HPC : Résolution de systèmes linéaires creux par la méthode du gradient conjugué

Tingting LI / Sonia Moussaoui

3 juin 2020

1 Parallélisation avec OpenMP

1.1 Adaptation du code pour la parallélisation OpenMP

- Dans les fonctions *extract_diagonal* et *cg_solve*, la parallélisation est effectuée sur la boucle for la plus externe avec la directive for d'OpenMP.
- Dans la fonction *dot*, la parallélisation est effectuée sur la boucle for la plus externe avec la directive for d'OpenMP. On associe à cette directive la clause de réduction avec l'opérateur *sum*.
- Dans le main, on parallélise les boucles for pour préparez la taille à droite et pour calculer $y = Ax - b$.

1.2 Performances

On effectue plusieurs test sur différents types de matrices.De plus, on a utilisé les machines de la salle 14-305 de la PPTI en passant par la passerelle.

- On commence par regarder pour des matrices de petite taille et dense.
  - La matrice *tmt_sym* :

Nombres de thread	séquentiel	2	4	6	8
temps d'exécution	63.1	40.4	36.5	39.4	41.0

    On a l'efficacité la plus grande avec $P = 2$: $E = 0.78$.

  - La matrice $cfd2$:

Nombres de thread	séquentiel	2	4	6	8
temps d'exécution	35.5	20.9	18.3	17.7	18.0

    On a l'efficacité la plus grande avec $P = 2$: $E = 0.85$.

  La répartition des données dans chaque instance est statique. Ainsi lorsque le travail est divisé pour chaque thread, on aura la même quantité de données à traiter. Si la répartition des itérations est aléatoire , des threads peuvent alors se retrouver à ne rien faire tandis que d'autres travaillent.
  C'est pourquoi nous avons pas choisi une répartition statique. On remarque que le temps d'exécution diminue dans la version openMp et que le multithreading n'est pas toujours le plus efficace car des fois la matrice est trop petit et y a beaucoup de threads ou bien trop de threads pour un type de machine.

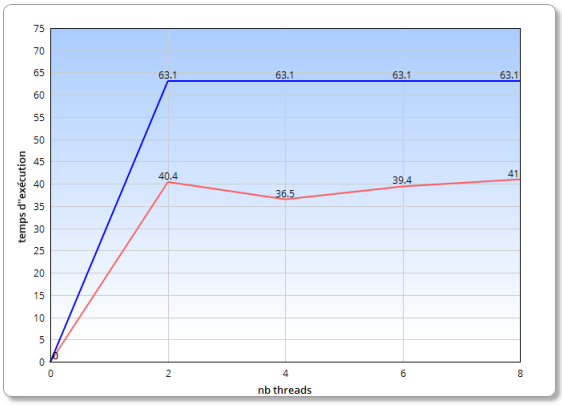

Figure 1: Temps d'exécution de *tmt_sym* en comparaison avec le séquentielle (bleu)

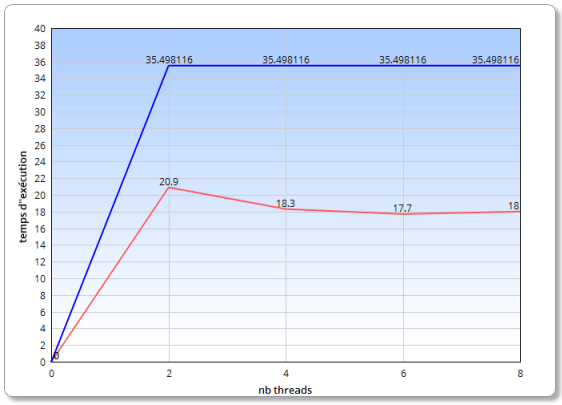

Figure 2: Temps d'exécution de $cfd2$ en comparaison avec le séquentielle (bleu)

