Projet 1 de Parallel Programming

Tristan Chenaille, MAIN4, Numéro étudiant 21217548

1 Thématique

Nous nous intéresserons au radiateur (= dissipateur de chaleur métallique) d'un CPU (= processeur) de modèle "*AMD EPYC Rome*" dans ce travail. Nous considérerons qu'il s'agit d'un parallélépipède rectangle par souci de simplification, et qu'il est composé de cuivre puisqu'il s'agit du métal standard pour les serveurs haute-performance (meilleur compromis parmis les métaux possibles). L'idée est d'allumer notre 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 fourni 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). L'objectif du présent projet est de paralléliser ce code, c'est à dire d'en répartir l'exécution de sur un nombre $n \geq 1$ de processus. Pour ce faire, nous utiliserons la librairie *OpenMPI* en *C*, qui fait correspondre n à n_{cpu}.

2 Relevé d'incohérence dans le code fourni

En essayant de paralléliser le code pour la première fois en 1D (en divisant le radiateur en zones de travail le long d'un seul axe), j'ai rencontré une aporie. En effet, le code fourni effectue ceci dans les définitions de constantes :

```
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 ceci au début de la fonction main() :

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

Qui fait donc correspondre n (resp. m, o) aux pas sur les axes x (resp. z, y).

Ensuite, dans la boucle de simulation de la fonction main() :

```
/* Update all cells. xy planes are processed, for increasing values of z. */
for (int k = 0; k < o; k++) {   // z
    int v = k * n * m;
    do_xy_plane(T, R, v, n, m, o, k);
}
```

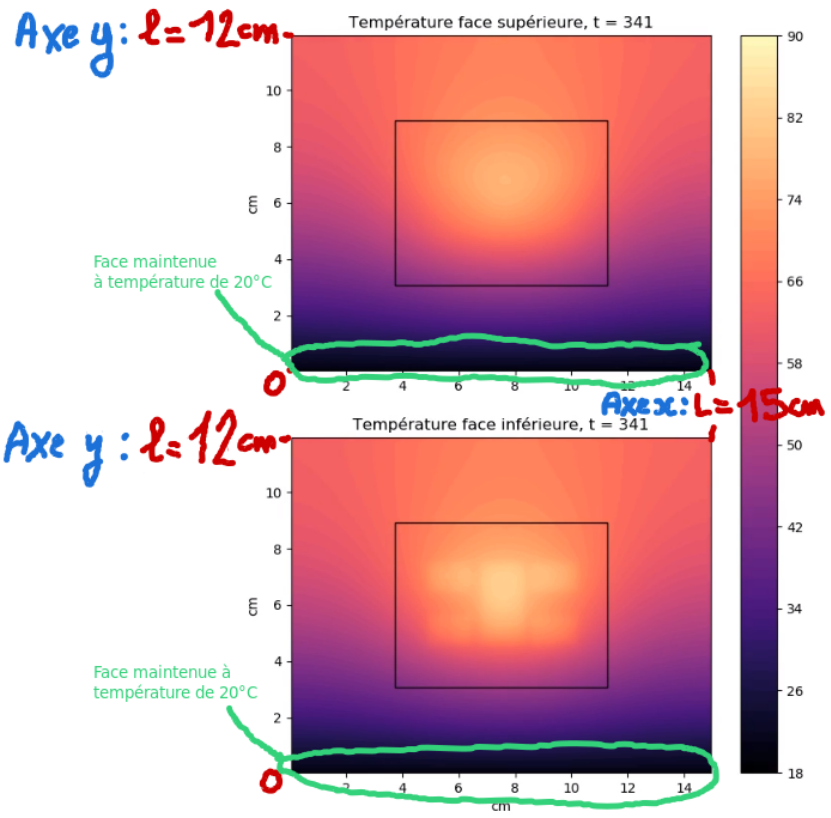
Le commentaire indique que tous les plans xy sont traités, et ce pour des valeurs de z croissantes. Cependant, la variable k augmente jusqu'à o, qui est le pas spatial pour l'axe y et non pour l'axe z.

