Rapport : Projet MPI+X

Mehdi Hala

1 Introduction :

Ce projet a pour but de d'implémenter différentes versions parallèles du code stencil pour observer les changement de performances entre celles-ci.

2 Version de base :

Le code proposé est stencil en 2 dimensions. À chaque itération, une cellule est mise à jour à partir de ces 4 voisins (haut, bas, droite et gauche). L'algorithme se termine s'il y a convergence (ie plus de mise à jour de cellules possible).

La version de base n'a pas de bonnes performances mais nous pouvons quand même gagner quelques centaines de MFlop/s.

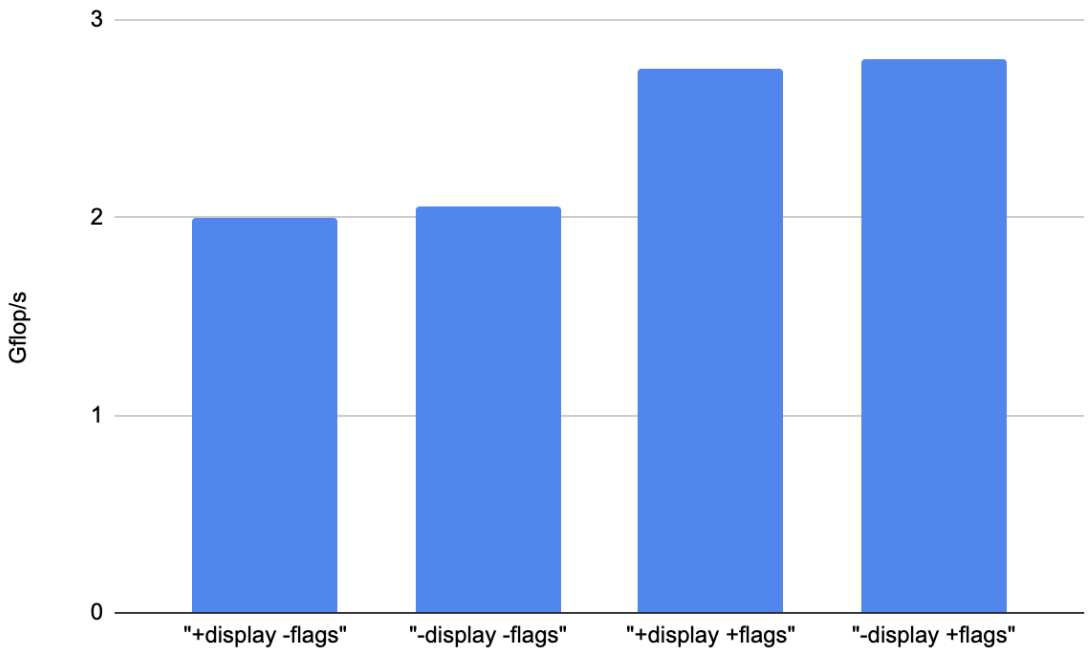

Figure 1 – Désactivation de l'affichage et ajout de flags

La figure ci-dessous représente la courbe de performance de la version séquentielle de stencil.

