

# Travail de programmation parallèle : Dissipation de la chaleur d'un processeur

Tristan Chenaille, Étudiant-ingénieur en MAIN4, 2023

<b>1</b>	<b>Thématique</b>	<b>1</b>
<b>2</b>	<b>Analyse et correction du code fourni</b>	<b>2</b>
<b>3</b>	<b>Parallélisation 1D : découpe sur une dimension</b>	<b>4</b>
3.1	Idée générale	4
3.1.1	Principe et intérêt d'une découpe	4
3.1.2	Mode de découpe choisi	4
3.2	Performances du code de base et du code parallélisé sur 1 processus	5
3.3	Performances de la parallélisation	5
3.3.1	Premiers tests	5
3.3.2	Influence du nombre de couches autogérées	6
3.3.3	Influence du nombre de buffers à échanger	8
3.3.4	Asynchronisation des échanges de buffers	10
3.4	Tests en mode CHALLENGE	12
<b>4</b>	<b>Diagnostic prévisionnel des découpes en dimension &gt; 1</b>	<b>13</b>
4.1	Parallélisation 2D	13
4.2	Parallélisation 3D	13

## 1 Thématique

Dans ce travail, je m'intéresse à un programme informatique concernant le radiateur (= dissipateur de chaleur métallique) d'un CPU (= processeur) de modèle "AMD EPYC Rome". L'idée du programme est d'allumer un processeur au temps  $t = 0$  et de simuler la diffusion de la chaleur dans son radiateur jusqu'à ce qu'un état stationnaire soit atteint. Le code séquentiel (de l'annexe) effectue cette simulation de manière paramétrique avec une gestion de toutes les physiques entrant en jeu (flux de chaleur, densité du flux de chaleur, capacité thermique massique, transfert thermique, convection, rayonnement du corps noir).

Un tel programme a un grand intérêt. En effet, la gamme de processeurs "EPYC" de chez "AMD" (équivalent de la gamme "Xeon" de chez "Intel") est utilisée par nombre de datacenters/supercalculateurs. Ces derniers se doivent de tirer le meilleur parti de leurs ressources matérielles, et donc de mettre au point les meilleures solutions de refroidissement. Le programme permet justement de déterminer quel dissipateur serait le plus efficace selon ses dimensions et son métal de composition.

Cependant, une telle simulation demande une puissance de calcul considérable. L'objectif de mon travail est donc de paralléliser le programme, c'est à dire d'en répartir l'exécution sur un nombre  $n \geq 1$  de processus. Grâce à cela, les simulations pourront elles-mêmes être effectuées sur un supercalculateur en un temps record. J'ai choisi d'opter pour la technique *MPI* (= Message Passing Interface) afin que les calculs puissent être répartis simultanément sur les processeurs de plusieurs machines distinctes pour peu qu'elles soient connectées en réseau. Pour ce faire, j'ai utilisé la librairie *OpenMPI* en C, qui fait correspondre  $n$  à  $n_{cpu}$  (le nombre de CPU, i.e. de processeurs). J'ai réalisé l'implémentation en langage C, qui a l'avantage d'être non-interprété.

## 2 Analyse et correction du code fourni

Lors de ma première revue du code, j'ai rencontré une aporie. En effet, les premières définitions de constantes sont effectuées comme suit :

```
1 const double L = 0.15;      /* length (x) of the heatsink (m) */
2 const double l = 0.12;      /* height (y) of the heatsink (m) */
3 const double E = 0.008;     /* width (z) of the heatsink (m) */
```

Puis comme cela au début de la fonction main() :

```
1 int n = ceil(L / dl);
2 int m = ceil(E / dl);
3 int o = ceil(l / dl);
```

Ces déclarations font donc correspondre  $n$  (resp.  $m$ ,  $o$ ) aux pas sur les axes  $x$  (resp.  $z$ ,  $y$ ).

Puis, dans la boucle de simulation de la fonction main(), j'ai remarqué cela :

```
1 /* [...] xy planes processed for increasing values of z. */
2 for (int k = 0; k < o; k++) {    // z
3     int v = k * n * m;
4     do_xy_plane(T, R, v, n, m, o, k);
5 }
```

Le commentaire indique que tous les plans  $xy$  sont traités, et ce pour des valeurs de  $z$  croissantes. Cependant, la variable  $k$  augmente jusqu'à  $o$ , qui est le pas spatial pour l'axe  $y$  et non pour l'axe  $z$ .

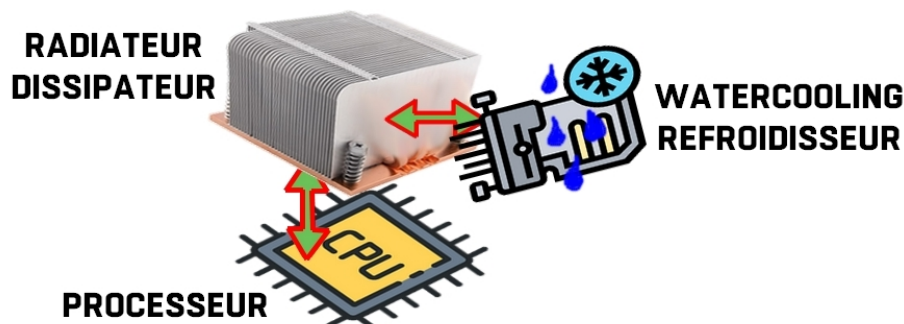
Et, dans la fonction do-xy-plane() :

```
1 static inline void do_xy_plane([.] *T, [.] *R, int v, int n, int m,
2 int o, int k){
3     if (k == 0)
4         // we do not modify the z = 0 plane: it is maintained
5         // at constant temperature via water-cooling
6         return;
```

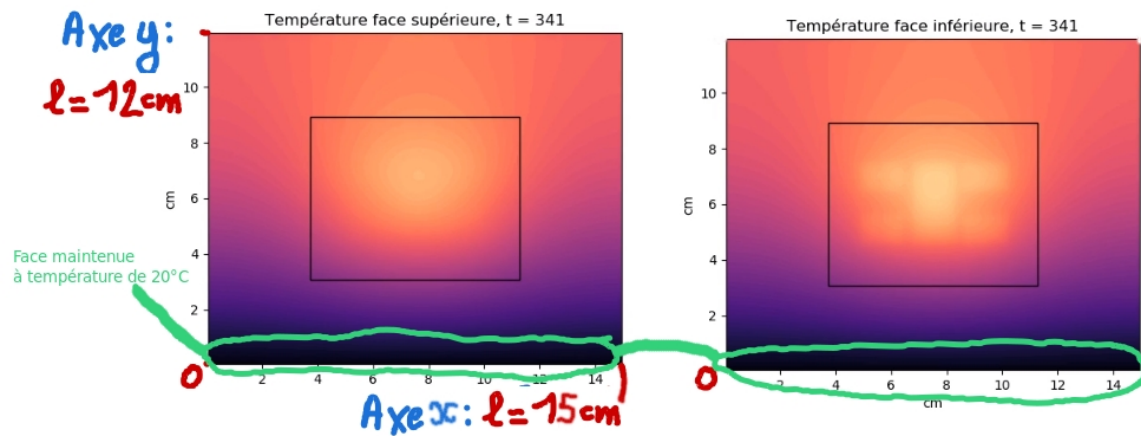
Le commentaire indique que le plan ( $k == 0$ ) correspond au plan ( $z = 0$ ).

