Programmation hybride MPI + X

Roland KIA Emmy Prats

16 janvier 2026

1 Version séquentielle

Dans un premier temps, nous étudions les débordements de cache de la version séquentielle du stencil. Sur l'architecture considérée, chaque cœur dispose d'un cache de données de niveau L1 (L1d) de 32 KB, ainsi que d'un cache de niveau L2 de 256 KB.

Le calcul est effectué sur une grille carrée de taille $N \times N$. L'algorithme utilise deux tableaux distincts, `values` et `prev_values`, qui sont alternativement lus et écrits à chaque itération. Chaque élément du tableau est de type `stencil_t`, correspondant à un `float`, soit 4 octets.

La quantité totale de mémoire manipulée est donc donnée par :

$$M(N) = 2 \times N \times N \times \text{sizeof(stencil\_t)} = 8N^2 \text{ octets.}$$

Dans une approche théorique idéale, on suppose que l'intégralité des données nécessaires tient simultanément dans le cache.

Pour que l'ensemble des données tienne dans le cache L1, il faut alors vérifier :

$$8N^2 \leq 32\,768 \implies N \leq 64.$$

De manière analogue, pour que les données tiennent intégralement dans le cache L2, la condition devient :

$$8N^2 \leq 262\,144 \implies N \leq 181.$$

Ces valeurs constituent des bornes théoriques pessimistes, dans la mesure où elles supposent que l'ensemble des deux tableaux est simultanément présent dans le cache, sans tenir compte de la localité spatiale et temporelle induite par le parcours ligne par ligne du stencil.

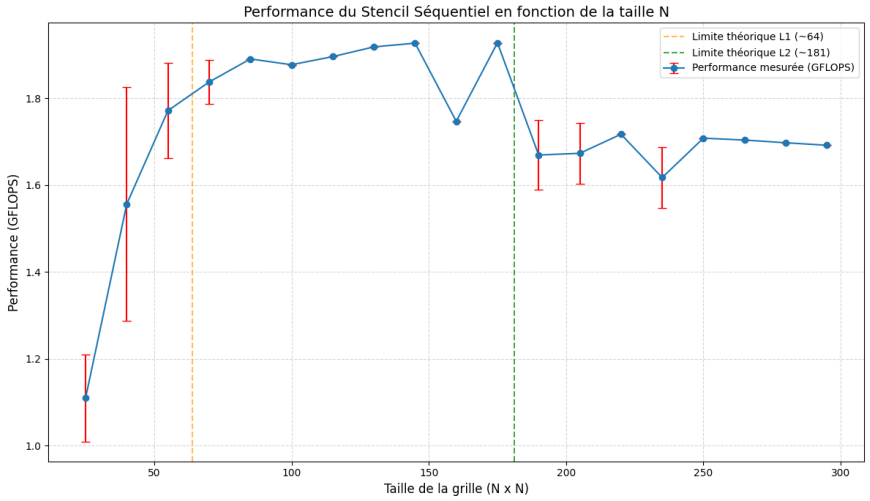

FIGURE 1 – GFLOPS moyen sur 3 itérations

En pratique, comme l'illustre la figure 1, les résultats expérimentaux confirment les prédictions théoriques concernant les sorties successives des différents niveaux de cache. En particulier, la sortie du cache L2 est clairement identifiable par une chute brutale du plateau de performances.

On observe également qu'entre les deux asymptotes verticales, correspondant respectivement aux dépassements des caches L1 et L2, les performances restent stables et maximales, atteignant environ 1,9 GFLOPS.

2 Version MPI

Pour la version MPI pure, le domaine global est découpé suivant l'axe des ordonnées (y) en blocs de lignes. Lorsque le nombre de lignes n'est pas un multiple du nombre de processus ($N \bmod P \neq 0$), les lignes restantes sont réparties équitablement afin de garantir un bon équilibrage de charge. Chaque processus calcule ainsi son décalage global (`global_y_start`) et la taille de son sous-domaine local (`local_size_y`).

Le découpage par lignes, plutôt que par colonnes, respecte l'organisation *row-major* de la mémoire en C et favorise la localité spatiale. Chaque processus alloue un domaine local étendu de deux lignes fantômes (halos). À chaque itération, les échanges de bordures sont réalisés via `MPI_Sendrecv`, permettant la synchronisation des halos sans risque d'interblocage.

