HPC:Parallélisation d'un programme pour résoudre le problème à N corps

Michael CHEN, Jérôme COURTOIS

24 janvier 2020

Table des matières

1 Introduction : contexte du problème à N corps.

1.1 Le contexte théorique

N corps dans l'espace s'attirent mutuellement. On modélise cette attraction avec la loi de la gravitation universelle et le principe fondamentale de la dynamique :
On pose :
$\mathbf{F_{Cj \to Ci}}$ la force subit par le corps j sur le corps i
m_i et m_j les masses des corps i et j
P_i et P_j les positions des corps i et j
r_ij la distance entre les corps i et j

$$\mathbf{F_{Cj \to Ci}} = \frac{\mathbf{m_i m_j}}{\mathbf{r_i j^3}}(\mathbf{P_i} - \mathbf{P_j})$$

$$\sum_{\substack{j=1 \\ j \neq i}}^{j=N} \mathbf{F_{C_j \to C_i}} = \mathbf{m_i A_i}$$

Une solution analytique de ce système différentiel existe pour deux ou trois corps. Au delà le problème est complexe, chaotique et dépend fortement des conditions aux limites et on ne connaît pas de solution analytique.
On ne peut donc faire des prédictions qu'en faisant une simulation informatique.

1.2 Modélisation informatique

1.2.1 Modélisation des corps

Un corps Ci est représenté par :
- une masse m_i
- un vecteur position $\mathbf{P_i} = (\mathbf{px_i}, \mathbf{py_i}, \mathbf{pz_i})$
- un vecteur vitesse $\mathbf{V_i} = (\mathbf{vx_i}, \mathbf{vy_i}, \mathbf{vz_i})$
- un vecteur accélération $\mathbf{A_i} = \mathbf{F_i}/\mathbf{m_i} = (\mathbf{fx_i}, \mathbf{fy_i}, \mathbf{fz_i})/\mathbf{m_i}$

On cherche à calculer la position des corps à chaque pas de temps t.

1.2.2 Modélisation algorithmique : schéma LeapFrog

A un temps t :
1- Kick : $\mathbf{V_{i,t+1/2}} = \mathbf{V_{i,t}} + \mathbf{A_{i,t}}.\mathbf{dt}/\mathbf{2}$
2- Drift : $\mathbf{P_{i,t+1}} = \mathbf{P_{i,t}} + \mathbf{V_{i,t+1/2}}$
3- Calcul des $\mathbf{A_{i,t}}$
4- Kick : $\mathbf{V_{i,t+1}} = \mathbf{V_{i,t+1/2}} + \mathbf{A_{i,t}}.\mathbf{dt}/\mathbf{2}$

1.2.3 Le programme en séquentielle : NBODY_direct

Le programme NBODY_direct utilise deux types de stockage des corps pour faire les calculs du problème à N corps : une 'structure de tableaux' ou 'un tableau de structures'. Le choix d'un type de stockage dépend des besoins de l'implémentation avec MPI. Nous verrons que nous avons opté pour la structure de tableaux.

2 La parallélisation en MPI

2.1 Kick and Drift

Le Kick et le Drift utilisent des données indépendantes, en distribuant à chacun des p processus les données de $\frac{N}{p}$ corps, la parallélisation ne pose pas de problème particulier.

2.2 Découpage en anneau

Il reste à traiter le calcul des forces $F_i = \sum_{\substack{i=1 \\ i \neq j}}^{i=N} \mathbf{F_{C_j \to C_i}}$ appliquées aux corps.

Les $\mathbf{F_{C_j \to C_i}}$ dépendent de positions et masses des corps i et j. Chaque processus ne possède que les données de $\frac{N}{p}$ corps mais nécessite pour les calculs des forces, les données de position et de masse de chacun des autres corps. Pour en faire la somme et calculer F_i, il faut donc transmettre les données de positions et de masses au processus correspondant. Comme les masses sont constantes au cours de l'ensemble des calculs, on choisit de toutes les transmettre (une seule fois) à l'ensemble des processus au début de l'exécution. Pour une meilleure extensibilité mémoire (mais demandant plus de communications), une variante possible aurait été de les transmettre à chaque communication en même temps et de la même façon que les positions.

C'est pour faciliter ces communications que l'on a choisi la 'structure de tableaux' qui nous permet de communiquer un ensemble de positions et de masses, et non la totalité des données d'un corps.

En raison du principe des interactions réciproques, la matrice C des interactions entre les corps est antisymétrique ($C_{i\to} = -Cj \to$) seule la moitié des interactions entre corps à besoin d'être calculée.