Cependant, suivant la logique du code, il devrait plutôt correspondre au plan ( $y = 0$ ).

Néanmoins, cela peut paraître contre-intuitif. Il est ici important de se représenter l'agencement physique de la structure étudiée afin d'avoir l'esprit clair. Le radiateur est positionné sur la face supérieure du CPU. Le dispositif d'accélération du refroidissement n'est ici pas un ventilateur, mais un système de water-cooling (=refroidissement par fluide conducteur). Contrairement aux normes, il est fixé sur un côté du dissipateur plutôt que sur sa face supérieure. Cela est parfois rendu nécessaire par les problématiques de gestion de l'espace dans les racks des centres de calcul. Voici un schéma :



On peut se rendre de compte cela en effectuant un rendu graphique des résultats du programme :



On remarque bien que la face maintenue à température constante  $T = 20^\circ\text{C}$  par watercooling est la face correspondant à  $(y = 0)$  et non  $(z = 0)$ .

Si l'on reprend le code de la fonction `do-xy-plane()` en intégralité :

```
1 static inline void do_xy_plane([.] *T, [.] *R, int v, int n, int m,
2 int o, int k){
3     if (k == 0)
4         // we do not modify the z = 0 plane: it is maintained
5         // at constant temperature via water-cooling
6         return;
7     for (int j = 0; j < m; j++) {           // y
8         for (int i = 0; i < n; i++) {       // x
9             int u = v + j * n + i;
10            R[u] = update_temperature(T, u, n, m, o, i, j, k);}}
```

On remarque que  $j$  allant de 0 à  $m$  est indiqué comme étant l'évolution le long de l'axe  $y$ , alors qu'il s'agit de l'évolution le long de l'axe  $z$ . La fonction ne traite donc pas les plans  $xy$  mais les plans  $zx$ .

Enfin, dans le code faisant la sortie :

```
1 #ifdef DUMP_STEADY_STATE
2     printf("##### STEADY STATE; t = %.1f\n", t);
3     for (int k = 0; k < o; k++) {           // z
4         printf("# z = %g\n", k * dl);
5         for (int j = 0; j < m; j++) {       // y
6             for (int i = 0; i < n; i++) {   // x
7                 printf("%.1f ", T[k * n * m + j * n + i] - 273.15);
8             }printf("\n");}}
```

L'évolution de  $k$  jusqu'à  $o$  (resp. de  $j$  jusqu'à  $m$  et de  $i$  jusqu'à  $n$ ) est décrite comme l'évolution le long de l'axe  $z$  (resp.  $y$  et  $x$ ) alors qu'il s'agit de l'évolution le long de l'axe  $y$  (resp.  $z$  et  $x$ ).

En définitive, cette incohérence n'affecte pas le bon déroulement de l'exécution ni l'exactitude des résultats mais perturbe lorsque l'on procède à la parallélisation. J'ai donc mis à jour les commentaires et changé le nom de `do-xy-plane()` en `do-zx-plane()` pour coller à la logique du code. Les plans traités sont toujours contigus en mémoire par row-major, et les performances ne sont pas affectées.

## 3 Parallélisation 1D : découpe sur une dimension

### 3.1 Idée générale

Dans la présente sous-section, je décris comment j'ai réalisé la parallélisation unidimensionnelle du code en langage informel. L'implémentation en C du code correspondant est le code "parallelisation1D.c" de l'annexe, qui est clairement commenté mais assez complexe.

#### 3.1.1 Principe et intérêt d'une découpe

Dans le programme, les calculs sont effectués pour chaque subdivision arbitraire de l'espace. Ces subdivisions sont appelées cellules. Il est donc évident que pour répartir la charge de calcul, il faut répartir les cellules en affectant chacune d'entre elles à un processeur.

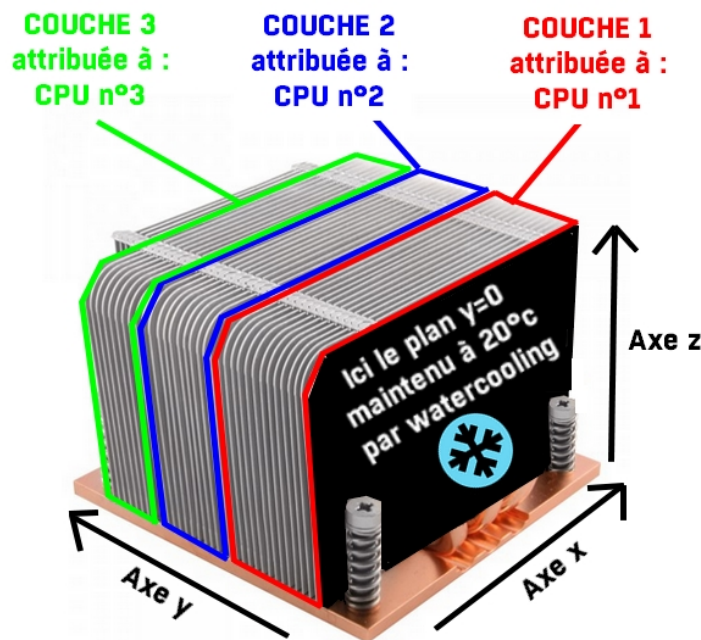
Dès lors, il convient de se demander comment effectuer cette répartition. En l'occurrence, la répartition revient à une découpe plus ou moins régulière du radiateur-dissipateur, qui est l'espace de travail. Faut-il le subdiviser en tranches ? En cubes ? En de toutes autres structures ? Ces structures devraient-elles être semblables entre elles ou non ?

