice M (appelons le «root») répartit équitablement le nombre de colonnes par processeur et récupère le nombre de valeurs non nulles pour chaque bloc de colonnes.
- Le processeur «root» envoie les portions de la matrice M appropriées à chaque processeur à travers les routines de communication collectives « MPI_Bcast» et «MPI_scatterV » (le dernier processeur prend les colonnes restantes dans le cas où le nombre de colonnes n'est pas divisible par le nombre de processeurs disponibles).

Découpage du vecteur *x*

Dans l'algorithme de Lanczos, le vecteur x est initialement de taille $M * n$. Afin d'effectuer la multiplication matricielle $M * x$ et suite au découpage de M précédemment proposé, chaque processeur va devoir manipuler des blocs de x de taille $m * n$ comme nous l'avons vu en cours.

De même que pour le découpage de M, la fonction ***subdiviseV*** découpe le vecteur x en désignant un processeur qui se charge de créer le vecteur x qu'il répartit en blocs de taille $m * n$ sur chaque processeur via la routine «MPI_scatterV».

Calcul des produits matrice-vecteur

A la fin de notre découpage, chaque processeur a en sa possession une portion de M et une portion de x. Tous les processeurs procèdent en interne au calcul du premier produit $y \leftarrow M * x$ et les y résultants sont réduits et partagés entre les processeurs en utilisant la routine «MPI_Allreduce» (nous avons redéfini son opérateur de sorte à ce qu'il fasse le modulo p sur chaque somme).

Quant au second produit $z \leftarrow M^t * y$ il est directement calculé et stocké par chaque processeur en interne.

La fonction chargée de faire ce calcul est «computeMatrixVectorProduct».

Calcul des produits bloc-bloc

Nous procédons au calcul des produits bloc-bloc de la même manière que nous avons calculé le premier produit matrice-vecteur. C'est à dire, la fonction «computeBlocsProduct» permet à chaque processeur de calculer en interne les produits $A \leftarrow z^t * z$ et $B \leftarrow x^t * z$ qui seront ensuite réduits et partagés entre chaque processeur grâce à la routine «MPI_Allreduce».

Calcul des produits restants

Le calcul des produits restants, en l'occurrence les produits de type $x \leftarrow x + y * C$ seront directement faits à travers la fonction «orthogonalize» puisque les vecteurs x et y sont déjà parallélisés.

Vecteur Final x

Au bout d'un certain nombre d'itérations, l'algorithme finit et chaque processeur aura calculé un bloc du vecteur x final. La dernière opération parallèle est de regrouper ces blocs en un seul vecteur au niveau du processeur «root». Cette opération est faite dans la fonction «gatherFinalV» et fait appel à la routine «MPI_Gatherv».

3.2 Résultats et interprétations

Dans cette partie, nous testons les performances de notre programme parallèle «MPI_version» en calculant son temps d'exécution et son accélération en fonction du nombre de noeuds et des matrices considérées.

A cet effet, nous utiliserons le cluster «gros» du site de Nancy:

```
gros(2019): 124 nodes, Intel Xeon Gold 5220,18 cores/CPU,
96 GB RAM, 480 GB SSD, 2x25GB Ethernet
```

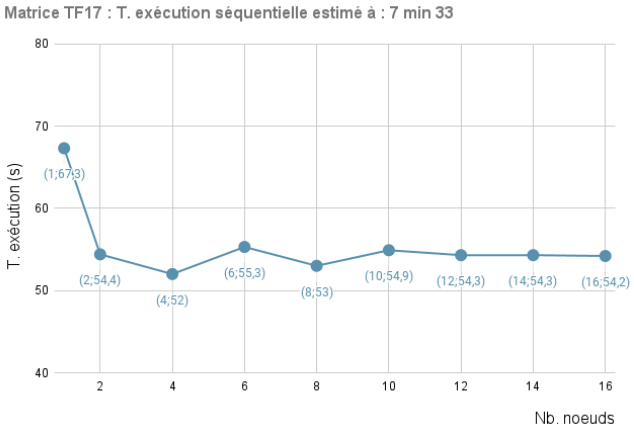
Nous allons donc tester notre programme sur 16 noeuds du cluster «gros». Pour réserver de telles ressources, nous avons exécuté la commande suivante (pour l'allocation de 16 noeuds par exemple):