Il s'agit de décomposer le calcul des éléments de cette matrice C et de définir les étapes de transmissions des données. On s'inspirera d'une structure en anneau (Figure 1) :

- Chaque processus reçoit $\frac{N}{p}$ corps.
- Chaque processus calcule les sommes de forces pour le système à $\frac{N}{p}$ corps dont il possède les données et les stocke dans un tableau F_i.
- Chaque corps transmet les données de positions P_i dans un tableau P_j du processus suivant dans l'anneau. P_i représente donc les positions des corps locaux et P_j des corps visiteurs.
- **Répéter** $\frac{N}{2} - 1$ fois (moitié de la matrice) :
  - Chaque processus calcule les sommes de forces dues aux interactions entre ses corps locaux et les corps reçus.
  - Chaque processus mets à jour la somme des forces pour les corps locaux que l'on a noté F_i
  - Chaque processus transmet les positions reçus ainsi que les sommes partielles des forces correspondantes au processus suivant noté F_j (somme sur les colonnes de la matrice C des forces d'interaction).

. Si p est pair, il faut une étape supplémentaire pour compléter le calcul. (voir Figure 2)

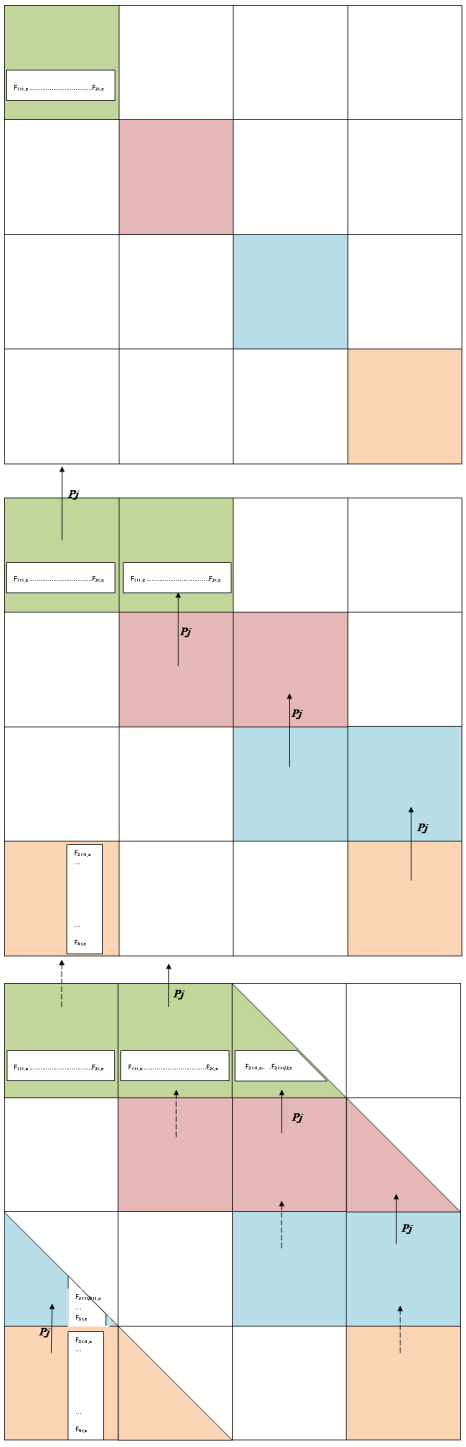

FIGURE 1 – Le découpage en étapes de calcul de la matrice des interactions. On fait la somme sur la ligne blanche horizontale pour obtenir un F_i local et la verticale correspond à un F_j visiteur.