#### 3.1.2 Mode de découpe choisi

La fonction `do-zx-plane()` est telle qu'elle traite les plans  $zx$  successifs pour des valeurs de  $y$  croissantes. Il est alors évident que la parallélisation 1D la plus facile à implémenter est d'attribuer des couches successives le long de l'axe  $y$  aux différents processeurs. Cependant, la division euclidienne ( $n_{couchesY}/n_{cpu}$ ) peut produire un reste  $R$ . Dans ce cas, on augmente de 1 le nombre de couches à traiter par les  $R$  premiers processeurs. Une fois la zone de travail de chaque processus déterminée, on attribue à ce dernier un tableau dont la taille correspond au nombre de cellules dont il a à s'occuper. On lui attribue également 2 ou 4 buffers (1 d'envoi + 1 de réception si le processus ne doit communiquer qu'avec un seul autre, et 2 d'envoi + 2 de réception s'il doit communiquer avec deux autres) pour éviter l'excès de communications point-to-point.

Pour chaque itération de la boucle de convergence, les buffers sont mis à jour et échangés par des `MPI_Sendrecv` afin que chaque processus ait bien accès aux données des régions frontalières de ses voisins directs pour effectuer ses propres calculs. La vérification globale de la convergence est effectuée par un `MPI_Allreduce` des données locales. Quand elle est atteinte, les données sont regroupées, les affichages de sortie sont effectués, et le programme s'arrête.

Voici un exemple de mise en pratique de cette découpe pour 3 CPU disponibles :



## 3.2 Performances du code de base et du code parallélisé sur 1 processus

Pour  $n_{cpu} = 1$ , j'ai comparé les temps d'exécution des deux versions :

Mode	Code de base	Code parallélisé	Temps supplémentaire
FAST	3,63 sec	3,74 sec	+ 3,03%
MEDIUM	60,49 sec	63,58 sec	+ 5,11%
NORMAL	961,44 sec	1058,87 sec	+ 10,13%

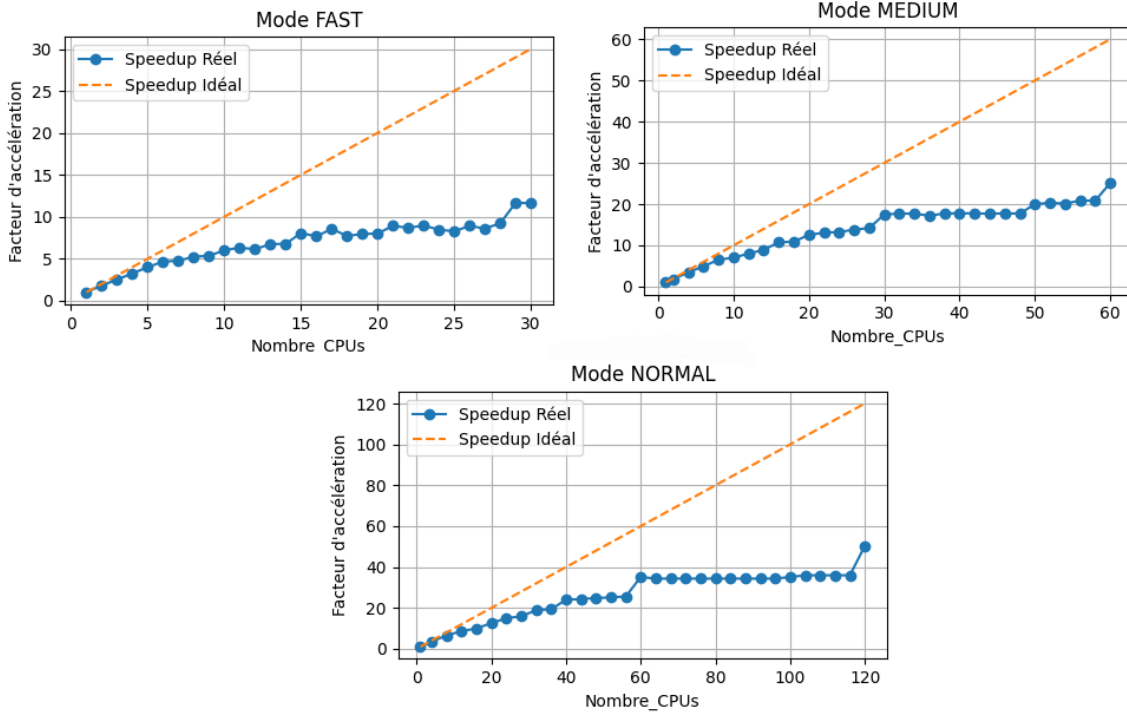
Tests réalisés sur 1 CPU i7-11800H @2,3GHz

Puisque ce surcoût temporel n'est pas constant en termes de secondes, il est probablement dû à l'ajout de comparaisons dans les boucles de traitement des cellules de la version parallélisée et aux appels MPI. Par conséquent, le code parallélisé est moins optimisé pour  $n_{cpu} = 1$  que le code de base, ce qui n'est pas très important puisqu'il est conçu pour être exécuté par plusieurs processus. En revanche, lors du calcul du facteur d'accélération, je prendrai les performances du code originel comme référence pour  $n_{cpu} = 1$  afin d'éviter tout biais.

## 3.3 Performances de la parallélisation

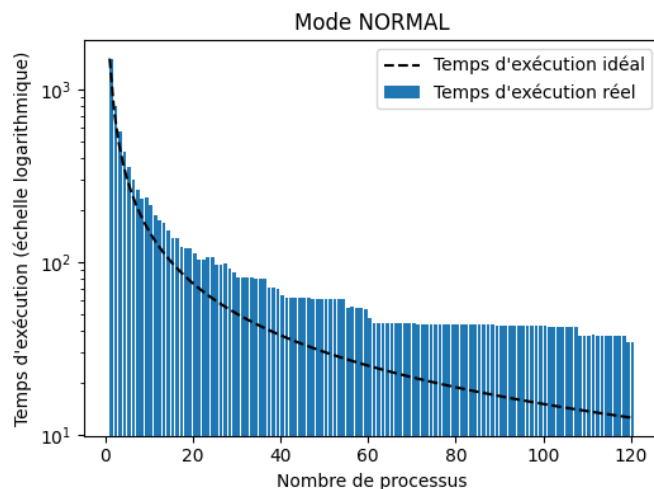
### 3.3.1 Premiers tests

Il m'a paru bon d'effectuer des essais pour les modes FAST, MEDIUM et NORMAL dans un premier temps. Ces modes divisent respectivement le dissipateur en 30, 60 et 120 couches  $y$ . Ma parallélisation 1D impose donc un maximum de 30, 60 et 120 processus dans chacun de ces cas pour ne pas se retrouver avec des processus inutiles sans zone de travail allouée. J'ai effectué les tests sur un réseau de calcul haute performance destiné à la recherche, appelé *Grid5000 (g5K)*. J'y ai accès grâce au laboratoire de programmation parallèle de Sorbonne-Université. J'ai stocké tous les résultats, et voici donc un graphique des facteurs d'accélération observés :