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

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

Puis, on peut observer l'exemple de rendu graphique donné sur le moodle (j'ai ajouté les annotations pour mieux visualiser) :



On remarque grâce à ce rendu que la face maintenue à température constante $T = 20°C$ par watercooling est la face correspondant à $(y = 0)$ et non $(z = 0)$. Le dispositif de watercooling est donc en contact avec un et un seul côté du dissipateur, ce qui est assez inhabituel ; il est souvent apposé à la face supérieure.

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

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

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

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 fichier "parallelisation1D.c" du répertoire courant, qui est assez commenté pour que lecteur puisse se passer de la lecture de la présente sous-partie s'il manque de temps.

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 processus. 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 processus. 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 l'exécution s'arrête.

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%

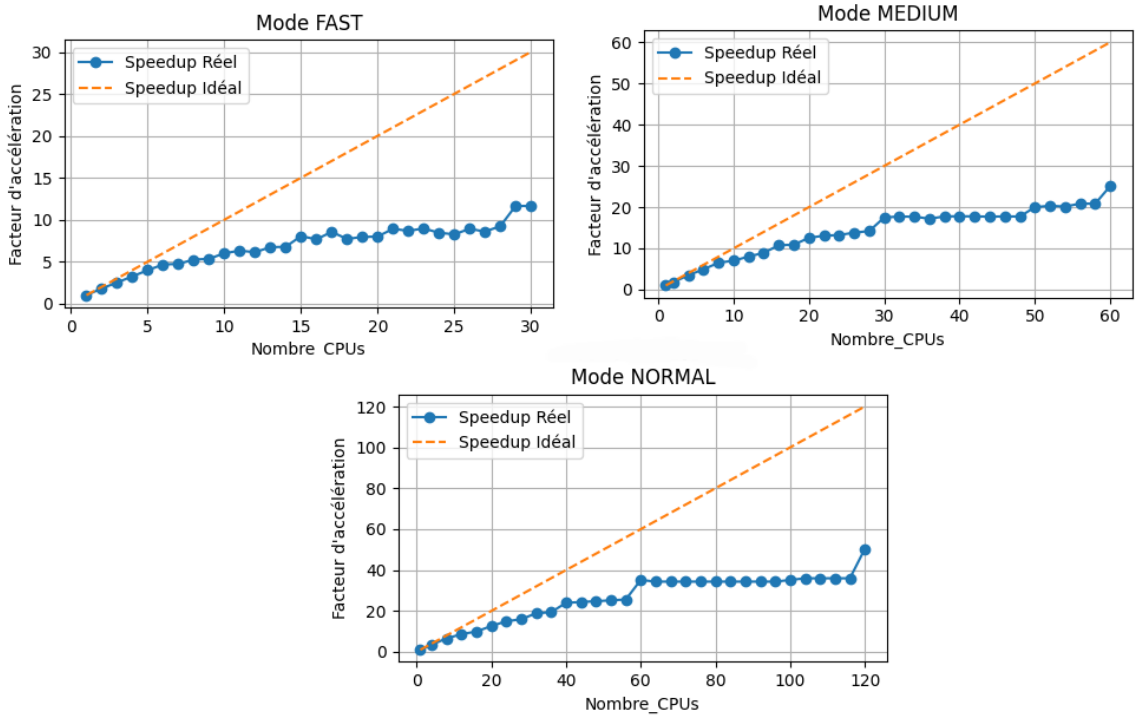
Tests réalisés sur 1 CPU i7-11800H @2,3GHz

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 éxécuté par plusieurs processus. En revanche, lors du calcul du facteur d'accélération, il faudra prendre les performances du code originel comme référence pour $n_{cpu} = 1$ afin d'éviter tout biais.

