Projet de Stencil OpenMP/MPI/MPI+X

Vincent Bridonneau

January 2020

1 Introduction

L'objectif de ce projet est d'implémenter différentes versions d'une application de stencil pour mesurer les gains de performances. Ces versions sont des variantes parallélisées d'une application de base déjà fournie. La première utilise des directives *OpenMP*, la seconde la bibliothèque *MPI* et la troisième est une hybride des deux versions précédentes.

Note: Par une mauvaise lecture du code de base, nous n'avons pas vu que le halo ne concerne que les voisins droite, gauche, haut et bas et non les diagonales. Dans notre code la gestion de ces cas apparaît mais n'impacte pas le reste de l'application. Il s'agit simplement d'une transmission inutile de données. Dans le rapport qui suit, nous n'expliquons que les solutions misent en place pour les échanges en lignes et colonnes.

De plus, les valeurs des performances en ordonnée dans les figures ne prennent pas en compte le nombre maximum d'itération. Il faut donc multiplier le résultat par 10000 pour avoir le bon résultat.

2 Version de base

L'application de base est une application de stencil en 2 dimensions. À chaque itération, une cellule à l'instant t se met à jour à partir de ses voisins directs à l'étape $t-1$, comme illustré figure 3. À cela s'ajoute un test de convergence. Celui-ci est fait après avoir mis à jour la grille. On constate alors, ayant un deuxième parcourt des données, que l'on va devoir relire la grille une deuxième fois après la mise-à-jour dans le pire des cas. On pourrait alors perdre en efficacité car si la grille est grande, il faudra sans doute recharger les données dans le cache pour le test. Cela peut être optimisé en faisant le test de convergence en même temps que la mise-à-jour.

2.1 Étude des performances

Pour commencer nous avons tracé la courbe des performances en temps pour l'application de base (figure 1). On observe une saturation pour une taille de

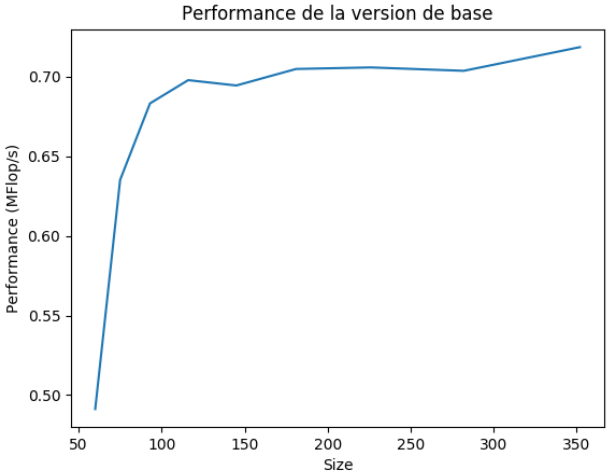


Figure 1: En abscisse le nombre d'éléments en x et et y (`STENCIL_SIZE_X`=`STENCIL_SIZE_Y`=`size`). En ordonnée le nombre d'opérations floattantes par sencodes en MFlop/s.

130. Or pour une telle taille, on a en mémoire 2 tableaux dans `values` (`prev` et `next`) chacun contenant 8 (taille des `doubles`) fois 130^2 octets, soit un total d'environ 270k octets. Or sur plafrim, les machines à disposition ont un cache L2 de 256k octets. On en déduit donc que la saturation est due au fait que les données ne tiennent pas dans le cache L2. Le nombre d'opération est calculé comme suit: *nombre d'itérations maximal* $*$ *nombre d'éléments du tableau à modifier* (`STENCIL_SIZE_X`-2)$*$(`STENCIL_SIZE_Y`-2) $*11$, où 11 correspond au nombre d'opération arithmétiques pour mettre un élément à jour. (Sur les graphiques, il manque un facteur 10000 correspondant au nombre d'itérations).

Observons à présent ce qu'il se passe quand on fait tout en une étape (2). Dans ce cas, les performances sont sensiblement plus faible. Ceci est due au fait qu'à chaque mise un jour on compare directement l'erreur par rapport à l'ancien terme, ce qui rajoute un coup supplémentaire (valeur absolue, ...), sans augmenter le nombre d'opération arithmétique (Flop), alors que précédemment, on sortait de la boucle de test de convergence dès que la différence était trop grande.