Tests réalisés sur 1 à 120 threads de CPU XeonGold-5220 @2,2GHz, cluster *gros*

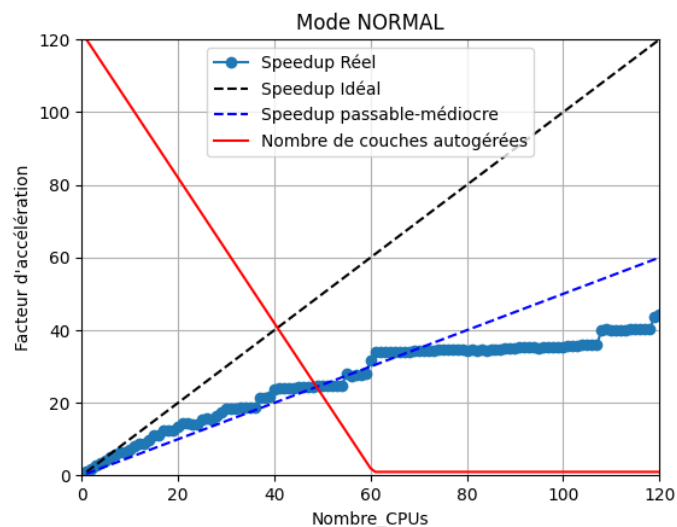
Concentrons-nous sur le mode NORMAL : c'est celui effectuant le plus de calculs des trois, ce qui est intéressant pour la convergence vers les temps de calcul théoriques via la loi des grands nombres. En outre, il s'agit de la meilleure approximation du mode CHALLENGE, qui est le plus précis. En observant l'axe horizontal (nombre de CPUs), on a un facteur d'accélération passable du début jusqu'au tiers, puis très médiocre du tiers à la moitié, et ensuite mauvais de la moitié à la fin. Pour mieux visualiser les données, j'ai refait les tests pour  $n$  allant de 1 à 120 avec un pas de 1. Voici le résultat :



On peut remarquer les quelques soubresauts et l'agencement en plateaux successifs, notamment celui correspondant à la quasi-stagnation de la progression à partir de  $n_{cpu} = 61$ .

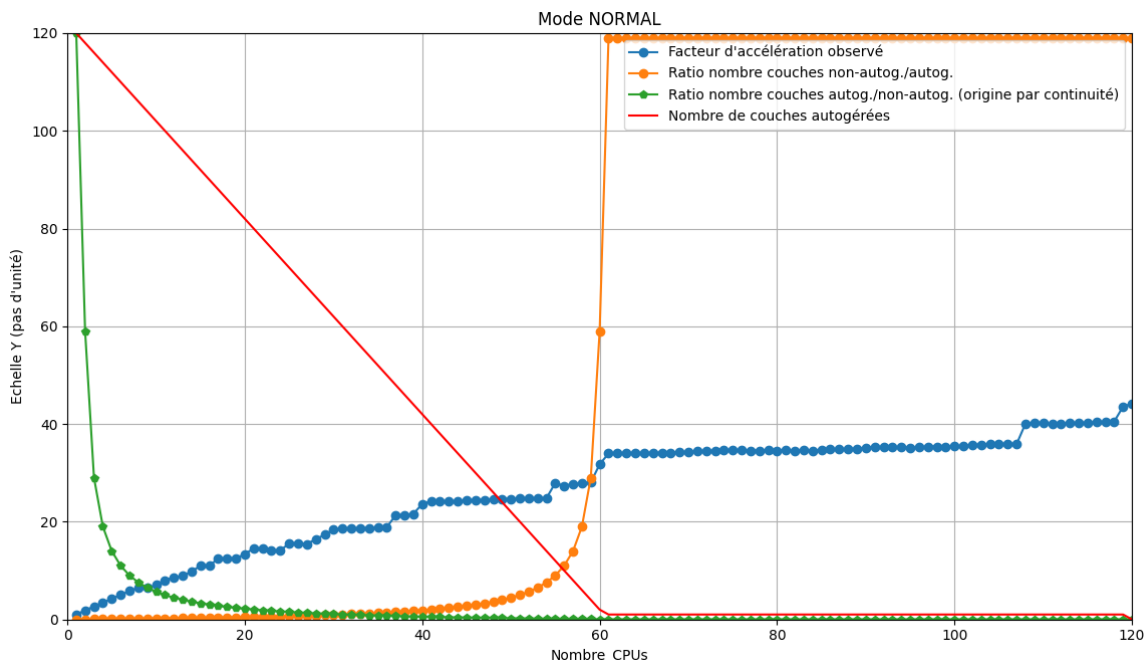
### 3.3.2 Influence du nombre de couches autogérées

Dans le cadre de notre parallélisation, appelons *couche autogérée* toute couche non-frontalière de la zone de travail d'un autre processus. Traçons maintenant le facteur d'accélération :