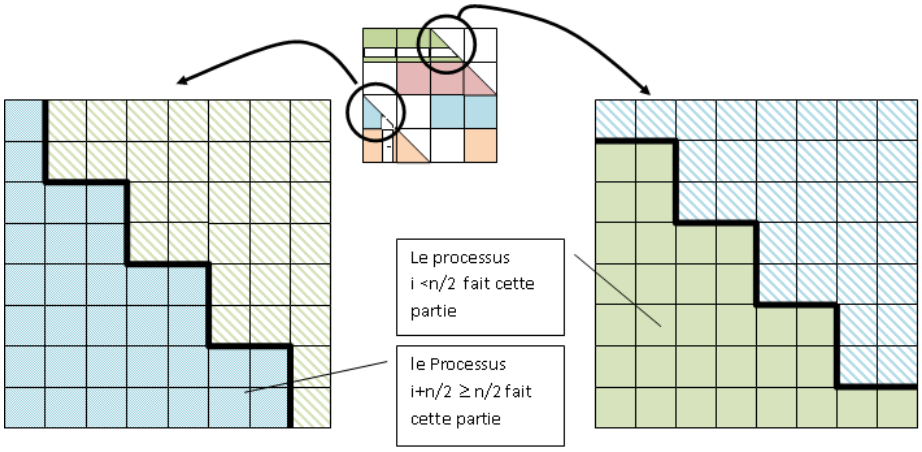

FIGURE 2 – Le découpage de la dernière itération du calcul dans le cas p pair

On peut ici distinguer deux types de blocs : en vertu du principe des interactions réciproques, la première et la dernière étape (dans le cas p pair) calculent des demi-blocs d'interactions tandis que les étapes intermédiaires calculent des blocs complets.

2.3 Equilibrage de charge

Avec l'implémentation décrite ci-dessus, chaque corps demande la même quantité de calcul. Le découpage statique en $\frac{N}{p}$ corps par processus donne une répartition de charge équilibrée.

2.4 Recouvrement par le calcul

En utilisant des buffers d'envois et de réceptions supplémentaires pour les forces et les positions, on est capable de recouvrir entièrement nos communications par le calcul. En effet, il n'y a pas de dépendance entre les calculs d'une itération de calcul à la suivante. Cependant, le transfert des sommes partielles F_j des forces en même temps que les positions P_j d'un processus à l'autre peut poser problème. Comme le processus met à jour les forces F_j pendant le calcul, il faut qu'il les ai reçu du processus précédent avant cette mise-à-jour, ce qui implique de créer un nouveau buffer pour les stocker entre temps. Nous avons opté pour une autre solution, qui consiste à renvoyer directement ces F_j au processus dont les P_j visiteurs sont originaires plutôt que de les faire circuler dans l'anneau. (Figure 3)

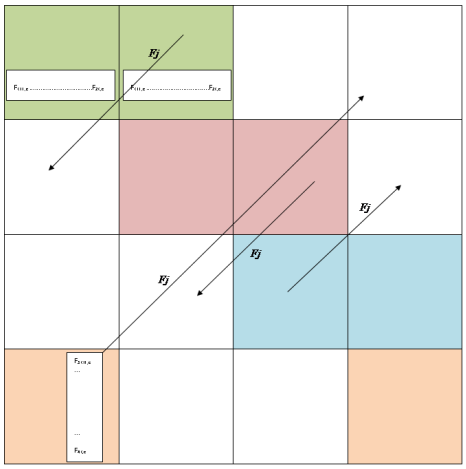

FIGURE 3 – Transfert des F_j entre processus : Chaque ligne correspondant à un processus, on renvoie les F_j calculés directement au processus concerné qui met alors à jour ses F_i.

3 Parallélisation multithread en OMP

3.1 Niveau d'hybridation MPI+OMP

Dans notre implémentation, on s'est restreint au mode MPI_THREAD_FUNNELED : seul le thread principal effectue des appels MPI. Cela est possible car aucune communication n'a lieu dans les zones OMP parallèles.

3.2 Parallélisation des boucles simples

Les étapes de Kick et Drift sont des boucles for qui se parallélisent et s'équilibrent sans difficultés avec une répartition de charge statique. On a fait de même pour la plupart des boucles for dans le code où les données sont indépendantes.

3.3 Parallélisation du calcul des interactions

3.3.1 Principe

Le calcul des interactions en revanche est une double boucle imbriquée pour lesquelles il faut choisir la boucle à paralléliser et gérer les problèmes d'accès concurrents conflictuels. La double boucle (simplifiée) est la suivante :

. Pour tous les corps locaux i
  . Pour tous les corps visiteurs j
    — Calculer $F_{C_j \to C_i}$
    — Mettre à jour $F_i += F_{C_j \to C_i}$ et $F_j -= F_{C_j \to C_i}$

En choisissant de paralléliser la boucle externe, la mise-à-jour des F_i ne pose pas de problème, chaque thread accède à sa portion du tableau. En revanche, la mise à jour des F_j se fait de façon concurrente.
Afin de traiter ce problème, dans un premier temps, nous avons utiliser des directive omp atomic, ce qui a donner des performances extrêmement mauvaises.

Pour éviter d'y recourir nous avons assigner un tableau F_j^k à chaque thread k destinés à recevoir les sommes partielles des F_j calculés par chacun des threads. Il suffit alors de sommer sur k pour retrouver F_j.