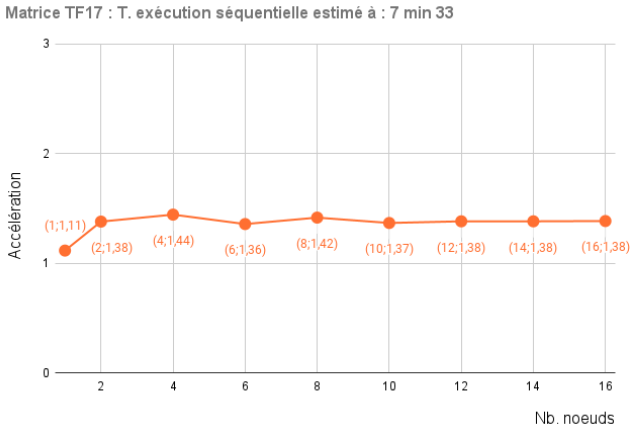
```
oarsub -p "cluster='gros'" -l /host=16/cpu=1 -I
```

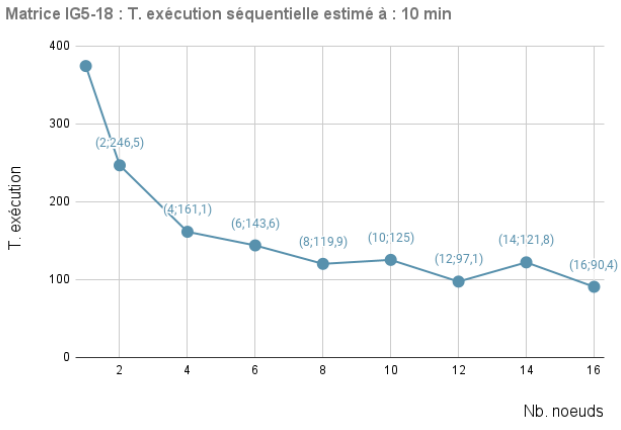
Et voici un exemple de commande pour l'exécution du programme sur la matrice TF17:

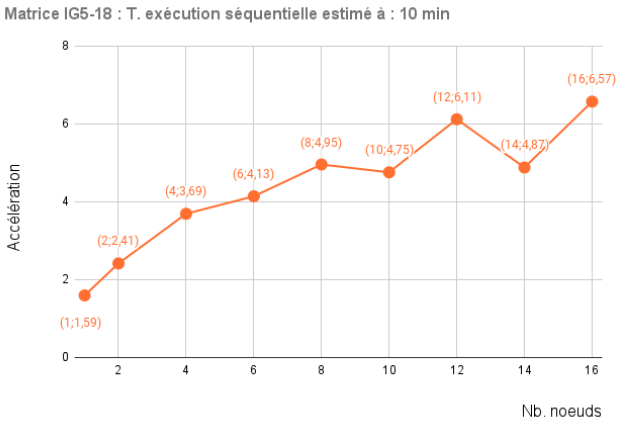
```
mpiexec --hostfile $OAR_NODEFILE ./lanczos_modp --matrix
../TF17/TF17.mtx --prime 65537 --right --n 2 --output
kernel.mtx
```

Étant donné que nous n'optimisons pas les produits matriciels lors de la parallélisation avec MPI seulement, nous fixons une petite valeur de n = 2 avec un p égal à 65537.