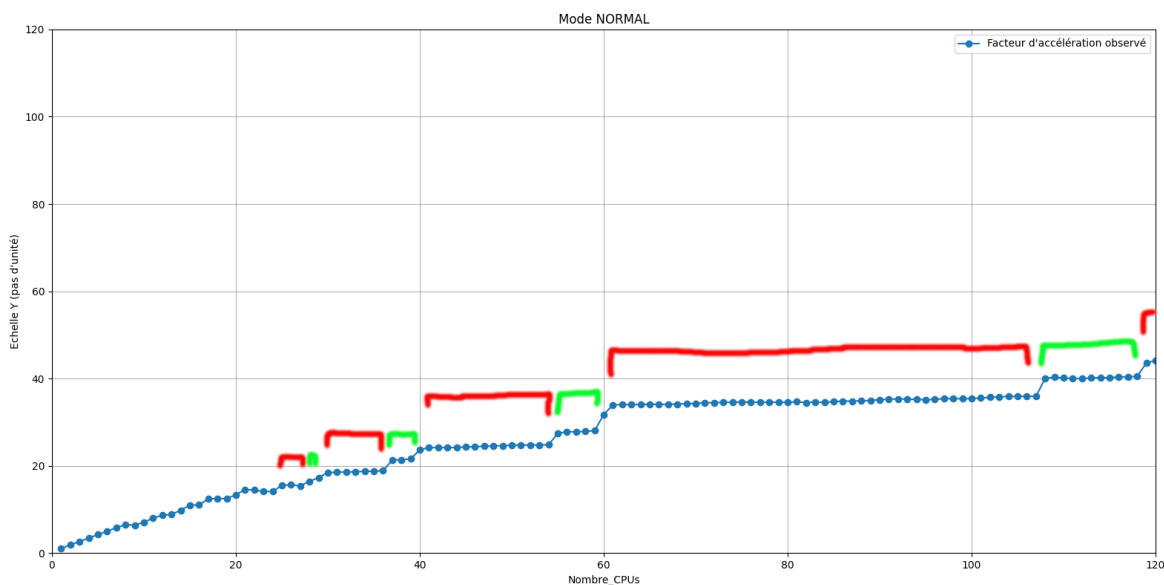


Plus le nombre de couches autogérées diminue, plus le nombre de buffers et de communications MPI à utiliser augmente (proportions inversées). Ainsi, ce surcoût devient maximal quand le nombre de couches autogérées atteint son minimum (1) pour  $n_{cpu} = 61$ . Cela pourrait expliquer en partie le ralentissement sur la deuxième moitié de l'axe X. Néanmoins, l'ampleur de ce surcoût et sa significativité restent incertaines.

Traçons un graphique en meilleure résolution pour explorer cela :



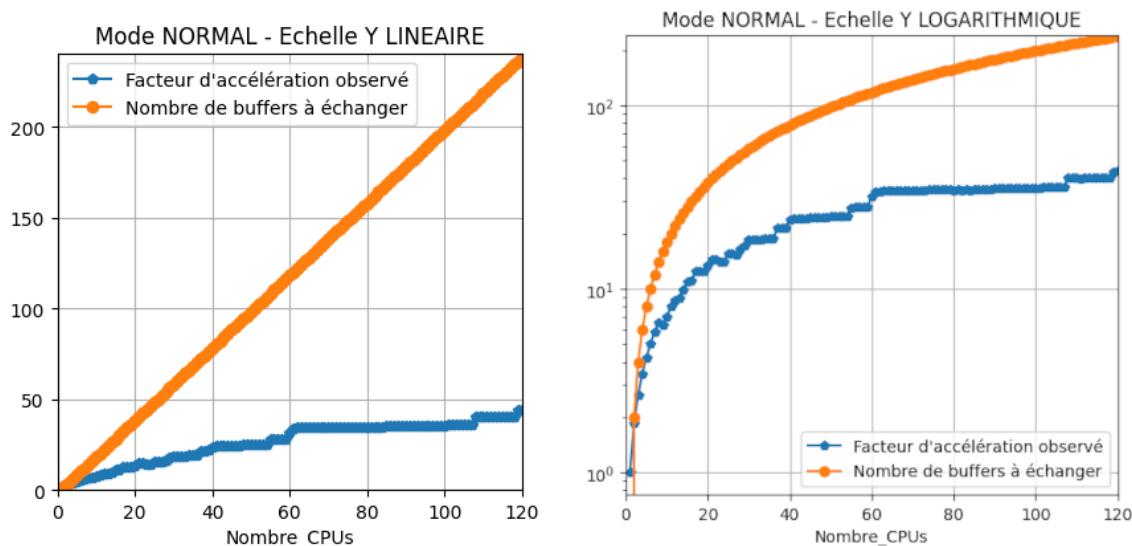
La proportion de couches autogérées ne semble pas affecter significativement notre facteur d'accélération, dont la croissance est logarithmiquement stable (ce qui est une mauvaise chose puisqu'on chercherait dans l'idéal à ce qu'elle soit affine). La courbe d'évolution du facteur d'accélération a le même comportement localement que globalement. On remarque un certain motif à (2+) composantes, que j'ai entouré dans le graphique ci-dessous pour les échelles distinguables ( $n_{cpu}$  suffisamment grand) :



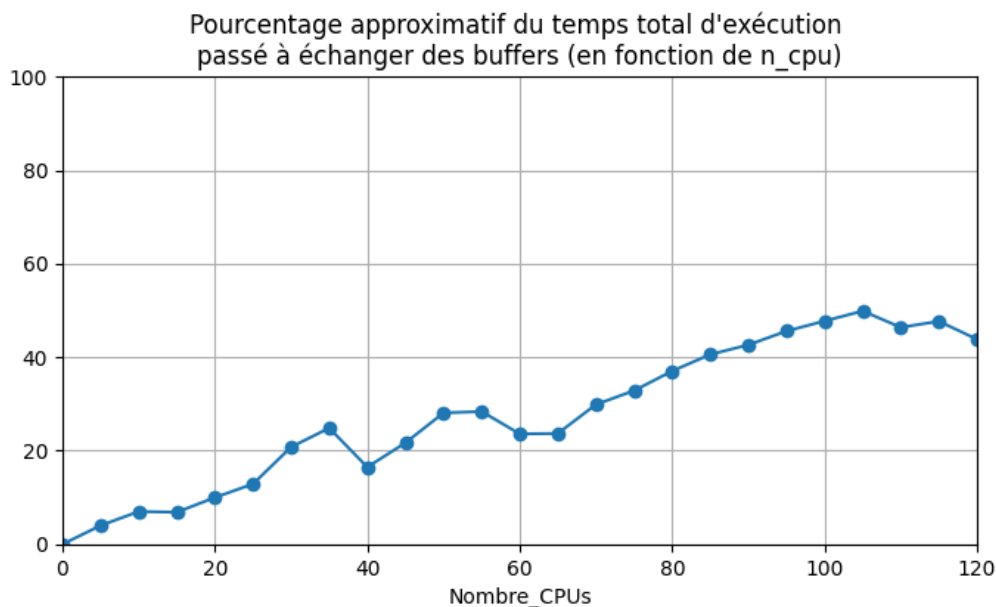
En somme, l'étude plus approfondie permet de comprendre que le temps de calcul ne semble pas être sensible au nombre de couches autogérées, et que la fonction modélisant l'évolution du facteur d'accélération est d'allure logarithmique à motif de paliers, celui à  $n_{cpu} = 61$  n'en étant qu'un parmi bon nombre d'autres. Les couches frontalières ne semblent donc pas causer de surcoût en allant chercher les informations dans les buffers.