3.3.2 Equilibrage de charge

Il faut distinguer les deux cas selon si on calcul l'ensemble du bloc d'interactions ou seulement la moitié. Dans le premier cas, un équilibrage de charge statique convient très bien. Dans l'autre, chaque itération de la boucle externe n'a pas le même nombre d'interactions à calculer. Notre implémentation répartit les charges selon la valeur de la variable d'environnement OMP_SCHEDULE.

4 Vectorisation SIMD

4.1 Vectorisation des boucles simples

Comme pour les boucles *for* trivialement parallélisées, il suffit de rajouter la directive omp simd pour vectoriser la plupart des boucles. Cependant, on s'aperçoit en test que les performances obtenues sont plus mauvaise que sans vectorisation. L'explication (donnée en TP) pourrait être que l'option de compilation -O3 (qui inclue -ftree-loop-vectorize) vectorise automatiquement ces boucles simples en AVX alors que la directive omp simd le fait en SSE.

4.2 Vectorisation du calcul des interactions

A l'aide d'intrinsèques AVX, nous avons seulement vectorisé une partie du calcul des interactions. Dans la structure en anneau, le premier bloc (et dans le cas p pair le dernier bloc) de calculs ne sont pas vectorisés. Toutes les interactions du bloc n'étant pas calculé, leur vectorisation était plus difficile et sans garanti de résultats. Vectoriser le calcul de tous les autres blocs était a la fois plus simple et plus significatif en termes de performances dès que $p > 2$: ils représentent la majorité $(1 - \frac{2}{p})$ des calculs. Les machines utilisés ne disposant pas d'AVX2, nous n'avons pas non plus exploité les FMA vectoriels.

5 Contrôle

Afin de tester la validité de nos implémentations, une première vérification à été d'observer si la somme des forces calculés étaient très proches de 0. Cette vérification peut ne pas suffire car elle ne tient pas compte du calcul des positions : on peut avoir des sommes nulles alors que les positions sont fausses. Nous avons comparer les dernières positions de quelques corps renvoyé par nos implémentations avec les résultats données par l'algorithme séquentielle.

6 Quelques remarques

— Au temps 0 le calcul des forces est toujours plus long qu'aux temps suivants qui restent homogènes jusqu'à la fin du calcul.
— Le processus 0 n'est pas toujours le processus le plus long et il n'y a pas toujours un temps de calcul équivalent entre les processus.

— Le fait que l'hybridation avec MPI+OMP ne soit pas bonne ne nous a pas encouragé à tester l'hyperthreading sur OPENMP. On s'est donc limité à 4 threads.
— Quand nous faisons nos calculs, la somme des forces sont de l'ordre de 10^{-7} ou 10^{-8} ce qui correspond aux résultats attendus.
— La différence entre les positions calculés par le séquentiel et nos implantations parallèles sont de l'ordre de 10^{-4} pour plus de 10 pas de temps. Il n'y a pas de différence pour les premiers pas de temps.

7 Résultats

Les temps de calculs ont été mesurés sur les machines de la salle 303.

7.1 Efficacités de l'implémentation MPI

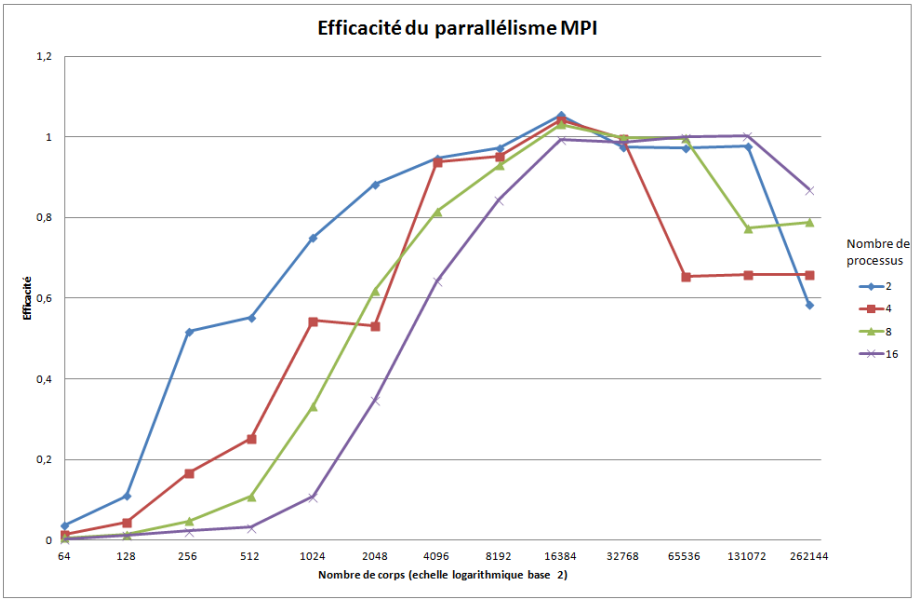

FIGURE 4 – Efficacités de l'implémentation avec MPI. Chaque courbe représente un nombre de processus MPI.