3.3 Performances de la parallélisation

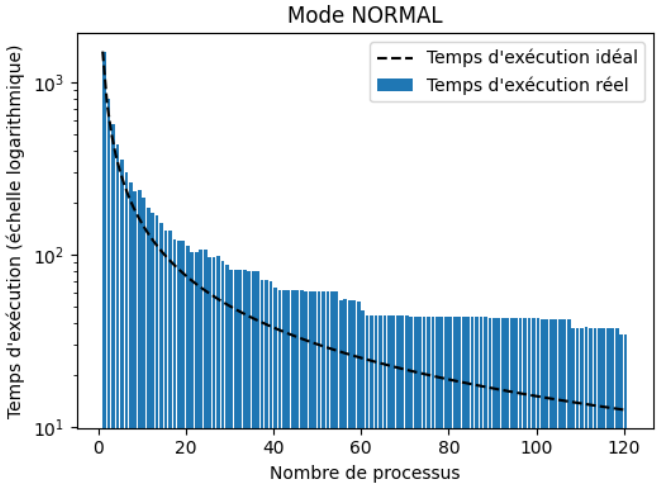
3.3.1 Premiers tests

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 $g5K$ et ai stocké tous les résultats. Voici un graphique des facteurs d'accélération observés :



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

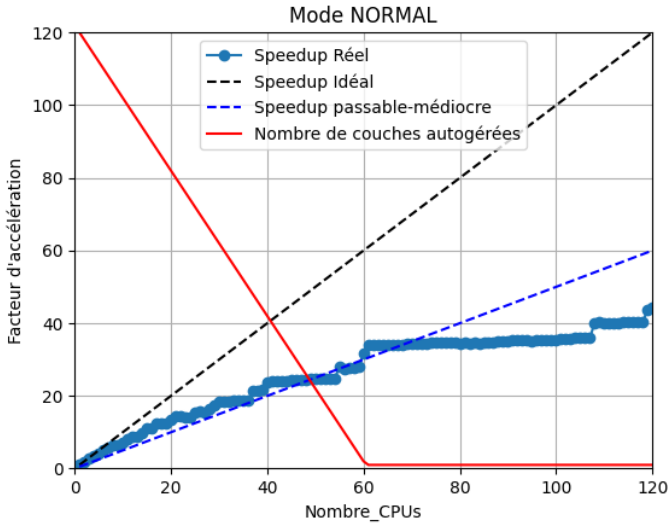
Concentrons-nous sur le mode NORMAL : c'est celui effectuant le plus de calculs des trois, ce qui est intéressant pour la convergence vers les temps de calcul théoriques via la loi des grands nombres. En outre, il s'agit de la meilleure approximation du mode CHALLENGE, qui est l'objectif de notre projet. En faisant nos observations de 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 $n_{cpu} = 61$.

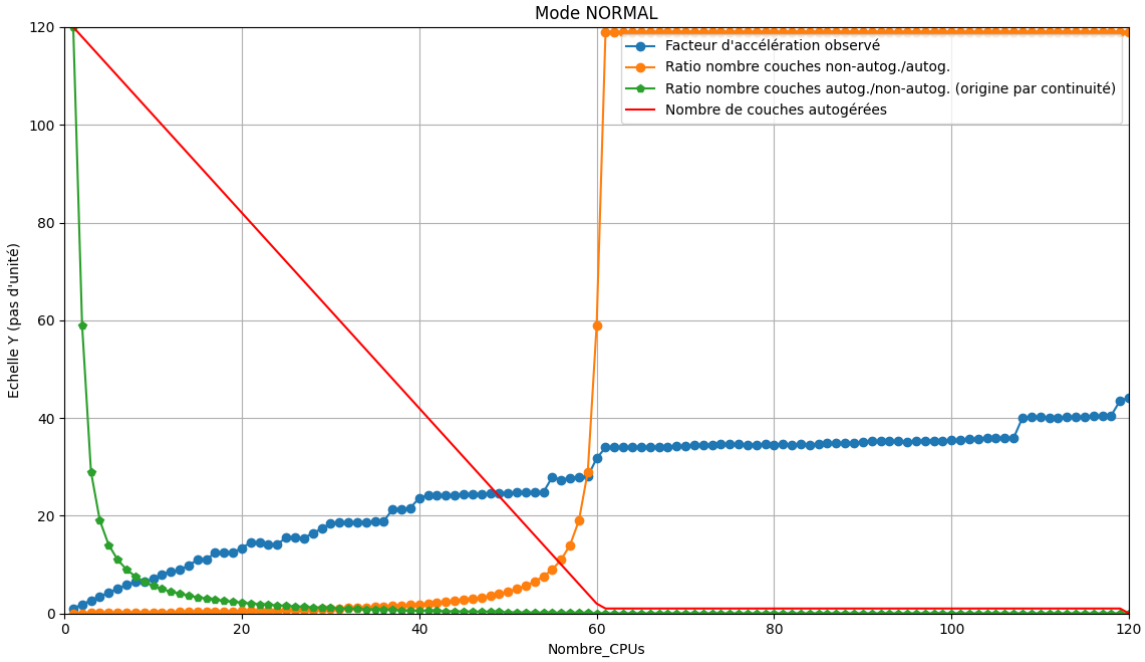
3.3.2 Influence du nombre de couches autogérées

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

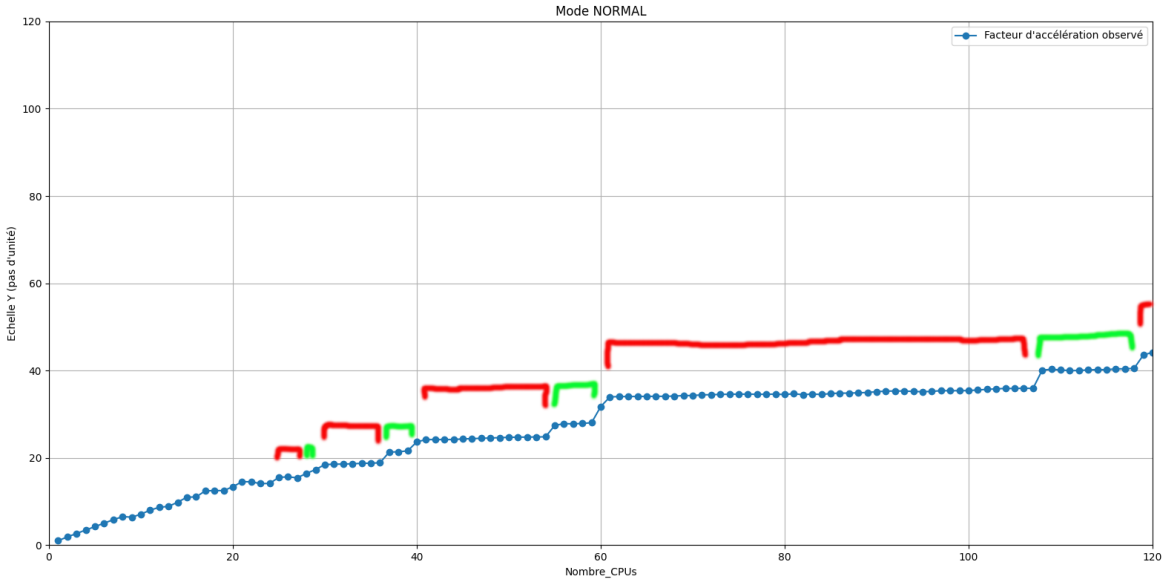