### 3.3.3 Influence du nombre de buffers à échanger

Considérons maintenant un autre type de surcoût, le nombre de buffers à échanger. En effet, chaque processus doit en envoyer et en recevoir  $(2 + 2)$  (sauf le premier et le dernier processus qui ne doivent en envoyer et en recevoir que  $(1 + 1)$ ). Le nombre total de buffers à échanger à chaque itération vaut donc  $((n_{cpu} - 1) \times 2)$  par calcul trivial. Traçons cela :



Le facteur d'accélération semble avoir une corrélation avec le nombre de buffers à échanger. Pour poursuivre l'analyse, j'ai inséré dans le code des éléments MPI temporels pour évaluer le temps passé par chacun des processus à échanger ses buffers. Puis, j'ai pris des mesures, j'ai fait les moyennes des distributions et les ai ramenées à des pourcentages pour aboutir à ceci :



On remarque que la courbe tracée présente des sortes de paliers évoquant l'opposé de ceux du facteur d'accélération. Pour savoir si ce dernier est corrélé au pourcentage de temps passé à échanger nos buffers - ce qui paraît probable - on peut calculer des coefficients de corrélation. Cependant, pour une question de simplicité, je n'ai effectué que 24 mesures de mon pourcentage. Effectuons le test statistique en conséquence, une première fois pour un pas de 20 sur  $n_{cpu}$ , puis de 10, et enfin de 5 :



Soit l'hypothèse nulle  $H_0$  : Il n'y a aucune corrélation avec le facteur d'accélération.  
 Soit l'hypothèse alternative  $H_1$  : Il y a une corrélation avec le facteur d'accélération.

Voici le code-type en Python permettant de faire le test (ici, pour un pas de 20) :

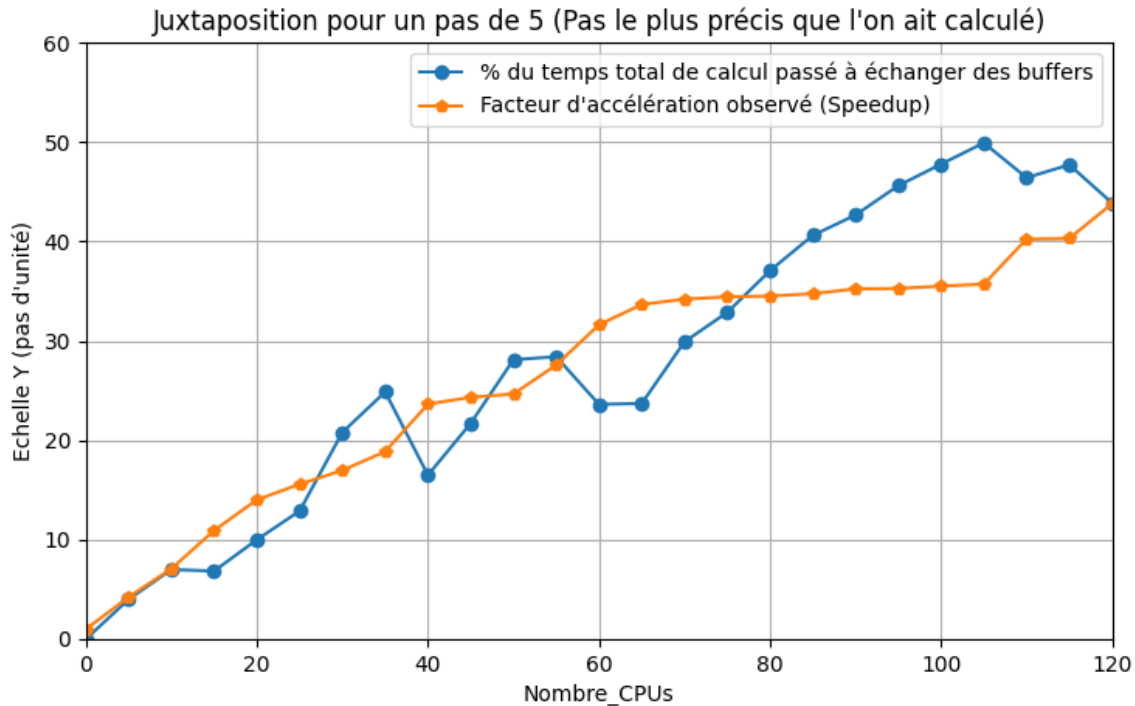
```
1 import scipy.stats as stats
2 # Donnees
3 array1 = [0, 10, 16.5, 23.61, 37.05, 47.76, 43.85]
4 array2 = [1511, 107.8, 63.95, 47.8, 43.8, 42.57, 34.5]
5 # Calcul du coefficient de corrélation de Spearman et p-valeur
6 spearman_corr, p_value = stats.spearmanr(array1, array2)
```

Et voici les résultats :

Pas	Coeff. de Spearman	p-valeur	Rejet de $H_0$
20	0,964	0,000454	Oui
10	0,967	$7,06 \times 10^{-8}$	Oui
5	0,963	$1,33 \times 10^{-14}$	Oui

Le coefficient de Spearman se stabilise autour de 0,96. La p-valeur diminue avec le pas : plus on affine l'aspect de la courbe, plus le rejet de  $H_0$  peut être fait de manière significative.

Ainsi, on a une très forte corrélation (Coefficient très proche de 1) entre le pourcentage de temps passé à échanger des buffers et le facteur d'accélération. Juxtaposons les deux courbes :



On remarque une symétrie des deux courbes par rapport à une courbe logarithmique (non-tracée), à laquelle elles se confondraient si elles n'avaient pas de soubresauts ni de paliers. J'attribue ces derniers à la répartition non-uniforme entre tous les processus des temps d'échange de buffers, les points confondus des deux courbes correspondant donc aux rares valeurs de  $n_{cpu}$  pour lesquelles ces temps sont uniformément répartis. La courbe logarithmique de symétrie sera donc l'approximation du modèle théorique d'accélération.

### 3.3.4 Asynchronisation des échanges de buffers

Nous avons remarqué dans la sous-section précédente qu’une possible (corrélation  $\neq$  causalité) cause de surcoût temporel est le temps passé par les processus dans la partie du code correspondant à l’échange de buffers via des communications bloquantes. Dans un premier temps, nous allons rendre ces communications non-bloquantes afin de permettre aux processus de mettre à jour les couches autogérées dans l’attente de la réception des buffers nécessaires à la mise à jour des couches frontalières.