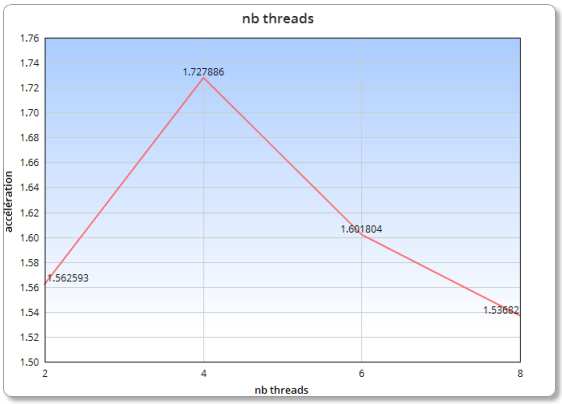

Figure 3: Accélération de *tmt_sym*

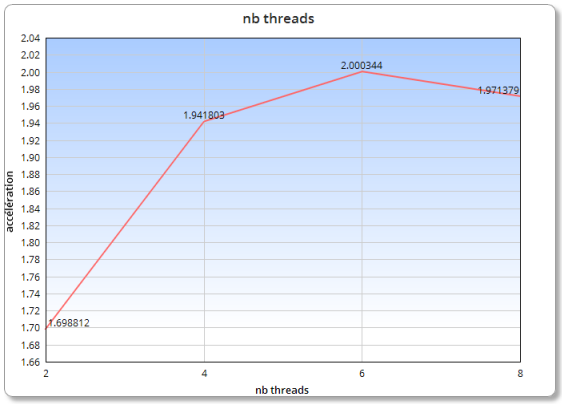

Figure 4: Accélération de $cfd2$

- Puis, on va regarder pour des matrices grandes et creuse.
  - Pour la matrice *G3_circuit*

Nombres de thread	séquentiel	2	4	6	8
temps d'exécution	89.2	67.9	66.8	69.4	71.8

On a l'efficacité la plus grande avec $P = 2$: $E = 0.66$.

  - Pour la matrice *thermal*2

Nombre de thread	séquentiel	2	4	6	8
temps d'exécution	153.8	99.1	84.2	88.5	91.0

On a l'efficacité la plus grande avec $P = 2$: $E = 0.78$.

On remarque que le temps d'exécution est plus petit on version OpenMP qu'en séquentielle et l'accélération augmente et atteint un maximum puis diminue. On peut donc penser que les quantités de calcul attribuer à chaque processus est trop petite. Donc la granularité de tâche de travail est trop petite par rapport au échange de données, ce qui augmente le temps d'exécution. On le voit surtout pour les matrices de grandes traille et creuse.

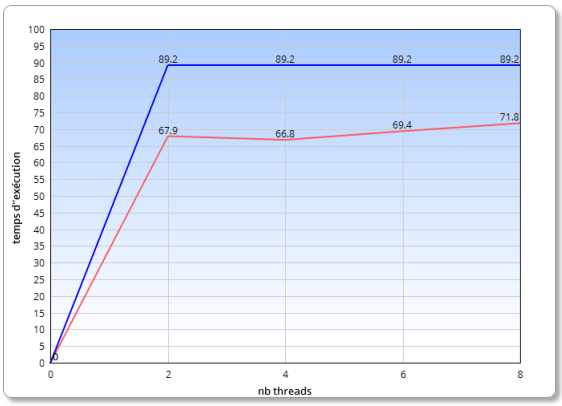

Figure 5: Temps d'exécution de *G3_circuit* en comparaison avec le séquentielle (bleu)

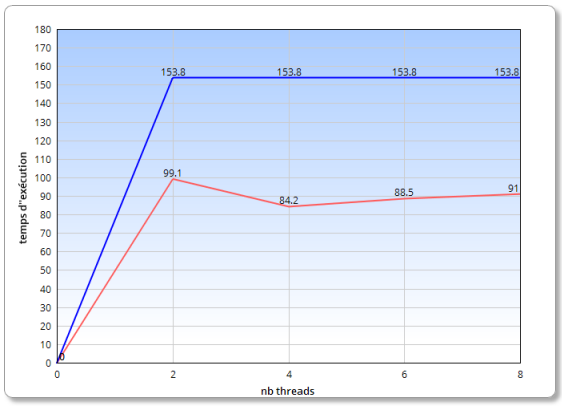

Figure 6: Temps d'exécution de *thermal2* en comparaison avec le séquentielle (bleu)