2.1 Scalabilité Forte

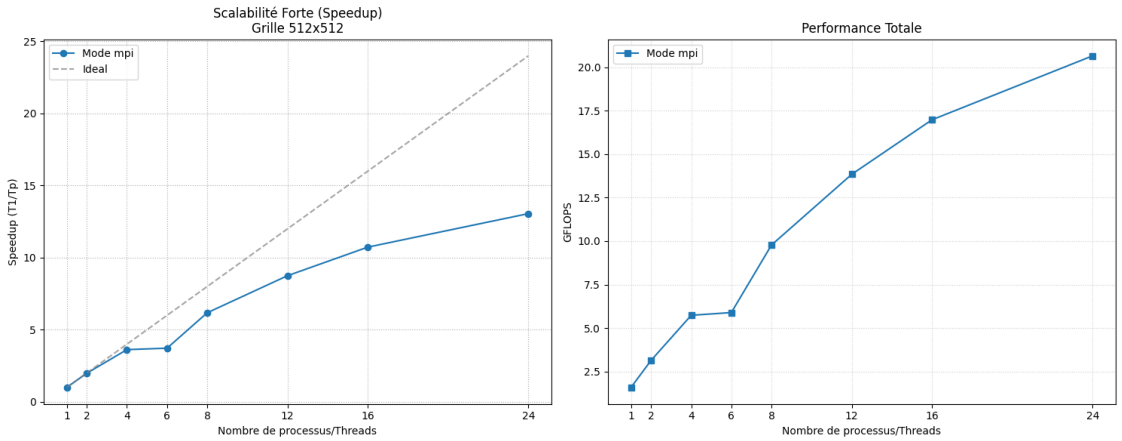

FIGURE 2 – Résultats scalabilité forte MPI

La figure 2 illustre les performances en fort passage à l'échelle. La figure de gauche présente le speedup en fonction du nombre de processus MPI. Entre 1 et 4 processus, on observe un speedup quasi idéal ; en particulier, pour 4 processus, le speedup atteint une valeur proche de 3,5. Cela indique une très bonne parallélisation de l'algorithme, le coût des communications MPI demeurant négligeable devant le temps de calcul dans cette plage.

En revanche, le speedup tend à se saturer entre 6 et 8 processus. Ce comportement s'explique par l'architecture matérielle : le processeur dispose de 6 cœurs physiques, permettant l'exécution efficace de jusqu'à 6 processus MPI sur un même socket. Au-delà, notamment à 8 processus, l'exécution s'étend au second processeur physique, ce qui implique des communications inter-sockets. Les échanges de halos doivent alors transiter par le bus inter-processeur, augmentant la latence, tandis que les processus se répartissent sur deux caches L3 distincts, rendant la synchronisation plus coûteuse.

Enfin, la figure de droite de la figure 2 présente le débit de calcul en GFLOPS en fonction du nombre de processus MPI. On observe une augmentation significative des performances, passant d'environ 1,6 GFLOPS pour un seul cœur à 13,9 GFLOPS pour 12 cœurs.