Plus le nombre de couches autogérées diminue, plus le nombre de buffers et de communications MPI à utiliser augmente (proportions inversées). Ainsi, ce surcoût devient maximal quand le nombre de couches autogérées atteint son minimum (1) pour $n_{cpu} = 61$. Cela pourrait expliquer en partie le ralentissement 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 :



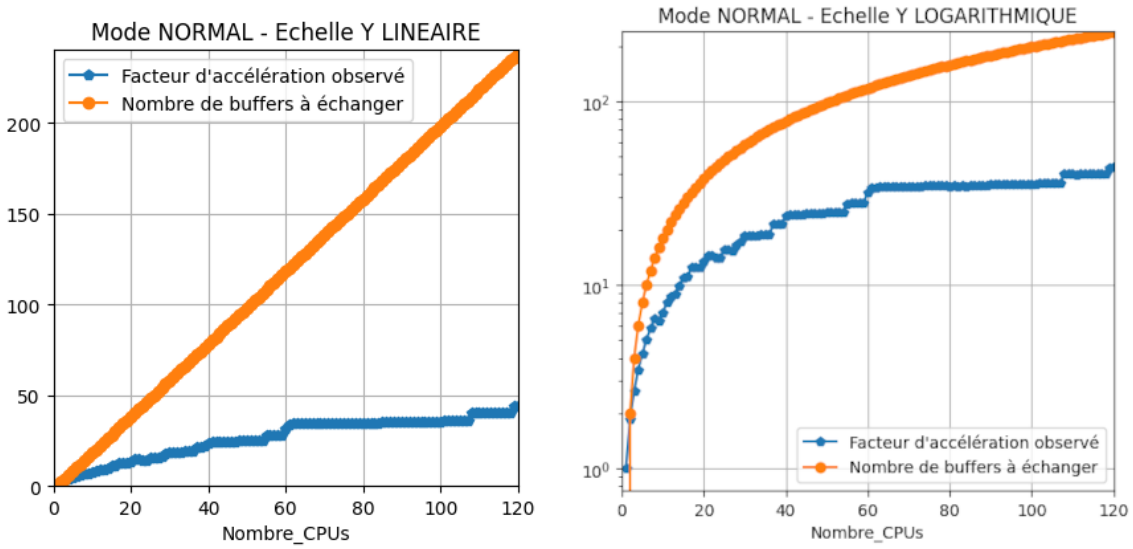
La proportion de couches autogérées ne semble pas affecter significativement notre facteur d'accélération, dont la croissance est logarithmiquement stable (ce qui est une mauvaise chose puisqu'on chercherait dans l'idéal à ce qu'elle soit affine). La courbe d'évolution du facteur d'accélération a le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



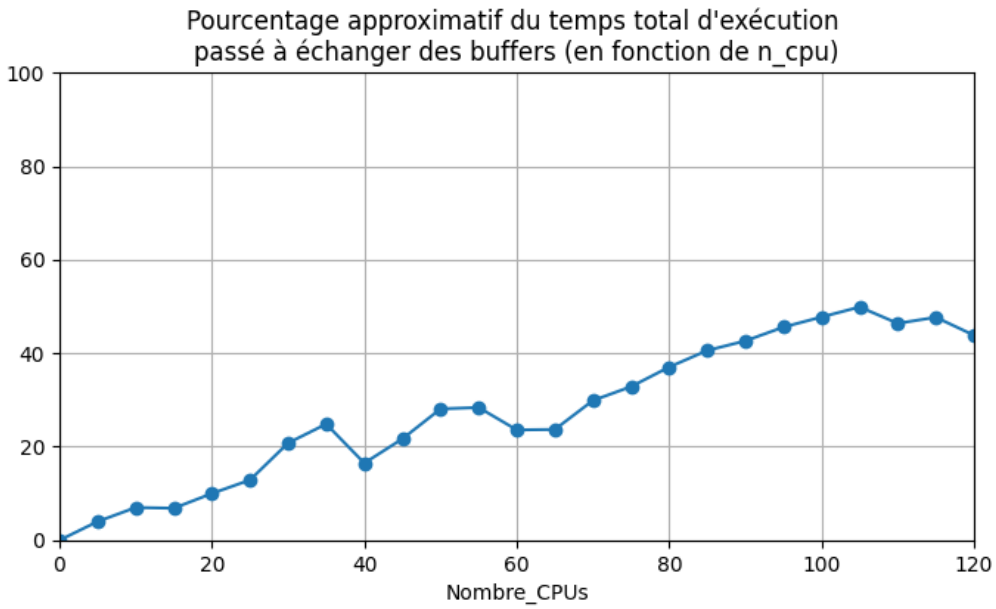
En somme, l'étude plus approfondie nous a permis de comprendre que le temps de calcul ne semble pas être sensible au nombre de couches autogérées, et que la fonction modélisant