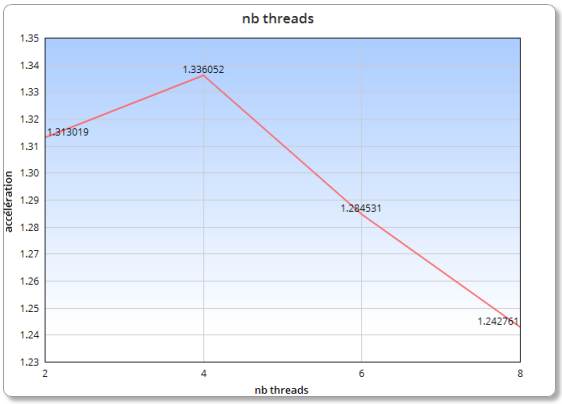

Figure 7: Accélération de *G3_circuit*

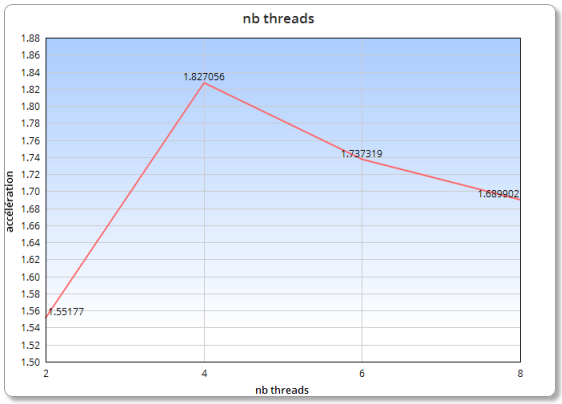

Figure 8: Accélération de *thermal2*

2 Parallélisation avec MPI

2.1 Adaptation du code pour la parallélisation MPI

- Dans *cg_solve*, on passe en plus en arguments le rang du processus qui appelle la fonction et le nombre de processus qu'on utilise en total. Il fait le calcul sur *new_size* éléments et puis en utilisant *Allreduce*, on somme tout les valeurs locales et *Allgather* pour rassembler tout les données et les communiquer à tout les processus.
- Dans *sp_gemv*, *dot* et *norm*, on les adapte à l'exécution sur plusieurs processus.
- Dans le main, seul le processus 0 lit la matrice en entière en utilisant la fonction $load_mm$ puis on utilise *Bcast* pour diffuser la matrice à tous les autres processus.

Remarque : On a aussi modifier le code de la fonction $do_computation()$ dans *runner.py* pour ignorer les lignes de sortie qui ne sont pas des flottantes. En effet, la sortie nous donne aussi les messages de connexion aux machines de la PPTI, donc on les ignore pour ne pas poser de problème lors de la vérification du résultat.

2.2 Performances

On effectue plusieurs test sur différents types de matrices.De plus, on a utilisé les machines de la salle 14-305 de la PPTI en passant par la passerelle.

- On commence par regarder pour des matrices de petite taille et dense.
  - La matrice $cfd2$:

Nombres de machine	séquentiel	2	4	6	8
temps d'exécution	35.5	23.4	22.2	77.6	116.4

On a l'efficacité la plus grande avec $P = 2$: $E = 0.75$.

  - La matrice tmt_sym :

Nombres de machine	séquentiel	2	4	6	8
temps d'exécution	63.1	47.2	53.4	390.5	372.1

On a l'efficacité la plus grande avec $P = 2$: $E = 0.65$.