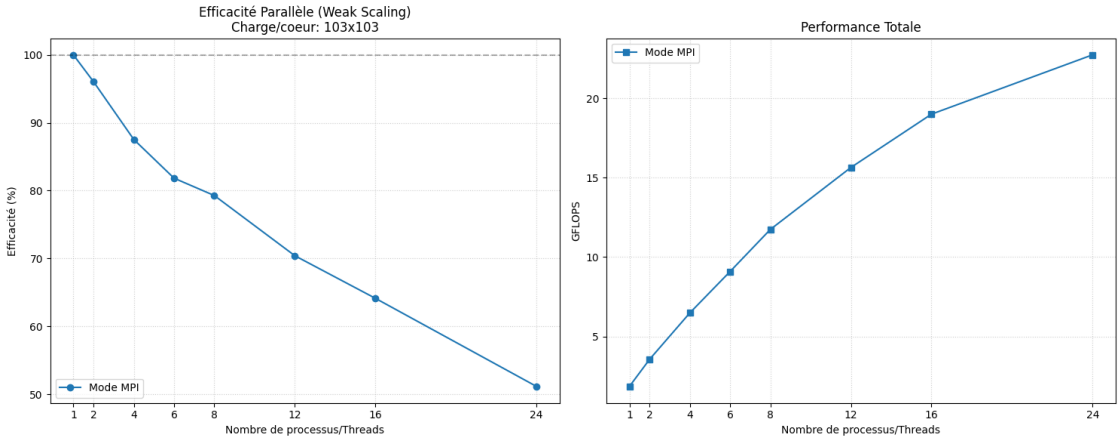

FIGURE 3 – Résultats scalabilité faible MPI

2.2 Scalabilité Faible

La figure 3 illustre les performances en faible passage à l'échelle. La figure de gauche présente l'efficacité en fonction du nombre de processus MPI. On observe une décroissance quasi linéaire de l'efficacité, qui chute pour atteindre environ 45 % à 12 processus.

La figure de droite montre l'évolution du débit de calcul en GFLOPS. Celui-ci sature rapidement : alors qu'un seul cœur atteint environ 1,6 GFLOPS, l'utilisation de 12 cœurs ne permet d'atteindre qu'environ 9 GFLOPS, bien en deçà des 19,2 GFLOPS attendus dans le cas d'une scalabilité faible idéale ($12 \times 1{,}6$).

Cette mauvaise scalabilité en faible passage à l'échelle, caractérisée par une efficacité de 45 % à 12 cœurs, indique que l'application est principalement limitée par la bande passante mémoire.

2.3 Conclusion

On observe que la version MPI pure permet d'atteindre un bon niveau de parallélisme et d'obtenir de solides performances en fort passage à l'échelle, tant que l'exécution reste confinée à un seul socket. En revanche, les résultats en faible passage à l'échelle, ainsi que le comportement au-delà d'un socket, mettent en évidence des limitations structurelles marquées, conduisant à l'atteinte rapide d'un plafond de performance.

3 Version OpenMP

La version OpenMP a été implémentée en utilisant un découpage par blocs de lignes, analogue à la décomposition de domaine employée dans l'implémentation MPI.

Le parallélisme est principalement introduit à l'aide de la directive `#pragma omp parallel for`, qui permet de répartir les itérations de la boucle externe, correspondant aux lignes de la grille, entre les différents threads disponibles.

La gestion du critère d'arrêt repose sur une clause `reduction(&& : convergence)`, qui agrège les indicateurs de convergence locaux calculés par chaque thread en une variable globale à la fin de chaque itération. Cette réduction implique des barrières de synchronisation implicites, obligeant l'ensemble des threads à attendre le plus lent avant de poursuivre l'exécution.

La figure 4 montre que les performances d'OpenMP et de MPI sont très proches pour un faible nombre de processus ou de threads. Toutefois, à partir d'un certain niveau de parallélisme, MPI surpasse OpenMP, aussi bien en termes de speedup que de performance globale. Cette différence s'explique principalement par une meilleure localité mémoire offerte par MPI, chaque processus disposant de son propre espace mémoire, généralement alloué sur le nœud NUMA local. À l'inverse, OpenMP repose sur un modèle de mémoire partagée, ce qui peut entraîner des accès mémoire non locaux lorsque les threads sont répartis sur plusieurs