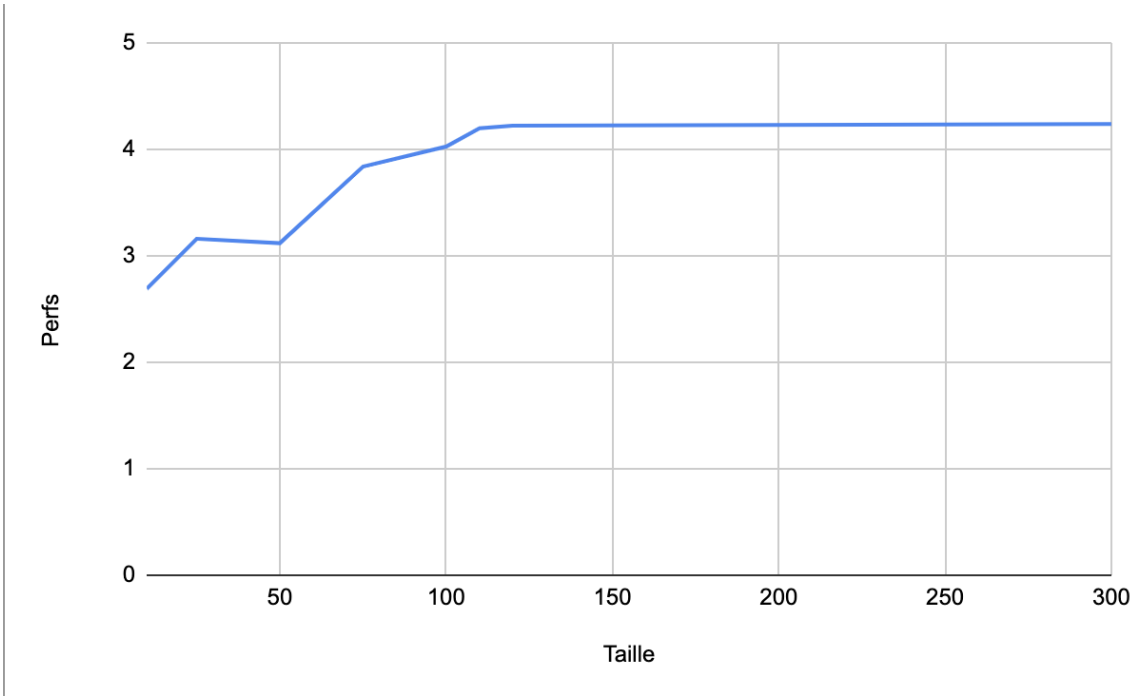

FIGURE 2 – Performances de la version de base en fonction de STENCIL_SIZE_X=STENCIL_SIZE_Y

On remarque que les performances stagnent aux alentours d'une taille de 120, celà est du au fait que Plafrim a un cache de 256Ko. Or notre programme a besoin d'allour deux tableaux de 120*120 doubles ce qui donne un total de :

$$2 * 120 * 120 * 8 = 230Ko \tag{1}$$

3 Version OpenMP :

Pour paralleliser ce code, il faut parralleliser la fonction de mise à jour et la fonction de test de convergence.

3.1 Fonction de mise à jour :

Pour paralleliser cette fonction, nous utilisons la clause suivante :

```
#pragma omp parallel for collapse(2)  shared(values)\
firstprivate(prev_buffer, next_buffer)
```

La variable *values*, étant partagée entre tous les threads, a été mise en **shared**. Les deux variables *prevbuffer* et *next_ buffer*, étant des constantes, ont été mises en **firstprivate**.

3.2 Test de convergence :

Avant de paralleliser cette fonction, il faudra enlever le *break* fait par le return à l'intérieur de la boucle. Cela coûtera des tours de boucles de plus mais il est nécessaire pour avoir du parallélisme. Ainsi, nous utilisons la clause suivante :

```
#pragma omp parallel for collapse(2) private(x, y)\
firstprivate(prev_buffer, values) reduction(&& : converged)
```

avec *converged* le booléen crée qui se met à jour à l'intérieur de la boucle.

3.3 Performances :

Nous avons testé les performances de cette version en variant deux paramètres, à savoir le nombre de threads et aussi la taille de X et Y. Nous remarquons qu'on utilisons moins de threads pour les matrices de plus petite taille, on a de meilleurs performances. Cela est du au fait qu'avec plus de thread, la charge de travail par thread n'est même pas suffisante pour compenser la création et destruction de ceux-ci. Mais pour une taille plus grande (exemple taille = 1000), plus de thread donne de meilleurs performances.
On remarque aussi que le seuil de stagnation des valeurs a été décalée à cause du fait que chaque thread travaille sur une partie des données stockées dans le cache.

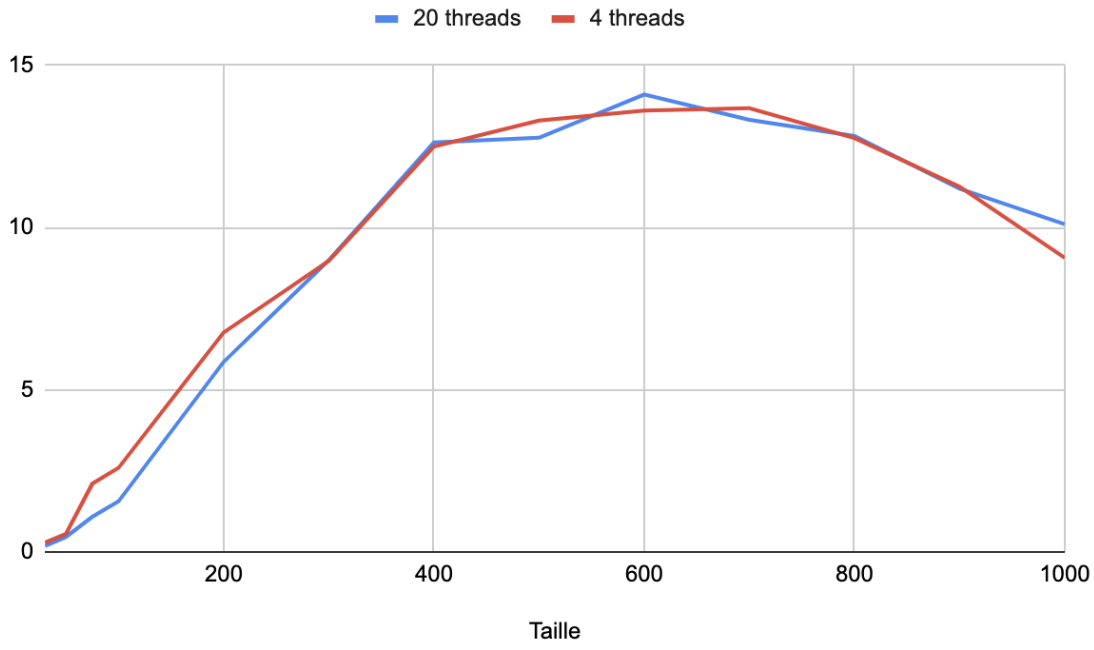

FIGURE 3 – Performances de la version de omp en fonction de STENCIL_SIZE_X=STENCIL_SIZE_Y

4 Version MPI :

4.1 Choix des datatypes pour les halos :

Les matrices ayant un ordre column major, Nous avions donc choisi de mettre les colonnes dans des **MPI_type_vector** et les lignes dans des **MPI_type_contiguous**.