3 Version OpenMP

Nous étudions à présent la version de base parallélisée avec OpenMP.

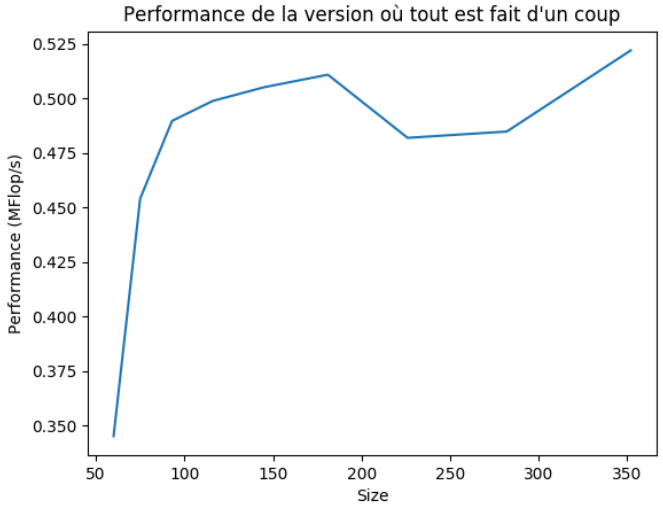


Figure 2: Même chose que pour la version de base.

3.1 Parallélisation de la mise à jour

Pour commencer, nous avons parallélisé les deux boucles for avec une directive `omp`. Nous commençons par fusionner les deux boucles. Nous n'oublions pas de déclarer la portée des variables. Le tableau `values` est partagé (`shared`) car tout le monde le modifie et a le même tableau. Les indices du tableau `prev` et `next` sont constants et les mêmes d'un thread à l'autre. Pour éviter le placement d'un mutex nous les déclarons `firstprivate`. Les indices des boucles étant propre à chaque thread, ils sont déclarés privés:

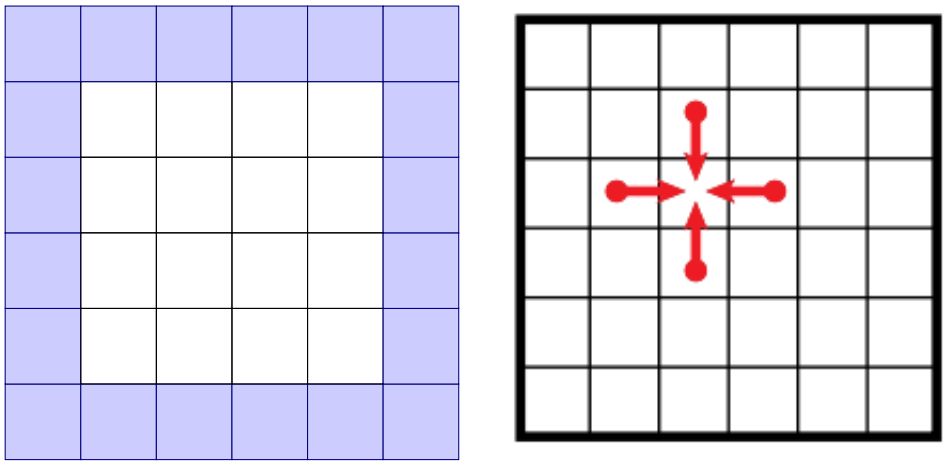
```
#pragma omp parallel for collapse(2) shared(values)\
firstprivate(prev_buffer, next_buffer)
```

3.2 Parallélisation du test de convergence

Pour paralléliser le test de convergence, nous devons déjà modifier le corps de la boucle la plus enfouie pour ne pas retourner de la fonction. Nous déclarons une variable contenant le résultat de la convergence et faisons une réduction avec un "et" logique :

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

Nous notons cependant qu'avec cette modification, nous ne pouvons pas retourner de la fonction dès que la différence dépasse le critère d'arrêt. Nous devons parcourir tout le tableau. Cela peut être inefficace surtout dans les



a **b**

Figure 3: Problème de stencil étudié. À gauche (**a**) le problème vu comme une grille 2D. La mise à jour ne concerne pas les cases sur les bords en bleu. Toutes les cellules modifiées ont besoin de leur voisin direct pour évoluer (**b**).

débuts où l'on ne parcourt que très peu d'éléments (ils changent de valeurs très souvent).