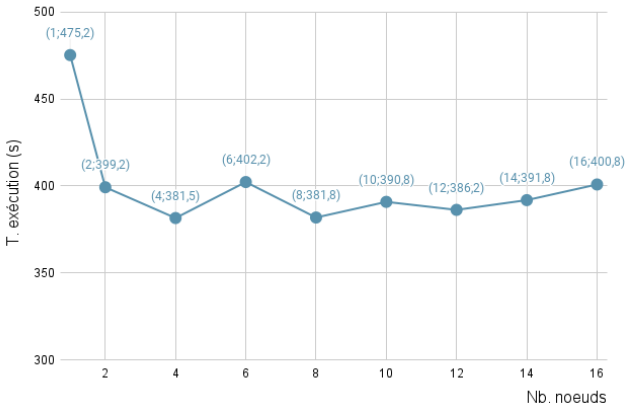




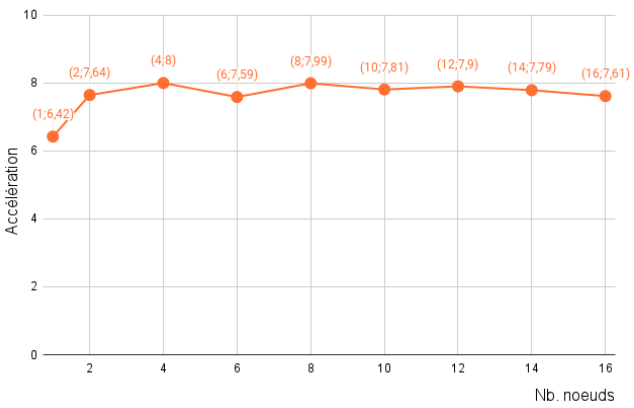




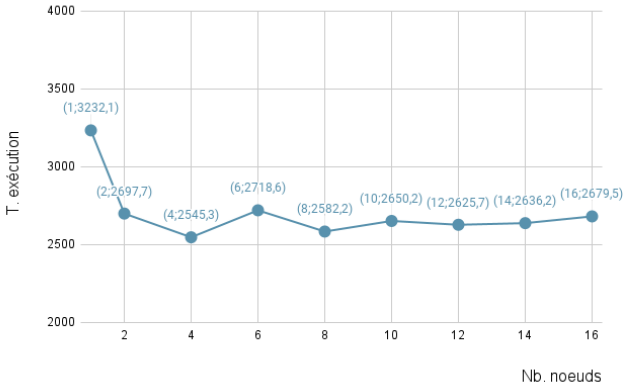
Matrice TF18 : T. exécution séquentielle estimé à : 50 min 51 s



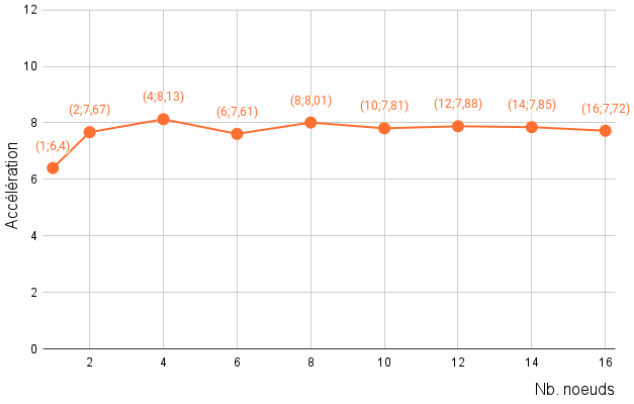
Matrice TF18 : T. exécution séquentielle estimé à : 50 min 51 s



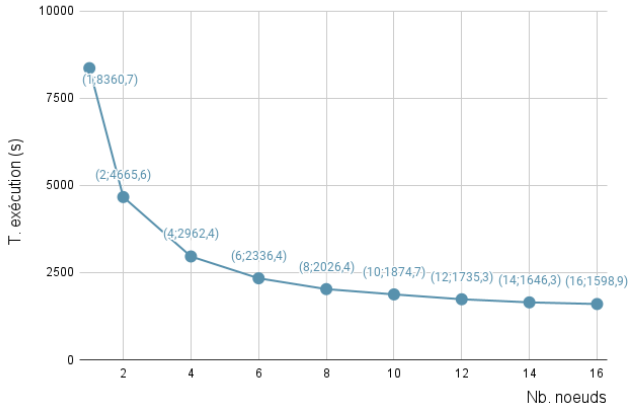
Matrice TF19 : T. exécution séquentielle estimé à : 5 h 44 min 49 s



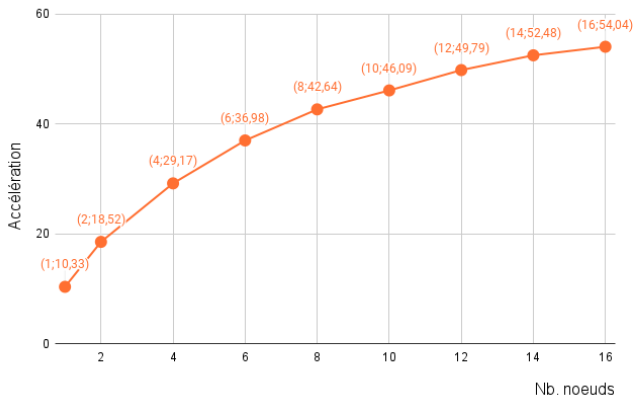
Matrice TF19 : T. exécution séquentielle estimé à : 5 h 44 min 49 s



Challenge medium : T. exécution séquentielle estimé à : 1 j



Challenge medium : T. exécution séquentielle estimé à : 1 j



Interprétation des résultats:

- Nous constatons lors de l'exécution du programme parallèle sur des matrices de petites tailles (telle que la matrice TF17) que le temps d'exécution diminuait considérablement et que l'accélération augmentait en conséquence (l'accélération reste sublinéaire) lorsque l'on prenait un petit nombre de noeuds. Au delà d'un certain seuil (Nb. noeuds = 4 soit 72 coeurs pour la matrice TF17) l'accélération commence à stagner.