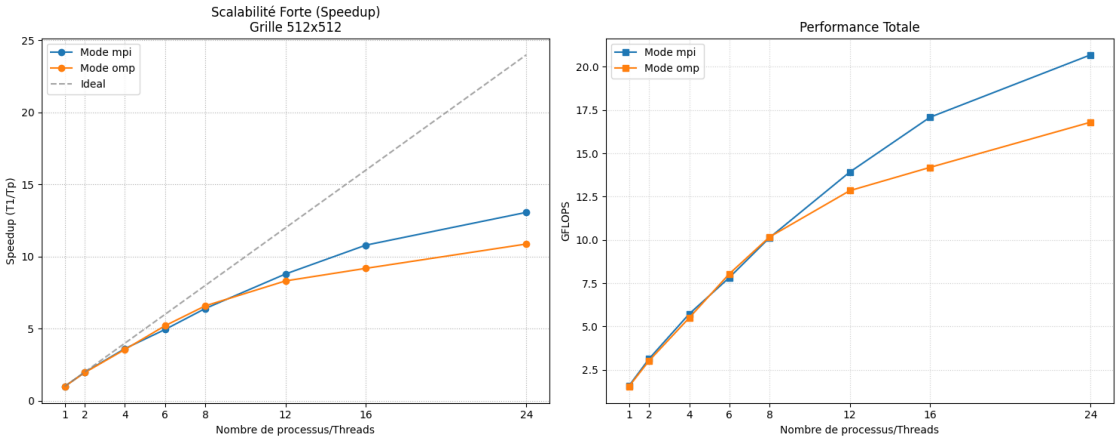

FIGURE 4 – Résultats scalabilité forte OpenMP

sockets. De plus, lorsque la taille du problème par thread diminue, les coûts associés aux synchronisations implicites, aux barrières et au faux partage deviennent proportionnellement plus importants. Ces facteurs limitent la scalabilité d'OpenMP et expliquent l'écart de performance observé en faveur de MPI pour des niveaux de parallélisme élevés.

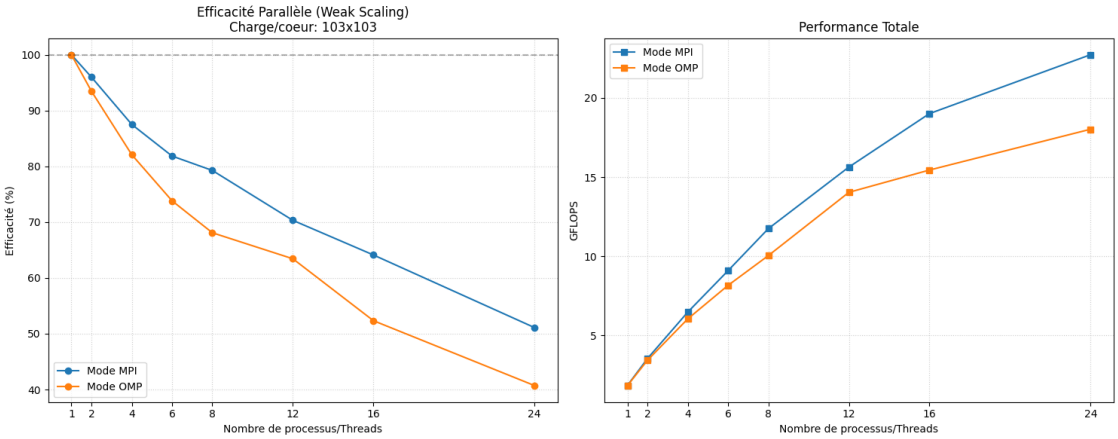

FIGURE 5 – Résultats scalabilité faible OpenMP

La figure 5 met en évidence une efficacité légèrement supérieure de l'implémentation OpenMP par rapport à MPI pour l'ensemble des configurations testées. Cet avantage s'explique par le modèle de mémoire partagée d'OpenMP, qui évite les communications explicites entre processus et limite ainsi le surcoût lié aux échanges de données. Tant que la charge de calcul par thread reste suffisante, ce faible overhead permet à OpenMP d'exploiter efficacement les caches partagés, en particulier le cache de dernier niveau (L3) au sein d'un même socket.

3.1 Conclusion

Ainsi, une implémentation uniquement OpenMP atteint plus rapidement un plafond de performance lorsque le parallélisme augmente, tandis que MPI offre une meilleure scalabilité sur des architectures multi-socket et multi-nœuds. Ces observations suggèrent qu'une approche hybride MPI+OpenMP permettrait de dépasser ce plafond en combinant la scalabilité inter-nœuds de MPI avec l'efficacité intra-nœud d'OpenMP.

4 Version hybride MPI+OpenMP

Compte tenu de l'architecture du nœud `miriel`, composée de deux processeurs physiques organisés en quatre domaines NUMA de six cœurs chacun, une stratégie de parallélisme hybride MPI+OpenMP a été mise en œuvre. L'objectif est d'aligner la hiérarchie logicielle sur la topologie matérielle afin d'optimiser la localité mémoire et l'utilisation des ressources. Pour ce faire, un processus MPI est lancé par domaine NUMA, soit quatre processus au total, chacun exploitant six threads OpenMP afin de saturer les cœurs locaux.