3.3 Performances

Comme pour les versions précédentes, nous avons tracé les courbes de performances. Ici, nous avons alloué une machine en exclusif avec *salloc* et fait varier le nombre de thread *OpenMP*. On obtient les courbes suivantes (4): On observe que pour une taille très faible (50), plus le nombre de thread est grand, moins les performances sont bonnes. Ceci s'explique par le fait que la quantité de travail n'est pas suffisamment grande pour pallier le temps de création / destruction des threads. En revanche la limite en performance augmente avec le nombre de thread. Cela s'explique par le fait que chaque thread travail sur une partie des données qui tient dans le cache L2 du processeur sur lequel il tourne, car plus petite que la tuile complète.

4 Version MPI Pure

Voyons à présent l'implémentation de cette application en MPI pure.

4.1 Le problème

Pour des raisons de simplicité, nous avons décidé que les processus auraient tous une tuile de dimension `STENCIL_SIZE_X` $\times$ `STENCIL_SIZE_Y`, ce qui permet de

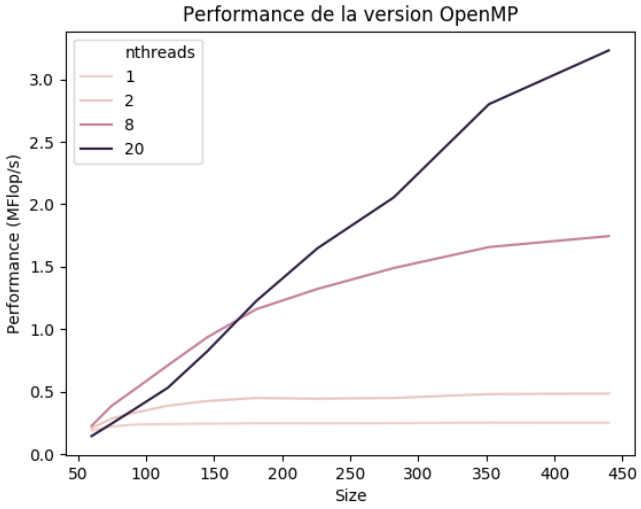


Figure 4: Test de performance pour différents nombre de thread: $1, 2, 8$ et 20.

ne pas toucher au nombre de données à traiter par processus. Pour faciliter les communications entre les processus nous avons décider d'utiliser une topologie MPI cartésienne de dimension 2. Le nombre de processus par ligne/colonne est décidé de la façon suivante : le plus grand diviseur en dessous de la racine carré du nombre de processus (notons le d) est le nombre de processus dans chaque direction de la dimension x et $\frac{n}{d}$ celui dans la dimension y, où n est le nombre de processus. Ainsi pour 12 processus, on aura 3 processus par ligne (x) et 4 par colonne (y).

4.2 Distribution des données

Nous l'avons vu, chaque processus à une tuile `STENCIL_SIZE_X` $\times$ `STENCIL_SIZE_Y`. L'idée est que seule les éléments d'indices $(x, y) \in [1;$ `STENCIL_SIZE_X`$-2] \times$ [1;`STENCIL_SIZE_Y`$-2]$ seront modifiés au court du temps. Le bord des tuiles correspond au condition de bord pour $x = 0, x = d - 1, y = 0$ et $y = \frac{n}{d} - 1$ et dans les autres cas il correspond aux valeurs des voisins directs. Par exemple, la colonne la plus à droite d'un processus correspond à la deuxième colonne de gauche de son voisin de droite. En voici une illustation figure 5:

Ceci permet de ne pas ajouter de complexité au code : pas besoin de traiter à part la mise à jour des cellules, elles ont toutes le même traitement.