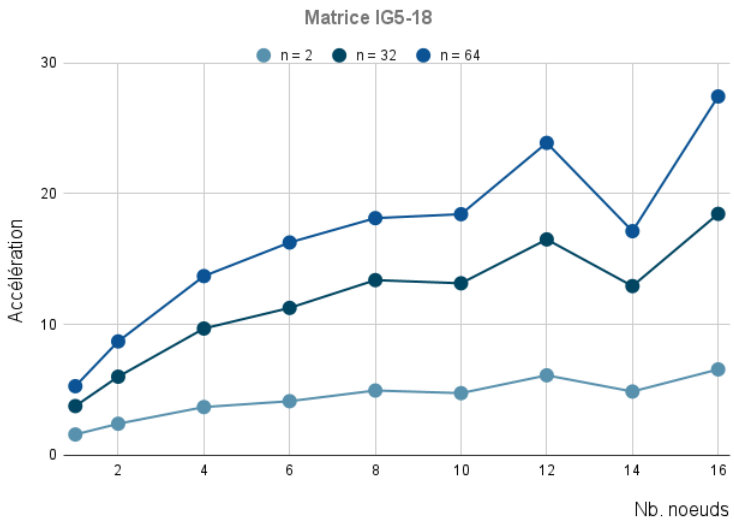
- On peut également observer le même comportement pour de grandes matrices au détail prés que le temps d'exécution (Resp. l'accélération) stagne beaucoup plus rapidement (pour de plus petits nombres de noeuds). Au bout d'un certain nombre de noeuds, on voit aussi que les performances du programme parallèle régressent (cas des matrices TF18 et TF19 au delà de 12 noeuds), on peut donc en déduire que pour ces valeurs là, le temps de communication prend le dessus sur le temps de calcul.

- On retrouve aussi quelques fluctuations sur les courbes, cela est forcement dû à notre choix de découpage de la matrice M qui fait que certains processeurs ont plus de charge de travail que d'autres (le même nombre de colonnes mais pas forcement les mêmes tailles de vecteurs de valeurs non nulles et c'est ce que l'on a constaté lorsque l'on a étudié le comportement de notre code sur la matrice TF17) et cela devient vite coûteux lorsque deux coeurs ayant une forte charge de travail et qui se trouvent sur différents CPUs veulent communiquer.

- En ce qui concerne les performances obtenues pour la matrice générée «challenge medium», on retrouve bien le seuil Nb. noeuds = 12 pour lequel les temps de calculs et de communications s'équilibrent et au delà duquel la diminution des performances commence à être observée.

3.3 Influence de la valeur de «n» sur les performances

Nous voulions connaître en pratique l'effet du paramètre n sur le temps d'exécution du programme MPI parallèle. C'est pourquoi nous avons étudié sur la matrice IG5-18, l'évolution de l'accélération en fonction du nombre de noeuds pour différentes valeurs du paramètre n:



On voit bien que plus on augmente la valeur de n, plus l'accélération augmente significativement. Lorsque l'on double la valeur de n fixé à 32, on a une augmentation de l'accélération qui est presque du même facteur ($\frac{3}{2}$). De même, entre l'accélération avec n fixé à 2 et n fixé à 64, on a un facteur de 4 ce qui n'est pas négligeable. Ces constatations nous ont donc motivé à augmenter le paramètre n pour les prochains tests.

4 Parallélisation avec OpenMP

4.1 Choix d'implementation

Lors de la parallélisation avec OpenMP, nous nous sommes focalisés sur l'optimisation des opérations de calcul des produits matriciels à savoir l'optimisation des boucles des fonctions déjà présentes dans le programme itératif or:

- ***«sparse_matrix_vector_product»*** : qui se charge du calcul des deux produits matrice-vecteur $y \leftarrow M * x$ et $z \leftarrow M^t * y$
- ***«block_dot_products»*** où l'on effectue le calcul des deux produits bloc-bloc $A \leftarrow z^t * z$ et $B \leftarrow x^t * z$

- ***«orthogonalize»***: qui s'occupe de calculer les nouvelles composantes du vecteur x

Il est a noté que toutes les fonctions appelées par ces fonctions ont également été optimisées.

4.2 Résultats et interprétations

Les tests de performances de notre programme parallèle avec OpenMp ont été fait sur un seul noeud du cluster «gros» présenté dans la section précédente. Chaque noeud de ce cluster dispose d'un CPU contenant 18 coeurs sur lequels on peut lancer deux threads chacun. Nous avons donc à disposition 36 threads pour effectuer nos tests:

```
oarsub -p "cluster='gros'" -l /host=1/cpu=1 -I
```

Comme nous calculons le temps d'exécution et l'accélération en fonction du nombre de threads, nous fixons le nombre de threads avec la commande:

```
export OMP_NUM_THREADS = 64
```