Cette configuration est exécutée en utilisant la syntaxe moderne d'Open MPI, permettant de garantir un placement optimal des processus et des threads :

```
export OMP_NUM_THREADS=6
export OMP_PROC_BIND=true
export OMP_PLACES=cores
mpirun -np 4 --map-by numa:PE=6 --bind-to core ./stencil_mpi_openmp 512
```

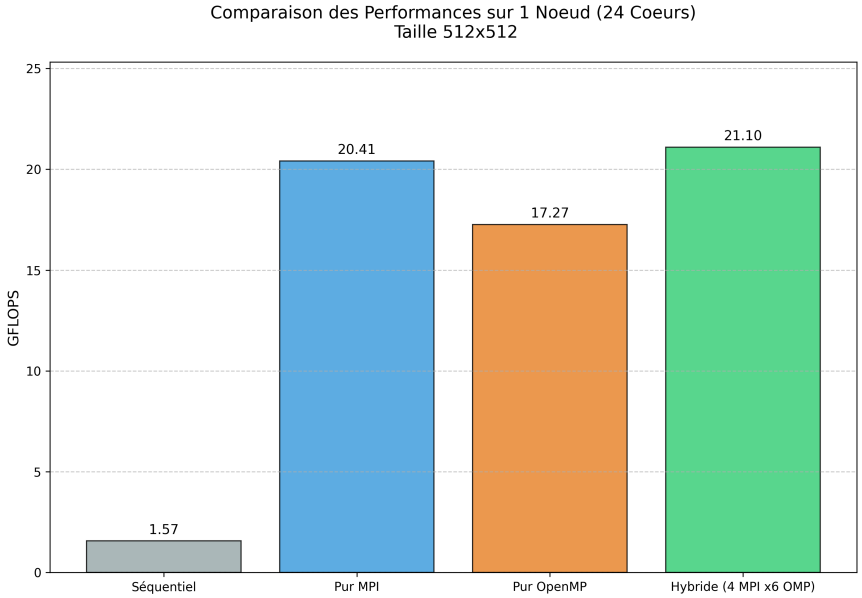

FIGURE 6 – Comparaison des Versions sur 1 Noeud

Comme on peut le voir sur la figure 6, les performances obtenues avec cette approche hybride atteignent 21,10 GFLOPS, dépassant celles des versions purement MPI (20,48 GFLOPS) et purement OpenMP (17,27 GFLOPS). Ce gain de performance s'explique par une réduction de l'overhead de communication et une meilleure exploitation de la hiérarchie mémoire. MPI permet d'isoler les domaines mémoire et de limiter les accès inter-sockets, problématiques dans le cas d'une implémentation OpenMP pure, tandis qu'OpenMP favorise les échanges par mémoire partagée à l'intérieur d'un même domaine NUMA, évitant ainsi des communications MPI inutiles. Cette combinaison permet de tirer parti des avantages respectifs des deux modèles de programmation.

Les figures 7 et 8 présentent les résultats de scalabilité sur un seul nœud. On observe que la version hybride MPI+OpenMP présente une dynamique de croissance intéressante, surpassant par moments la version MPI pure en termes d'efficacité relative sur de faibles nombres de cœurs.

Cependant, l'extension des tests à deux nœuds (figures 9 et 10) montre des résultats plus contrastés. Bien que les performances globales augmentent, la version hybride ne produit pas le gain "drastique" initialement espéré par rapport au mode MPI pur.

Plusieurs facteurs peuvent expliquer ce comportement, notemment la gestion du placement des processus et des threads devient critique. Comme l'illustre la figure 9, le "binding" MPI semble influencer significativement les performances lorsque la machine n'est pas pleinement exploitée.