Le code synchrone est de forme suivante :

---

**Algorithm 1:** Fonctionnement de la boucle de simulation du code synchrone

---

**while** (*convergence*==0) **do**

- 1) Mise à jour (calcul) de toutes les cellules de l’espace de travail du processus courant;
  - 2) Mise à jour de tous les buffers avec les nouvelles températures et échange de ces derniers avec les autres processus via des `MPI_Sendrecv`;
  - 3) Test de convergence (et affichage de l’avancement pour P0);
  - 4) Mise à jour des tableaux locaux du processus courant (`Local_R` et `Local_T`) ;
- 

Et voici ma version asynchronisée (fichier "parallelisation1Dasynchrone.c" de l’annexe) :

---

**Algorithm 2:** Fonctionnement de la boucle de simulation du code asynchrone

---

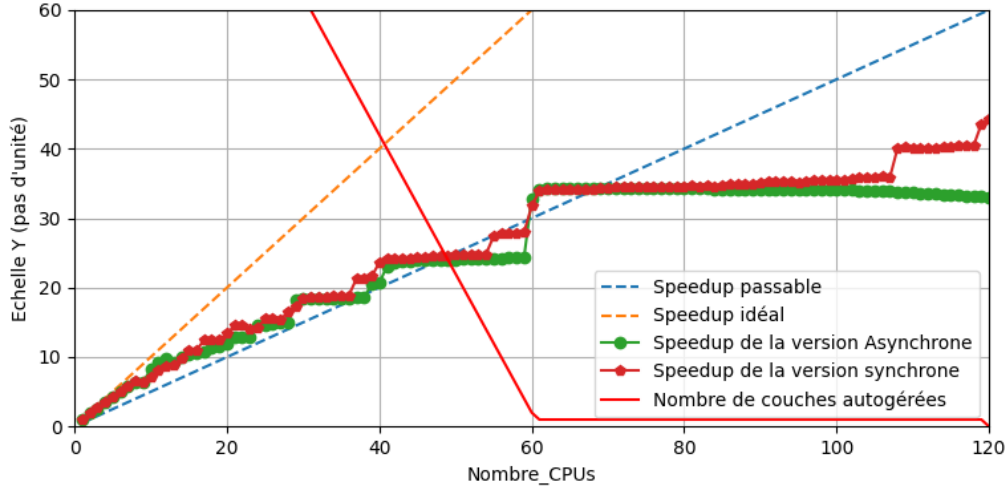
**while** (*convergence*==0) **do**

- 1) Demande de réception des buffers dont le processus courant a besoin pour faire ses calculs de manière non-bloquante via des `MPI_Irecv`;
  - 2) Mise à jour (calcul) de toutes les cellules de l’espace de travail du processus courant qui ne sont pas frontalières de la zone de travail d’un autre processus et qui ne nécessitent donc pas de buffers;
  - 3) Attente de la réception des buffers demandés en 1) via des `MPI_Wait`;
  - 4) Mise à jour (calcul) des cellules de l’espace de travail du processus courant qui sont frontalières de la zone de travail d’un autre processus et qui nécessitaient donc ces buffers;
  - 5) Mise à jour de tous les buffers avec les nouvelles températures et demande d’envoi de ces derniers aux autres processus de manière non-bloquante via des `MPI_Isend`;
  - 7) Test de convergence (et affichage de l’avancement pour P0);
  - 8) Mise à jour des tableaux locaux du processus courant (`Local_R` et `Local_T`) ;
- 

J’ai fait le choix de ne pas inclure de `MPI_Barrier` (qui aurait pu servir à s’assurer que chaque clôture d’itération soit faite simultanément par les processus) car les décalages observés (dans toutes les configurations possibles) sont minimes et uniformément répartis autour de 0, ce qui assure leur annulation mutuelle par LGN, et donc la préservation du résultat souhaité.

La prochaine étape consiste à effectuer les mêmes mesures pour le code asynchrone que celles préalablement effectuées pour le code synchrone, afin de pouvoir comparer les performances des deux.

Voici ces mesures juxtaposées :



Tests réalisés sur 1 à 120 threads de CPU XeonGold-5220 @2,2GHz, cluster *gros*

Les soubresauts n'adviennent pas exactement pour la même valeur de  $n_{cpu}$ . De plus, lorsque le nombre de couche autogérée tombe à 0, le programme asynchrone voit ses performances stagner. Mais le plus intéressant est le fait que les deux programmes présentent des performances sensiblement identiques :

Dans la partie **4.3.3**, on a vu que le temps passé par le programme dans sa partie bloquante d'échange de buffers était une cause importante de surcoût temporel.

Or, en rendant ces échanges non-bloquants, les performances ne se sont pas sensiblement améliorées. Donc, le surcoût est davantage imputable aux délais de transmission des buffers qu'à l'attente inter-processus.

Chaque CPU utilisé sur le cluster *gros* peut se trouver sur un *node* différent. Le cas échéant, la transmission des buffers est faite par le réseau, en l'occurrence de l'Ethernet basique. Il serait donc intéressant de tester de nouveau les performances avec un réseau haute-performance.

J'ai choisi d'effectuer ce test comparatif sur les clusters *grisou* et *grimoire* puisque leurs nodes ont les mêmes CPU et la même RAM, ce qui garantit des performances identiques pour le programme séquentiel. En revanche, le premier dispose d'un réseau de  $(4 \times 10 + 1)Gb/s$ , contre  $(1 \times 56)Gb/s$  pour le second (soit 36,6% de plus). Voici les résultats obtenus :

Configuration testée	41Gb Ethernet, <i>grisou</i>	56Gb Infiniband, <i>grimoire</i>	Gain de temps
1CPU - Code Séquentiel	2020,5 sec	2020,8 sec	0%
15CPU - Code Synchrone	182,1 sec	182,5 sec	0%
15CPU - Code Asynchrone	182,8 sec	182,9 sec	0%
60CPU - Code Synchrone	65 sec	53,2 sec	18,2%
60CPU - Code Asynchrone	66,1 sec	57,4 sec	14,2%
100CPU - Code Synchrone	65,2 sec	47,6 sec	27%
100CPU - Code Asynchrone	67,7 sec	51,7 sec	23,7%
120CPU - Code Synchrone	63 sec	38,4 sec	31,1%
120CPU - Code Asynchrone	64,3 sec	50,7 sec	21,2%