4.2 Communications entre processeurs :

Nous sommes dans ce tp dans une topologie cartésienne, nous utilisons donc des **MPI_Cart_shift** pour retrouver le rang des 4 voisins de chaque processus (**MPI_PROC_NULL** dans le cas

où on se trouve sur les bords). Cela permettra de faire les envoie et réception de données.

4.3 Mise à jour et convergence :

La mise à jour est la même que celle des versions précédentes. Tandis qu'on utilise un **MPI_Allreduce** pour tester la convergence (Pas de grandes différences entre **MPI_Allreduce** et **MPI_Reduce** sauf que le premier fait aussi un broadcast.).

4.4 Performances :

On obtient de très bonnes performances et de mieux en mieux si nous augmentons le nombre de processus par noeud.

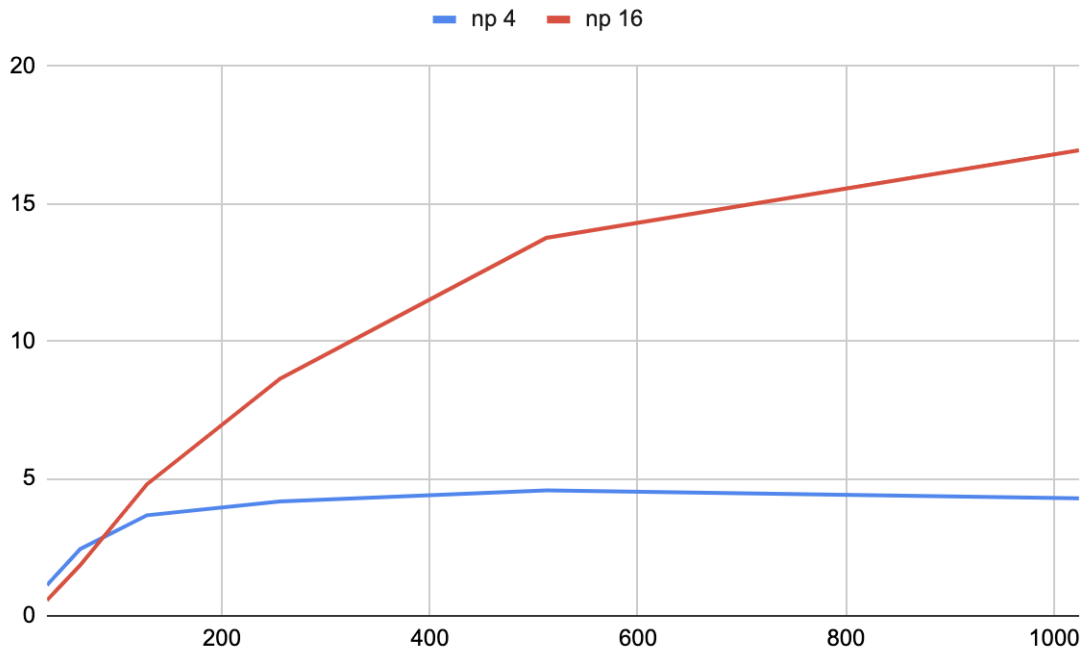

FIGURE 4 – Performances de la version de omp en fonction pour les tailles 64, 128, 256, 512 et 1024

On remarque aussi que tout comme la version OMP, la charge par processus doit être assez raisonnable sinon il seront allouer sans raison et ce qui causera de mauvaises performances comme dans le cas des tailles inférieur à 128 en faisant un -np 16, on obtient de performances moins bonnes qu'on faisant un -np 4.

5 Version MPI+X :

Cette version reprend le code de la version précédente en ajoutant une clause OpenMP à la phase de mise à jour.

```
#pragma omp parallel for private(x, y) collapse(2) reduction(&&: local_converged)
```

Mais cette version ne donne pas les performances attendues. Les performances sont moins bonnes que ceux de la version MPI pure et aussi de la version OpenMP.

6 Conclusion :

Ce projet nous a permit de mettre en place plusieurs versions parallèles du même code, de pouvoir comparer leurs performances et aussi de s'habituer à observer les problèmes relatifs à la localité. Ainsi, qu'il nous a permis de faire un premier pas vers la programmation hybride.