4.3 Définitions des types

Nous avons défini des types pour communiquer des lignes et des colonnes entre les différents processus. Le stockage des données étant en colonne ma-

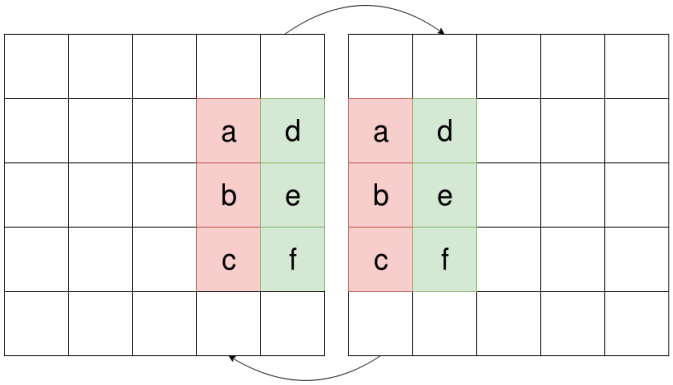


Figure 5: Deux processus voisins. La colonne droite du processus de gauche contient à chaque étape la deuxième colonne en partant de la gauche de son voisin de droite.

jeure, les éléments des colonnes sont contiguës alors qu'il y a un décalage de `STENCIL_SIZE_Y` entre deux éléments d'une même ligne. L'idée est de créer deux types, Un `MPI_Type_contiguous` pour les colonnes et un `MPI_Type_vector` pour les lignes. Cependant, lors des tests, des problèmes répétés en utilisant le type ligne ont fait que nous avons traité les échanges de lignes séparément. Pour gérer ce cas, nous avons d'abord stocker les données dans un buffer temporaire et avons envoyer `STENCIL_SIZE_X`-2 `MPI_DOUBLE` avant de les replacer dans la grille (mécanisme similaire à *Pack/Unpack*).

4.4 Communication entre les processus

Pour commencer, étudions comment déterminer le voisin d'un processus. Étant sur une topologie cartésienne, nous avons utilisé la routine `MPI_Cart_shift` pour connaître les voisins directs. Cette routine permet de connaître les rangs des processus qui vont nous envoyer des données et ceux qui vont les recevoir lors d'un appel à `MPI_Sendrecv`. De plus, une valeur spéciale est renvoyée dans le cas des processus sur les bords : `MPI_PROC_NULL`. Cette valeur permet de ne pas traiter séparément les échanges quand le processus est sur un bord de la grille (Send et Recv retournent directement). Il faut cependant bien initialiser le buffer de réception pour rester très générique dans la gestion des échanges, sinon on est obligé de gérer les bords comme des cas particuliers. Dans notre cas, nous précisons directement l'adresse dans le buffer `values` qui va recevoir, comme ça la valeur du buffer est la même en retour de `Sendrecv` sur les bords, ce qui nous convient car les bords de la grille du stencil global sont invariants.

Les communications sont illustrées figure 6. Dans ce cas particulier, 0 reçoit de, et 1 envoie vers, `MPI_PROC_NULL`, c'est-à-dire rien ne se passe dans ces cas pour 0 et 1. Dans les autres cas i envoie vers $i+1$ et reçoit de $i-1$.

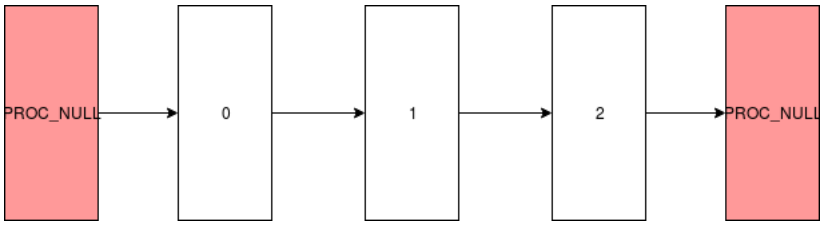


Figure 6: Exemple de communication des données de gauche à droite. `PROC_NULL` fait référence à `MPI_PROC_NULL`.

4.5 Mise-à-jour

L'étape de mise à jour ne change pas par rapport à l'algorithme de base, car les tuiles sont conçues de telle manière que toutes les tuiles sont mise-à-jour de la même manière.

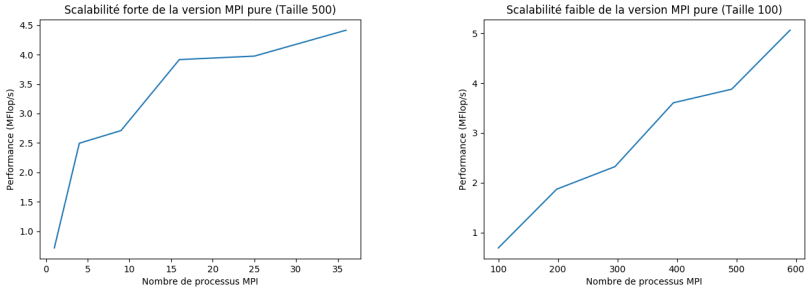
4.6 Performances

Dans ce cas ci, nous avons réalisé des tests de scalabilité faible et forte pour observer ce qu'il se passe pour différent nombre de thread.