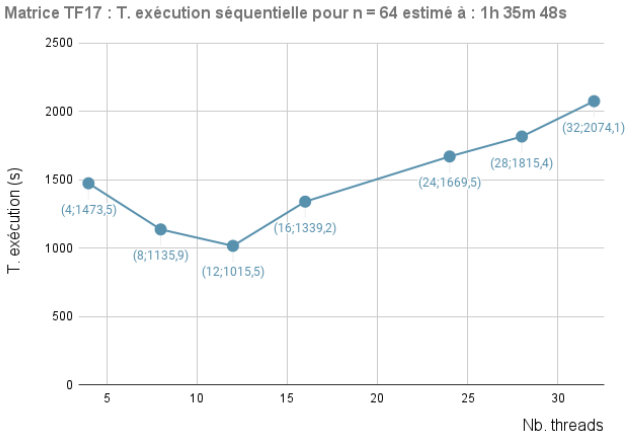
Puis nous exécutons le programme parallèle pour une matrice donnée avec les valeurs de p et n suivantes:

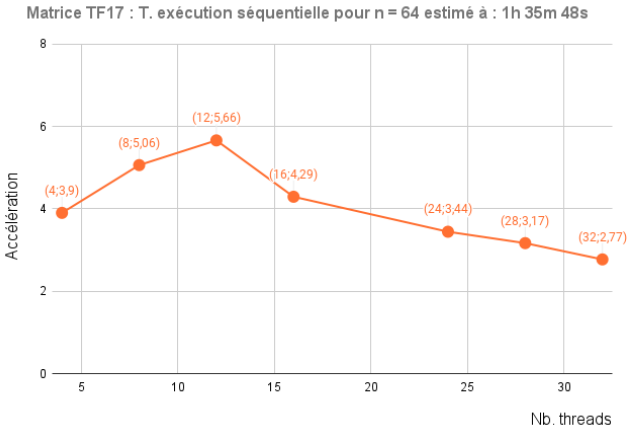
```
./lanczos_modp --matrix ../TF17/TF17.mtx --prime 65537
--right --n 64 --output kernel.mtx
```

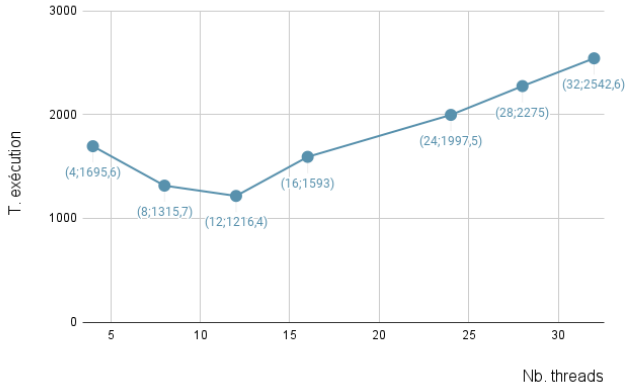
Afin de prévenir d'éventuels dépassements mémoire, nous augmentons l'espace des piles en exécutant ces deux commandes:

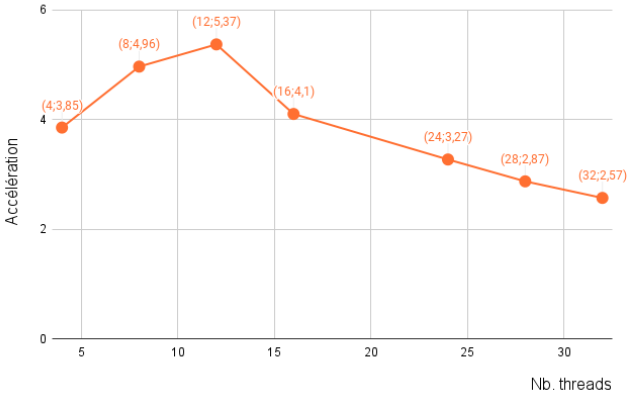
```
ulimit -s unlimited
export OMP_STACKSIZE=1000m
```

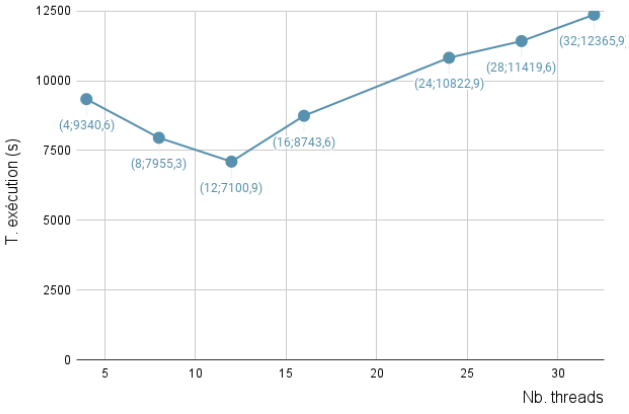
Nous présentons sur les graphes suivants, les résultats obtenus pour quelques matrices :