l'évolution du facteur d'accélération est d'allure logarithmique à motif de paliers, celui à $n_{cpu} = 61$ n'en étant qu'un 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 :



Notre facteur d'accélération semble avoir une corrélation avec le nombre de buffers à échanger. Pour poursuivre l'analyse, j'ai inséré dans mon 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) :

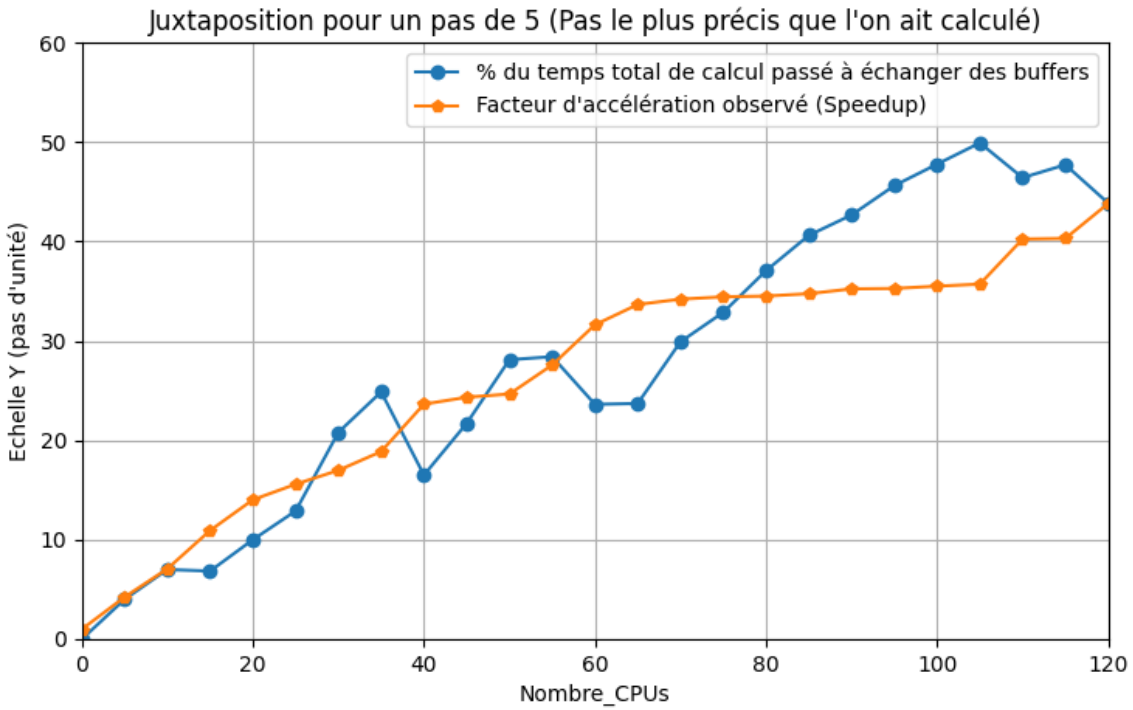
```
import scipy.stats as stats
# Donnees
array1 = [0, 10, 16.5, 23.61, 37.05, 47.76, 43.85]
array2 = [1511, 107.8, 63.95, 47.8, 43.8, 42.57, 34.5]
# Calcul du coefficient de correlation de Spearman et p-valeur
spearman_corr, p_value = stats.spearmanr(array1, array2)
```

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 notre 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 de notre 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" du répertoire courant) :

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`;