La stratégie de distribution utilisée pour l'exécution sur deux nœuds est la suivante :

```
export OMP_NUM_THREADS=6
export OMP_PROC_BIND=spread
export OMP_PLACES=cores
mpirun -np 8 --map-by node:PE=6 --bind-to core ./stencil_mpi_openmp 512
```

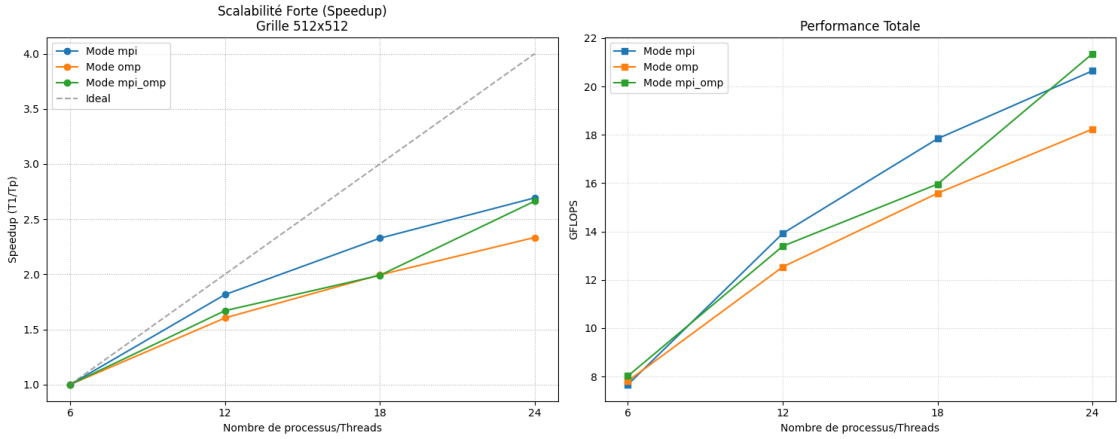

FIGURE 7 – Résultats scalabilité Faible MPI+OMP sur 1 Noeud

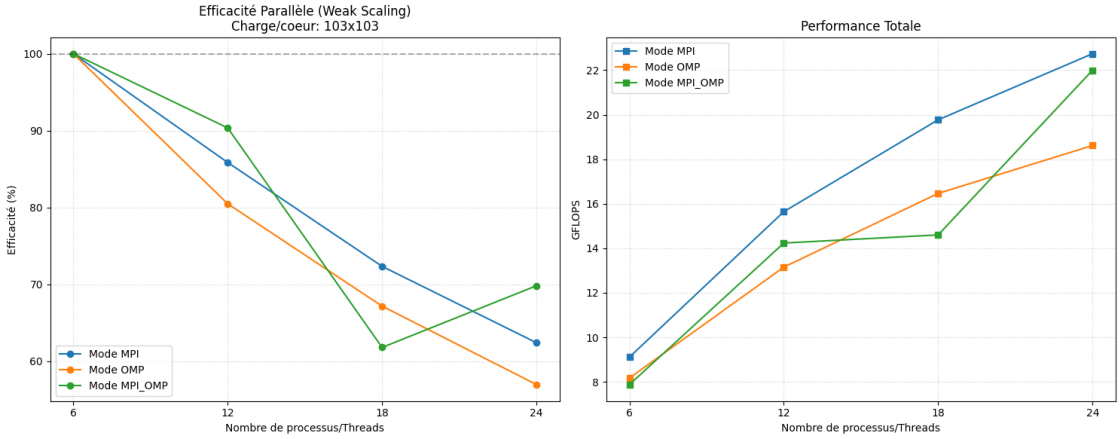

FIGURE 8 – Résultats scalabilité Faible MPI+OMP sur 1 Noeud

Cette configuration (8 processus MPI avec 6 threads chacun) vise à équilibrer la charge sur les deux nœuds. Toutefois, les résultats suggèrent qu'à cette échelle, la communication inter-nœuds et la gestion de la localité mémoire (NUMA) limitent l'efficacité de l'approche hybride, rendant la version MPI pure souvent plus compétitive grâce à une segmentation plus directe des données.