Pour les tests de scalabilité forte, la taille du problème ne doit pas varier. Nous avons choisi, pour des raisons de simplicité, de considérer des grilles de processus carrés. Notons N la taille globale désirée et n l'argument des matrices `values` (i.e. (STENCIL_SIZE_X = STENCIL_SIZE_Y n)). Les premiers et derniers processus de chaque ligne ont $n-1$ éléments, car la dernière ou la permière ligne / colonne est une copie du voisin, alors que pour les autres, on a $n-2$ éléments, car la dernière et la permière ligne / colonne sont une copie des voisins. On a ainsi, $N = 2(n-1) + (p-1)(n-2) = 2 + p(n-2)$. On en déduit que pour avoir N fixé on doit choisir $n = \frac{N-2}{p} + 2$.

Les résultats des tests de scalabilité fortes et faible sont présentés figure 7 et 8.

Pour le test de scalabilité forte, on observe pour une taille de 500 que les performances ont une croissance plus faible une fois 16 processus dépassé. Pour expliquer cela, notons que pour exécuter les tests, nous avons choisi de mapper les processus par cœur et de les binder par cœur également. Or sur plafrim, il y a 20 cœurs par machine. Ainsi pour 25 et 36, respectivement 5 et 16 processus sont sur un autre noeud. Les communications sont donc ralenties car l'information doit passer d'une machine à l'autre. Concernant les tests de scalabilté faible, on observe aucune saturation pour $n = 100$ et une forte saturation pour $n = 300$. Pour $n = 100$, cela est due au fait que la grille tienne entièrement dans la cache L2 (un total de 160k octets soit moins des 256k dans L2) alors que Pour $n = 300$, ce n'est pas du tout le cas.



a **b**

Figure 7: Test de scalabilité. À droite, un test de scalabilité faible **b** : en abscisse, la taille de la grille de stencil N. Pour obtenir le nombre de processus *MPI*, il suffit de diviser la valeur en abscisse par $n = 100$. À gauche **a**, un test de scalabilité forte ($N = 500$) : en abscisse, le nombre de processus *MPI*.

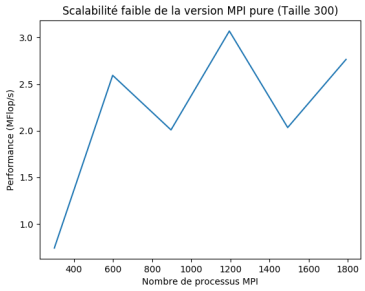


Figure 8: Test de scalabilité faible où des saturations apparaissent. En abscisse la taille de problème N. Pour obtenir le nombre de processus *MPI*, il faut diviser la valeur en abscisse par $n = 300$.

5 Version MPI + OpenMP

Dans cette version, nous utilisons notre code MPI pure et nous rajoutons des directives OpenMP. Nous avons testé les mêmes cas que dans la version OpenMP pour ce qui est des directives. Ici le plus important est de trouver le bindings des threads *OpenMP* et des processus *MPI* qui maximise les performances. Par manque de temps, nous n'avons pas pu chercher quel binding était le mieux. La principale piste envisagée était un mapping des processus *MPI* par noeuds et un bindings des threads *OpenMP* par coeur. Les threads *OpenMP* restant sur la même machine on aurait une localité mémoire intéressante (voire la version OpenMP). Les processus *MPI* aurait quant à eux permis d'échanger de façon optimisée les données entre les différents noeuds.

6 Conclusions

Ce projet aura permis de mettre en place différentes façon de paralléliser un code séquentiel. Une version *OpenMP* simple à implémenter dont les performance montent à 8MFlop/s (sans le facteur 10000 du nombre d'itération maximal) pour 20 threads comparé à une version MPI pure plus complexe (création de type, communication entre processus, ...), mais dont les performances montent à 4MFlops (toujours sans le facteur itération) pour 16 processus et plus.

Le projet aura également permis d'observer les problèmes de localité en mémoire et les coûts des communications entre les processus MPI qui impactent les performances de manière significative.