Tests réalisés sur des CPU Xeon E5-2630 v3 @2,4GHz

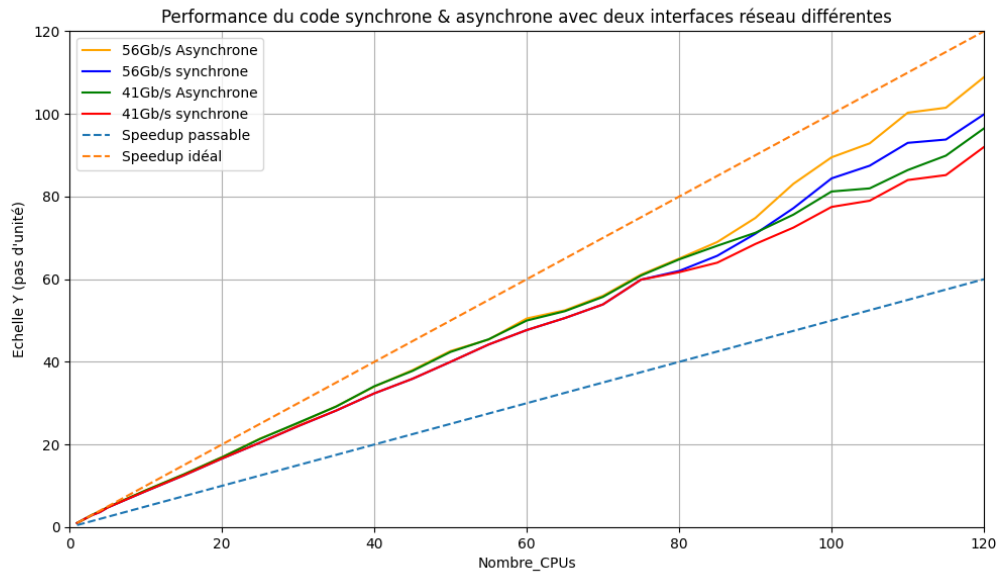
On a ainsi la preuve que le réseau influence les performances. Il reste maintenant à tester cela en mode challenge : puisqu'il est plus précis, la taille des buffers/zones à traiter/itérations augmente.

### 3.4 Tests en mode CHALLENGE

Il y a deux obstacles à surmonter en mode challenge :

- Le nombre d'itérations est très grand, alors plutôt que de mesurer le temps total d'exécution, on peut calculer le facteur d'accélération à partir du temps moyen passé à faire les  $X$  premières itérations (ici, on effectuera les 1000 premières 20 fois pour chaque mesure afin d'assurer une convergence par LGN).
- On a ( $n_{couchesY} = 1200$ ), mais *g5K* ne propose aucun cluster à 1200 cores avec un réseau HPC pour le niveau de privilège basique. On restreint donc les mesures à la portion de  $n_{cpu}$  présentant le facteur d'accélération le plus convaincant jusqu'ici, soit de 0 à 1/10, ce qui correspond à 120 CPU maximum.

Voici les mesures sur les clusters *grisou* (41Gb/s) et *grimoire* (56Gb/s) pour un pas de 5 sur  $n_{cpu}$  :



Voici mes observations sur la plage  $n_{cpu} \in [1, 120]$  :

- Chacune des 4 parallélisations est satisfaisante, le pire facteur d'accélération observé représente 75% du facteur idéal.
  - Le code asynchrone semble toujours un peu plus performant que le code synchrone. Cela s'inverse nécessairement à partir de  $n_{cpu} = 601$ , seuil à compter duquel il n'y a plus de couche autogérée que le code asynchrone peut traiter prioritairement (ce dernier n'apporterait donc plus que les surcoûts associés à son surplus d'appels MPI).
  - À partir de  $n_{cpu} = 80$ , les performances sont meilleures pour un réseau de 56Gb/s que de 41Gb/s. Cela correspond logiquement au moment où le réseau est saturé par les échanges de buffers. Plus son débit est élevé, plus cette saturation advient tardivement sur l'axe x (correspondant à  $n_{cpu}$ ). Autrement dit, le réseau **Ethernet** sature plus vite que le réseau **Infiniband**, qui saturera lui-même plus vite qu'un réseau **Omni-Path**, qui saturera lui-même plus vite qu'un réseau à plus haut débit, et ainsi de suite. Le débit du réseau joue donc un rôle crucial dans cette parallélisation. Si l'on suppose qu'il est illimité, alors on peut estimer que le facteur d'accélération vaut 80 à 85 % du facteur d'accélération linéaire idéal.
- NB : La part d'influence du débit réseau est pondérée par celle de la puissance de calcul des composants, et vice versa. C'est pour cela que les améliorations de performances suivent l'accroissement du débit sans se calquer dessus.

## 4 Diagnostic prévisionnel des découpes en dimension $> 1$

### 4.1 Parallélisation 2D

Selon la logique de définition et de découpe row-major, une parallélisation 2D n'assure pas nécessairement un échange de portions contiguës en mémoire. De plus, la complexité du code augmenterait nettement par rapport à une parallélisation 1D : la taille des buffers n'étant plus fixe, leur gestion serait bien plus compliquée. Ensuite, la potentielle amélioration ne serait nette qu'au-delà du seuil ( $n_{cpu} > n_{couchesY}$ ), palier à partir duquel la parallélisation 1D n'alloue plus d'espace de travail aux CPU supplémentaires. Mais, en mode challenge, on a ( $n_{couchesY} = 1200$ ), ce qui pousse ledit seuil trop loin pour les effectifs de nos clusters. En définitive, cette implémentation 2D prendrait beaucoup de temps, alors que les améliorations qu'elle apporterait ne sont qu'incertaines.

### 4.2 Parallélisation 3D

Je vois deux manières d'aborder une découpe 3D.

La première possibilité, assez simple dans l'idée, serait d'effectuer une découpe en sous-parallélépipèdes répartis de la manière la plus uniforme possible avec une gestion du reste. Cela ne serait pas catastrophique pour la contiguïté, mais ne se contrôlerait pas de la taille des buffers à échanger.

La deuxième possibilité serait de trouver le meilleur compromis, sachant que l'on veut maximiser le nombre de coupes (pour pouvoir utiliser le plus de CPUs possibles) et minimiser la somme des surfaces de contact desdites coupes (pour utiliser le moins de communications MPI possible). On opterait donc pour une découpe en zigzag. En effet, en alternant les dimensions des coupes, on pourrait augmenter le nombre de ces dernières tout en minimisant la somme de leurs surfaces de contact. Néanmoins, l'efficacité de cette technique dépend du parallélépipède rectangle considéré, et la contiguïté en mémoire n'est absolument pas contrôlée.

Dans les deux cas, l'implémentation prendrait beaucoup de temps, et de nouveaux surcoûts apparaîtraient. Selon moi, tout comme la parallélisation 2D, elle n'aurait d'intérêt que quand le nombre de CPU disponible est nettement supérieur au nombre de couches sur l'axe  $y$  ET QUE le débit réseau disponible permet une non-saturation.

En somme, cela n'aurait d'intérêt que lorsque deux conditions sont remplies :

- ( $(n_{cpu} \gg n_{couchesY})$ )
- ( $\text{débit\_réseau\_disponible} \geq \text{débit\_réseau\_saturable}$ )

Autrement, la parallélisation 1D présenterait des performances relativement proches, ce qui est à considérer à l'aune de sa bien plus grande facilité d'implémentation et de maintenance. Une gestion plus poussée des buffers et du cache pourrait néanmoins être étudiée puisque l'on est ici memory-bound.