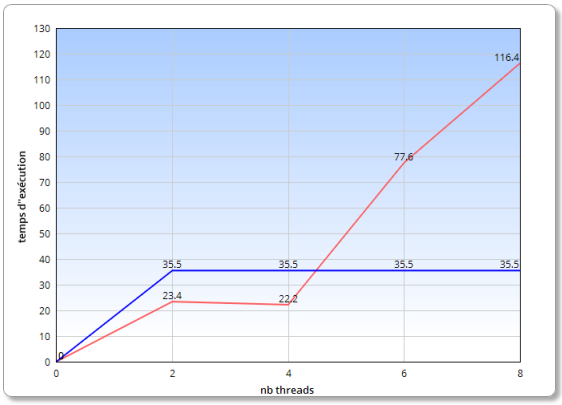

Figure 9: Temps d'exécution de $G3\_circuit$ en comparaison avec le séquentielle (bleu)

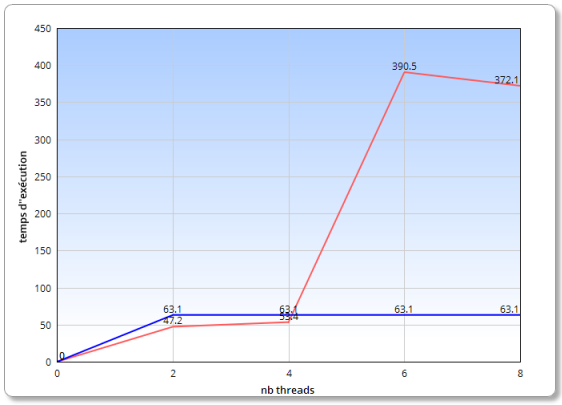

Figure 10: Temps d'exécution de $thermal2$ en comparaison avec le séquentielle (bleu)

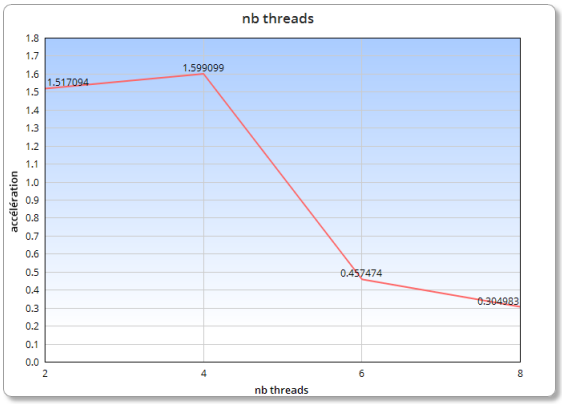

Figure 11: Accélération de tmt_sym

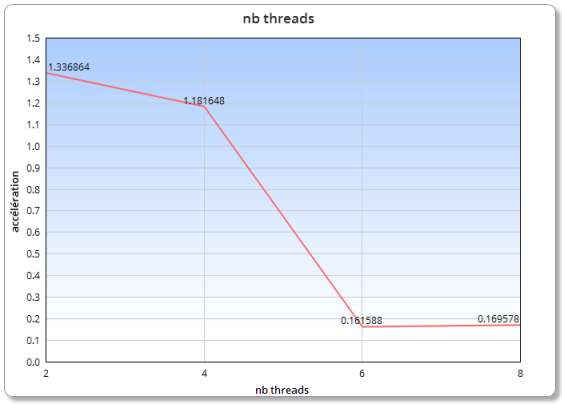

Figure 12: Accélération de $cfd2$

- Puis, on va regarder pour des matrices grandes et creuse.
  - La matrice $G3_circuit$:

Nombres de machine	séquentiel	2	3	3	5
temps d'exécution	89.2	84.2	87.1	91.9	460.5

On a l'efficacité la plus grande avec $P = 2$: $E = 0.55$.

  - La matrice $thermal2$:

Nombres de machine	séquentiel	2	3	4	5
temps d'exécution	153.8	111.5	109.5	111.5	505.6

On a l'efficacité la plus grande avec $P = 2$: $E = 0.69$.