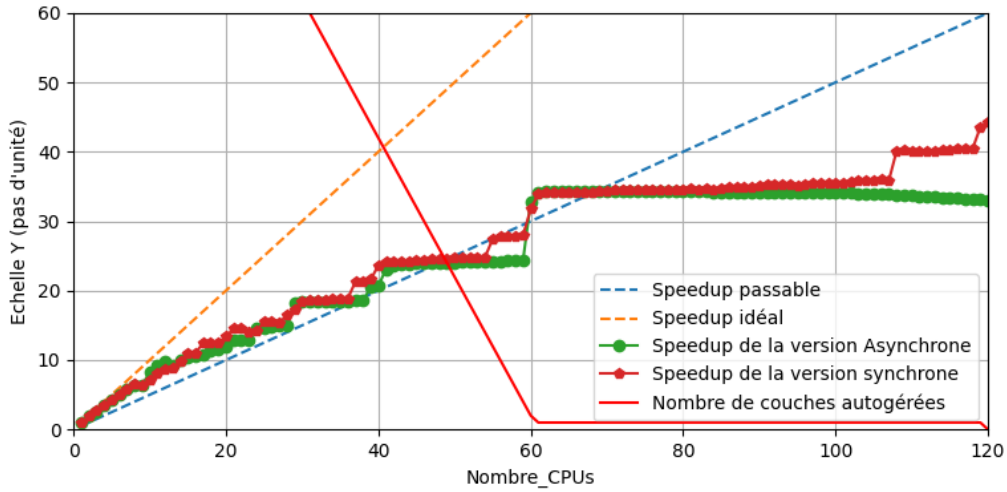
7) Test de convergence (et affichage de l'avancement pour `P0`);

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**, nous avons 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, notre transmission des buffers est faite par le réseau, en l'occurence 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, *grisou*	56Gb Infiniband, *grimoire*	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

Nous avons la preuve que le réseau influence les performances. Il reste maintenant à tester cela en mode challenge puisque c'est notre objectif et que la taille des buffers/zones à traiter/itérations augmente.

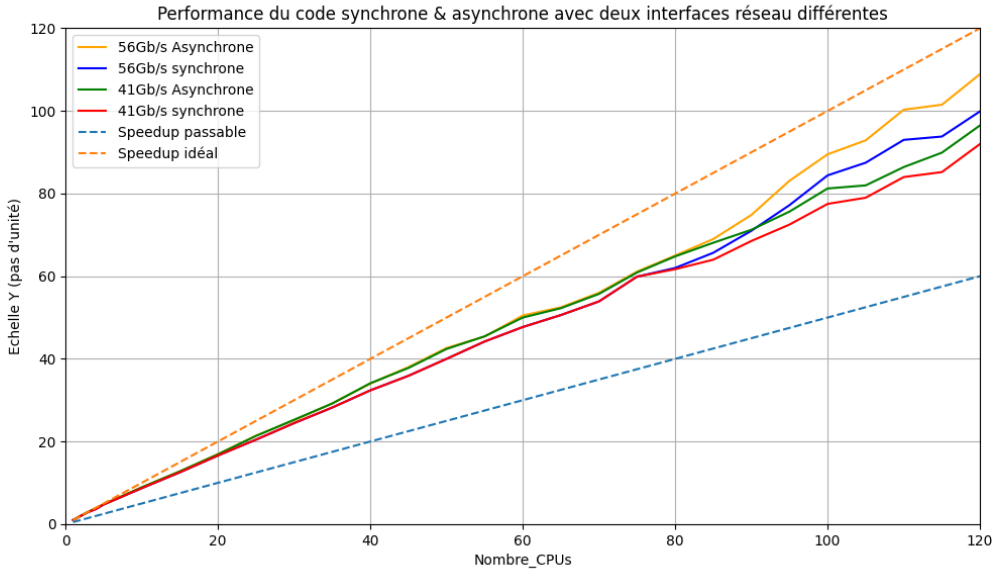
3.4 Tests en mode CHALLENGE

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 les obserations sur notre 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 surcouts 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 $41Gb/s$. Selon moi, cela correspond au moment ou 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, notre réseau `Ethernet` sature plus vite que notre 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 role crucial dans notre parallélisation. Si l'on suppose qu'il est illimité, alors on peut estimer que notre 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 Parallélisation 2D

Selon notre logique de définition et de découpe, une parallélisation 2D n'assure pas nécessairement un échange de portions contigües 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, je pense que cette implémentation 2D prendrait pas mal de temps, alors que les améliorations qu'elle apporterait ne sont que très incertaines.

5 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 garantirait une relative contiguïté en mémoire des zones traitées, 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 lorsque :

$((n_{cpu} \gg n_{couchesY})\ and\ (\text{débit\_réseau\_disponible} \geq \text{débit\_réseau\_saturable})) == 1$