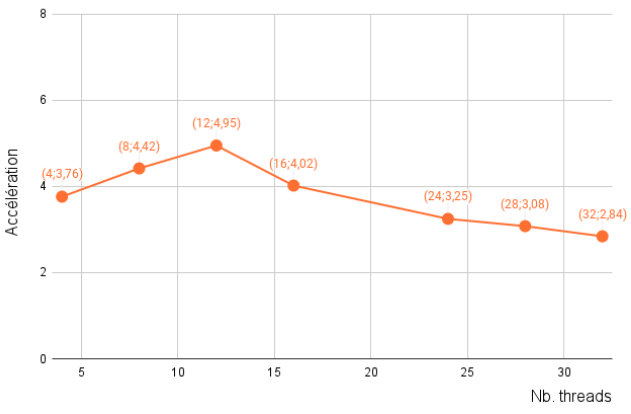

Matrice TF17 : T. exécution séquentielle pour n = 64 estimé à : 1h 35m 48s
T. exécution (s)
Nb. threads
(4;1473,5)
(8;1135,9)
(12;1015,5)
(16;1339,2)
(24;1669,5)
(28;1815,4)
(32;2074,1)


Matrice TF17 : T. exécution séquentielle pour n = 64 estimé à : 1h 35m 48s
Accélération
Nb. threads
(4;3,9)
(8;5,06)
(12;5,66)
(16;4,29)
(24;3,44)
(28;3,17)
(32;2,77)


Matrice IG5-18 : T. exécution séquentielle pour n = 64 estimé à : 1h 48m 52s
T. exécution
Nb. threads
(4;1695,6)
(8;1315,7)
(12;1216,4)
(16;1593)
(24;1997,5)
(28;2275)
(32;2542,6)


Matrice IG5-18 : T. exécution séquentielle pour n = 64 estimé à : 1h 48m 52s
Accélération
Nb. threads
(4;3,85)
(8;4,96)
(12;5,37)
(16;4,1)
(24;3,27)
(28;2,87)
(32;2,57)


Matrice TF18 : T. exécution séquentielle estimé à : 9h 45m 39s
T. exécution (s)
Nb. threads
(4;9340,6)
(8;7955,3)
(12;7100,9)
(16;8743,6)
(24;10822,9)
(28;11419,6)
(32;12365,9


Matrice TF18 : T. exécution séquentielle estimé à : 9h 45m 39s
Accélération
Nb. threads
(4;3,76)
(8;4,42)
(12;4,95)
(16;4,02)
(24;3,25)
(28;3,08)
(32;2,84)

Interprétation des résultats:

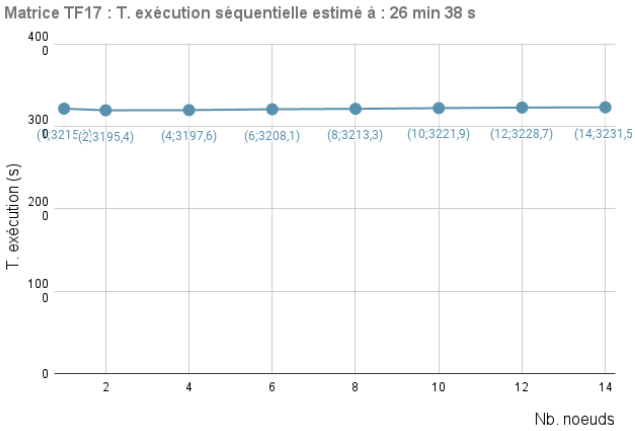
- On retrouve pour la version OMP du programme, des accélérations sublinéaires.
- On remarque également que l'on a des accélérations maximales ne dépassant pas 6 mais qui suffisent à être plus importantes que celles de la version MPI pour certaines matrices telle que TF17.
- Pour toutes les matrices testées, on observe une même limite de threads à partir de laquelle les performances se dégradent et c'est lorsque le nombre de threads est fixé à 12. On peut donc supposer qu'au delà de cette valeur, créer de nouveaux threads rajoute un sur-coût non négligeable.