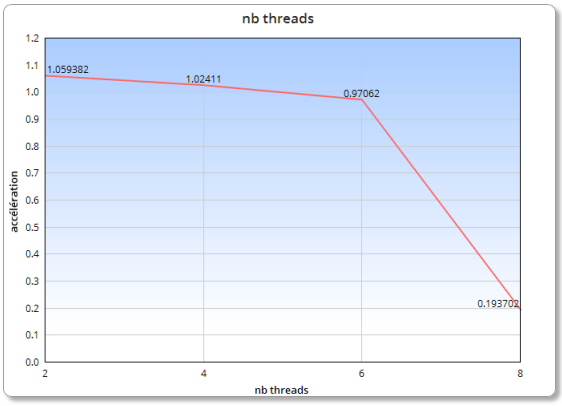

Figure 13: Accélération de $G3_circuit$

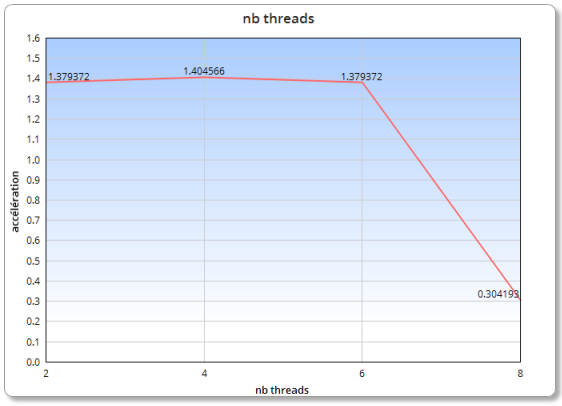

Figure 14: Accélération de $thermal2$

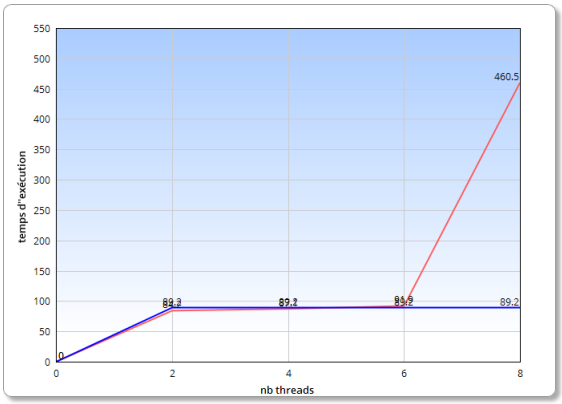

Figure 15: Temps d'exécution de $G3\_circuit$ en comparaison avec le séquentielle (bleu)

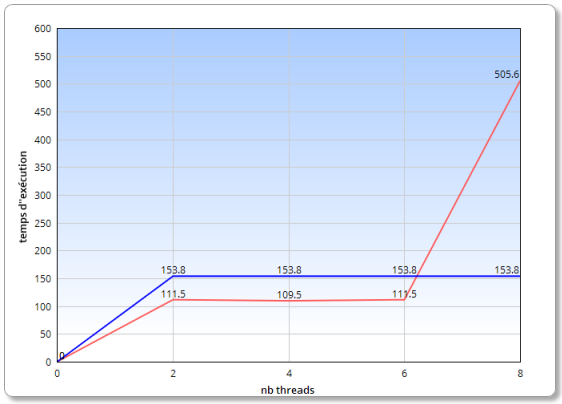

Figure 16: Temps d'exécution de $thermal2$ en comparaison avec le séquentielle (bleu)

De même que pour la parallélisation en OpenMP, on remarque l'accélération augmente et atteint un maximum puis diminue. On peut donc penser que la granularité de tâche de travail est petite par rapport au échange de données, ce qui augmente le temps d'exécution. On le voit surtout pour les matrices de grandes traille et creuse.

3 Parallélisation avec MPI+OpenMP

3.1 Adaptation du code pour la parallélisation MPI+OpenMP

On fusionne les modifications faites dans les deux parties précédentes.

3.2 Performances

On va regarder pour la matrice $cfd2$:

threads \ machines	2	3
2	21.0	21.6
4	21.6	806.3

On remarque que pour la matrice $cfd2$, le temps d'exécution n'est pas meilleur qu'avec la parallélisation MPI seule ou OpenMp seule. De plus quand on augmente le nombre de machine, l'exécution devient très lente.)