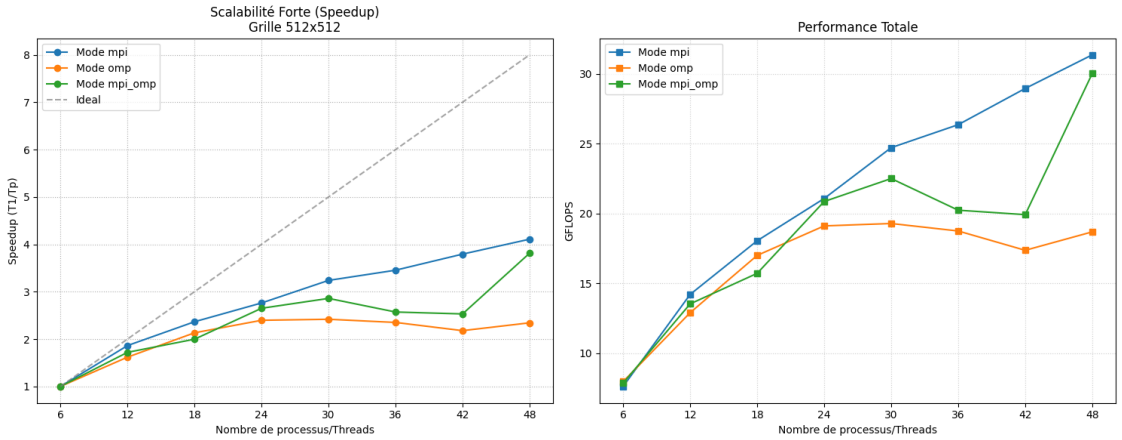

FIGURE 9 – Résultats scalabilité Faible MPI+OMP sur 2 Noeud (OMP reste à 24 threads max)

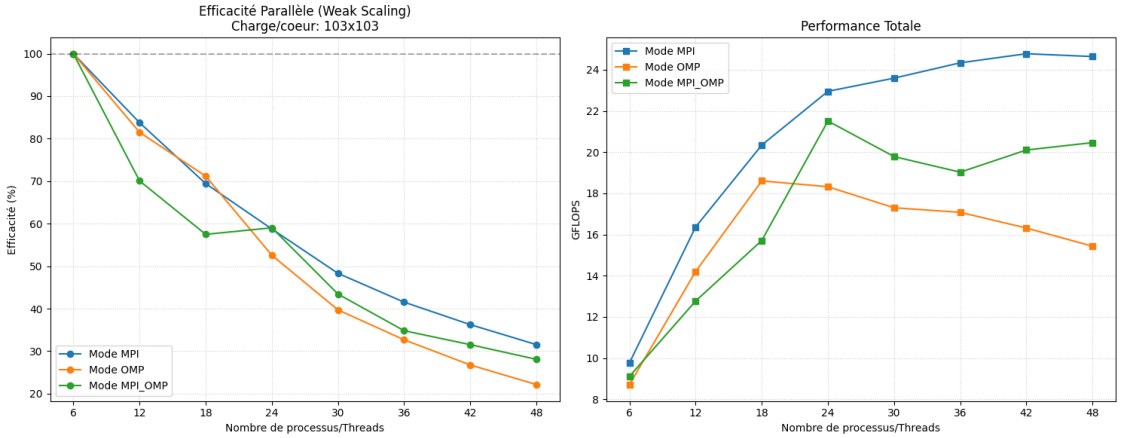

FIGURE 10 – Résultats scalabilité Faible MPI+OMP sur 2 Noeud (OMP reste à 24 threads max)