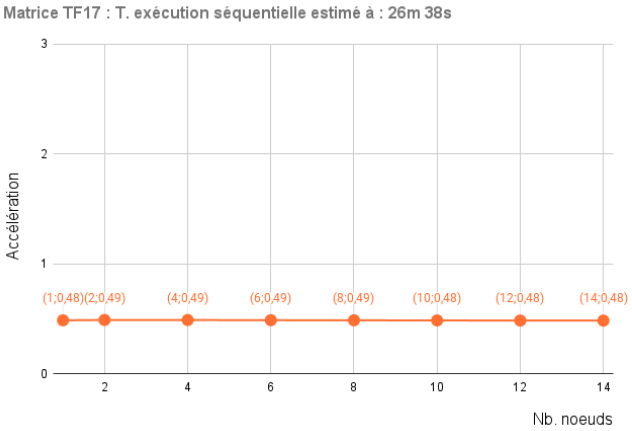
5 Parallélisation avec MPI + OpenMP

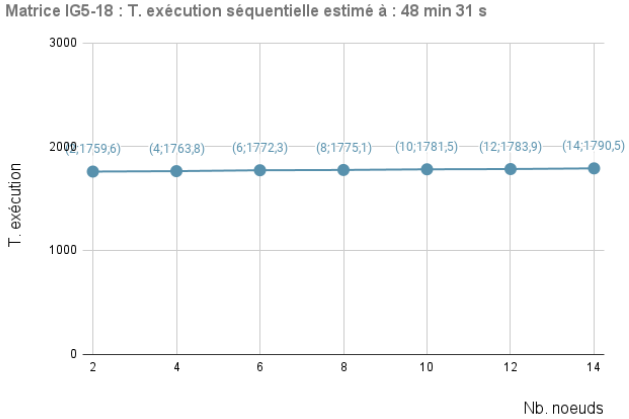
Cette parallélisation n'est autre que la combinaison des deux versions du programme: la version MPI et la version OMP.

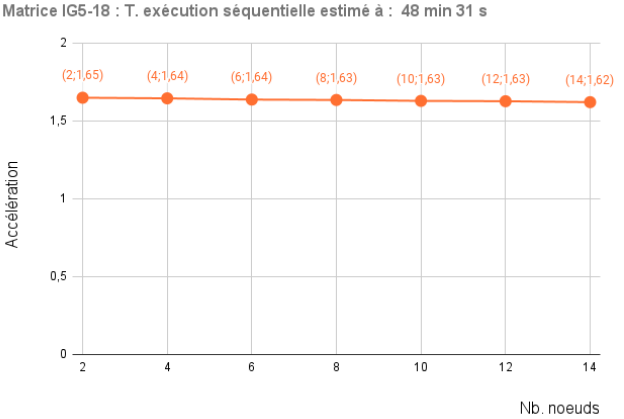
Le challenge dans cette partie était de fixer les bons paramètres pour une meilleure parallélisation. Après plusieurs tests et en fonction des résultats obtenus dans les deux premières parties, nous avons fixé le nombre de threads à 12, la valeur du paramètre n à 16 (car même si les performances auraient été meilleures avec un n plus grand, les temps d'exécutions, eux, auraient été conséquents pour nos tests) puis nous faisons varier le nombre de noeuds en fonction du temps d'exécution (Resp. l'accélération).

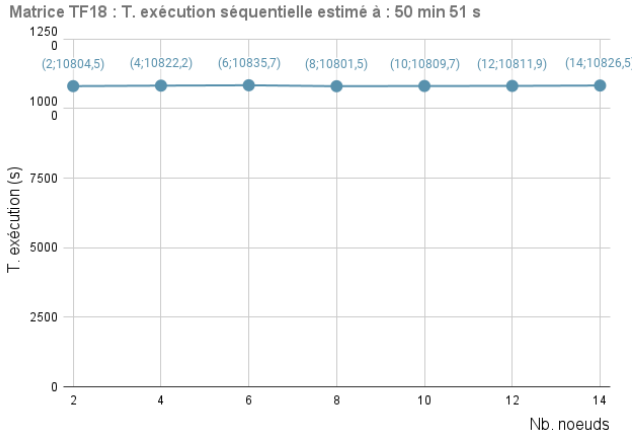
Les tests sont une fois de plus effectués sur 16 noeuds du cluster «gros»:

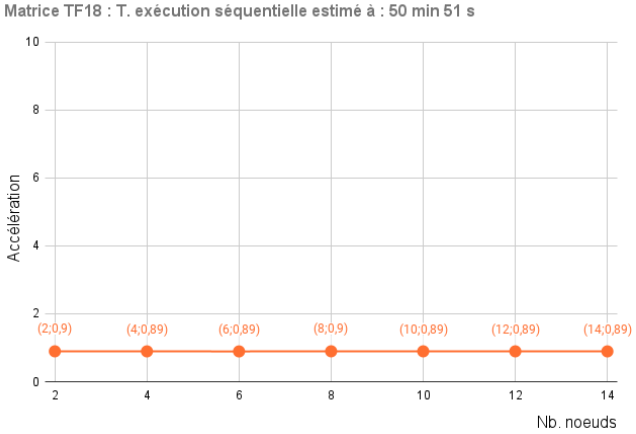

Matrice TF17 : T. exécution séquentielle estimé à : 26 min 38 s
T. exécution (s)
Nb. noeuds
(2;3195,4)
(4;3197,6)
(6;3208,1)
(8;3213,3)
(10;3221,9)
(12;3228,7)


Matrice TF17 : T. exécution séquentielle estimé à : 26m 38s
Accélération
Nb. noeuds
(1;0,48)(2;0,49)
(4;0,49)
(6;0,49)
(8;0,49)
(10;0,48)
(12;0,48)
(14;0,48)


Matrice IG5-18 : T. exécution séquentielle estimé à : 48 min 31 s
T. exécution
Nb. noeuds
(4;1763,8)
(6;1772,3)
(8;1775,1)
(10;1781,5)
(12;1783,9)
(14;1790,5)


Matrice IG5-18 : T. exécution séquentielle estimé à : 48 min 31 s
Accélération
Nb. noeuds
(2;1,65)
(4;1,64)
(6;1,64)
(8;1,63)
(10;1,63)
(12;1,63)
(14;1,62)

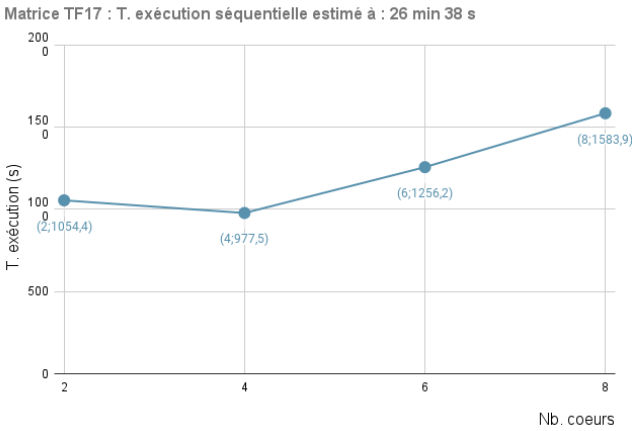

Matrice TF18 : T. exécution séquentielle estimé à : 50 min 51 s
T. exécution (s)
Nb. noeuds
(2;10804,5)
(4;10822,2)
(6;10835,7)
(8;10801,5)
(10;10809,7)
(12;10811,9)

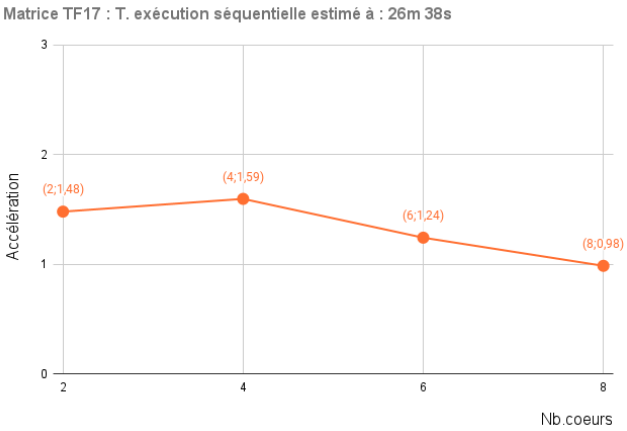

Matrice TF18 : T. exécution séquentielle estimé à : 50 min 51 s
Accélération
Nb. noeuds
(2;0,9)
(4;0,89)
(6;0,89)
(8;0,9)
(10;0,89)
(12;0,89)
(14;0,89)

Interprétation des résultats:

- On constate que les résultats ne sont pas bons sur les matrices étudiées. On retrouve une accélération en dessous de 1 et des temps d'exécution deux fois moins bons que ceux de la version itérative pour les matrices TF18 et TF17 et une amélioration constante des performances pour la matrice IG5-18.
- On conclut qu'il n'y pas grande amélioration des temps d'exécution en rajoutant de nouveaux noeuds et que la combinaison des deux versions nécessite plus d'optimisation que de combiner les bouts de code des deux programmes. Une autre théorie serait d'avoir pris trop de threads ce qui rendrait les temps de communication trop prenants.

Nous avions également testé sur la matrice TF17, le cas où l'on réduisait le nombre de noeuds à 1 (toujours sur le cluster «Gros») et que l'on faisait plutôt varier le nombre de coeurs :





On voit bien qu'au bout de 4 coeurs, les performances commencent peu à peu à se dégrader.

6 Conclusion

Ce projet fut très constructif car il nous a permis de traiter un cas réel de problèmes en HPC et nous à appris à toujours questionner les performances de nos algorithmes et à faire face aux difficultés rencontrées quant à l'application des paradigmes de parallélisation vus en cours.

Comme perspectives d'amélioration, nous voulions implémenter une version du programme MPI avec un découpage 2D mais faute de temps nous n'avions pas pu finaliser cette version.