5 Version MPI+OpenMP recouvrement calcul/communications

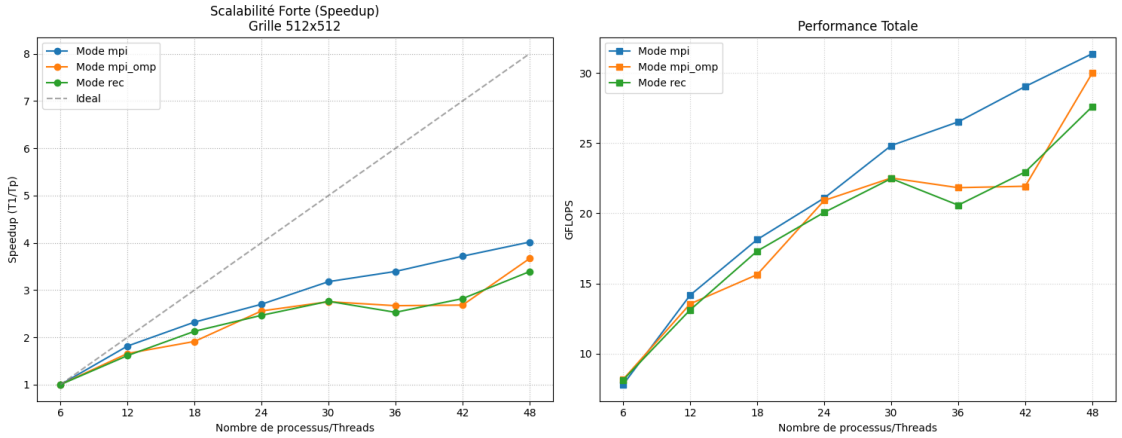

FIGURE 11 – Résultats scalabilité Faible Recouvrement MPI+OMP sur 2 Noeud

Sur le graphique de Performance Totale (figure 11), on observe que le mode REC suit une courbe très proche de la version hybride standard, avec une accélération notable au passage de 42 à 48 cœurs. La version REC montre une meilleure progression finale que la version hybride classique sur la pleine configuration du cluster (48 cœurs).

Le recouvrement semble moins efficace sur les configurations intermédiaires (entre 24 et 42 processus/threads), où l'on observe une stagnation, voire une légère baisse de performance. Cela peut s'expliquer par un volume de calcul interne insuffisant pour masquer totalement le coût des échanges aux frontières.

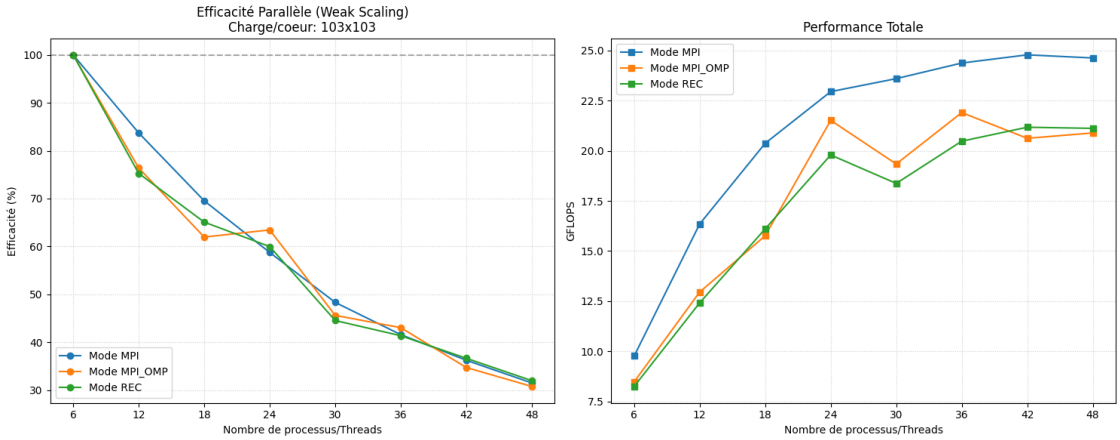

FIGURE 12 – Résultats scalabilité Faible Recouvrement MPI+OMP sur 2 Noeud

En scalabilité faible (figure 12), l'efficacité parallèle du mode recouvrement est quasiment identique à celle du mode MPI + OpenMP. À 48 cœurs, l'efficacité chute aux alentours de 32%, ce qui est cohérent avec les autres modes testés.

L'implémentation du recouvrement (Mode REC) montre un gain de performance à pleine charge (48 cœurs) par rapport à la version hybride simple mais l'avantage reste marginal par rapport au MPI pur.

6 Fin et Axe d'amélioration

L'ensemble des scripts de benchmark en scalabilité forte et faible a été implémenté, les procédures d'utilisation étant détaillées dans le fichier README.md. Bien qu'une décomposition 2D de la matrice ait été envisagée comme axe d'amélioration, les retours d'expérience suggèrent que les gains de performance ne sont pas flagrants en pratique. Concernant la charge de calcul, les tests de scalabilité forte ont été limités à une grille de 512×512 afin de maintenir des temps d'exécution raisonnables, les essais sur des matrices de 1024×1024 ayant montré des durées de l'ordre de la dizaine de minutes par expérience.