La figure 4 présente les différentes efficacités obtenues.
Pour un petit nombre de corps, elle est mauvaise en raison du ratio (nombre de calcul) / (quantité de communication) qui est alors très faible. Plus le nombre de corps augmente, plus l'efficacité s'améliore jusqu'à 16384 corps.
A 16384 corps, on observe une anomalie : les efficacités sont supérieurs à 1. On observe par ailleurs une chute de l'efficacité à 65536 pour 4 processus, à 131072 pour 8 processus et à 262144 pour 16 processus, ce qui nous amène à supposer que lorsque l'on dépasse 16384 corps par processus, il y aurait une chute de performances, qui pourrait expliquer la valeur d'efficacité supérieur à 1. Il est possible que la mémoire soit mieux utilisée quand on est en dessous de 16384

corps par processus.

On note globalement une bonne efficacité du parallélisme MPI, et en particulier pour un grand nombre de corps. Il aurait très intéressant de vérifier si la chute évoquée ci dessus continue pour plus de 260 000 corps ou si l'efficacité se stabilise.

7.2 Comparaison versions MPI et hybride MPI + OMP

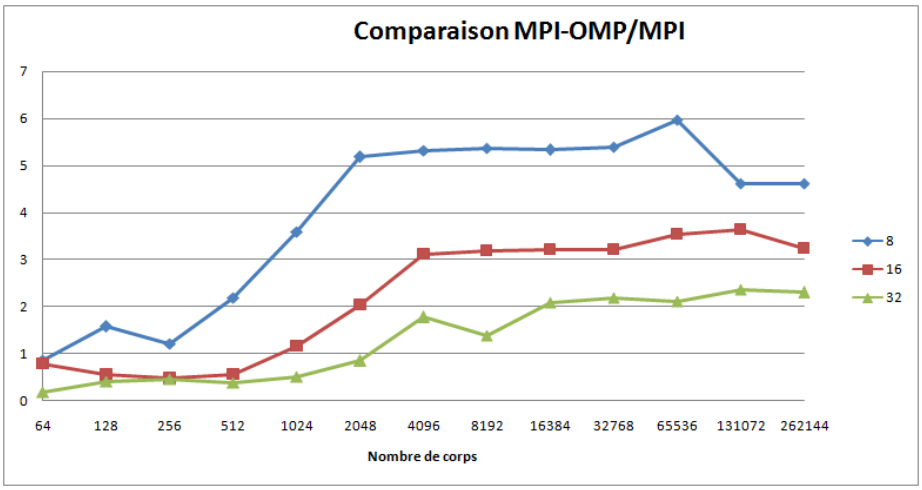

FIGURE 5 – Rapport entre les temps obtenus avec l'implémentation hybride avec MPI+OMP par rapport au MPI pure, sur le même nombre de noeuds et de coeurs.

Tous les calculs ont été fait avec 4 threads. On observe que pour des ressources identiques, l'implémentation avec MPI pure est meilleure que l'implémentation hybride avec MPI+OMP.
Dans cette comparaison, la variable OMP_SCHEDULE n'ayant pas été définie, $\frac{2}{p}$ des calculs d'interactions dans la version hybride n'ont pas été reparties de façon optimale. Ceci peut expliquer les écarts entre les trois courbes. Plus le nombre de processus p est grand, plus cette proportion diminue.
Toutefois, au vu des écarts, MPI pure étant au moins 2 fois plus rapide, on extrapole aisément que même en optimisant cette répartition, la version MPI pure resterait meilleur. (Ce qu'il faudrait confirmer expérimentalement).
Le surcoût d'une implémentation avec MPI venant essentiellement des communications qui, dans notre cas, ont été recouvertes, cette constatation n'est pas forcément absurde, la version hybride comprenant dans ses surcoûts la mise en place du multithreading ainsi que le regroupement (somme) des forces calculés par chaque thread.

7.3 Temps de calcul et option de compilation -fast-math

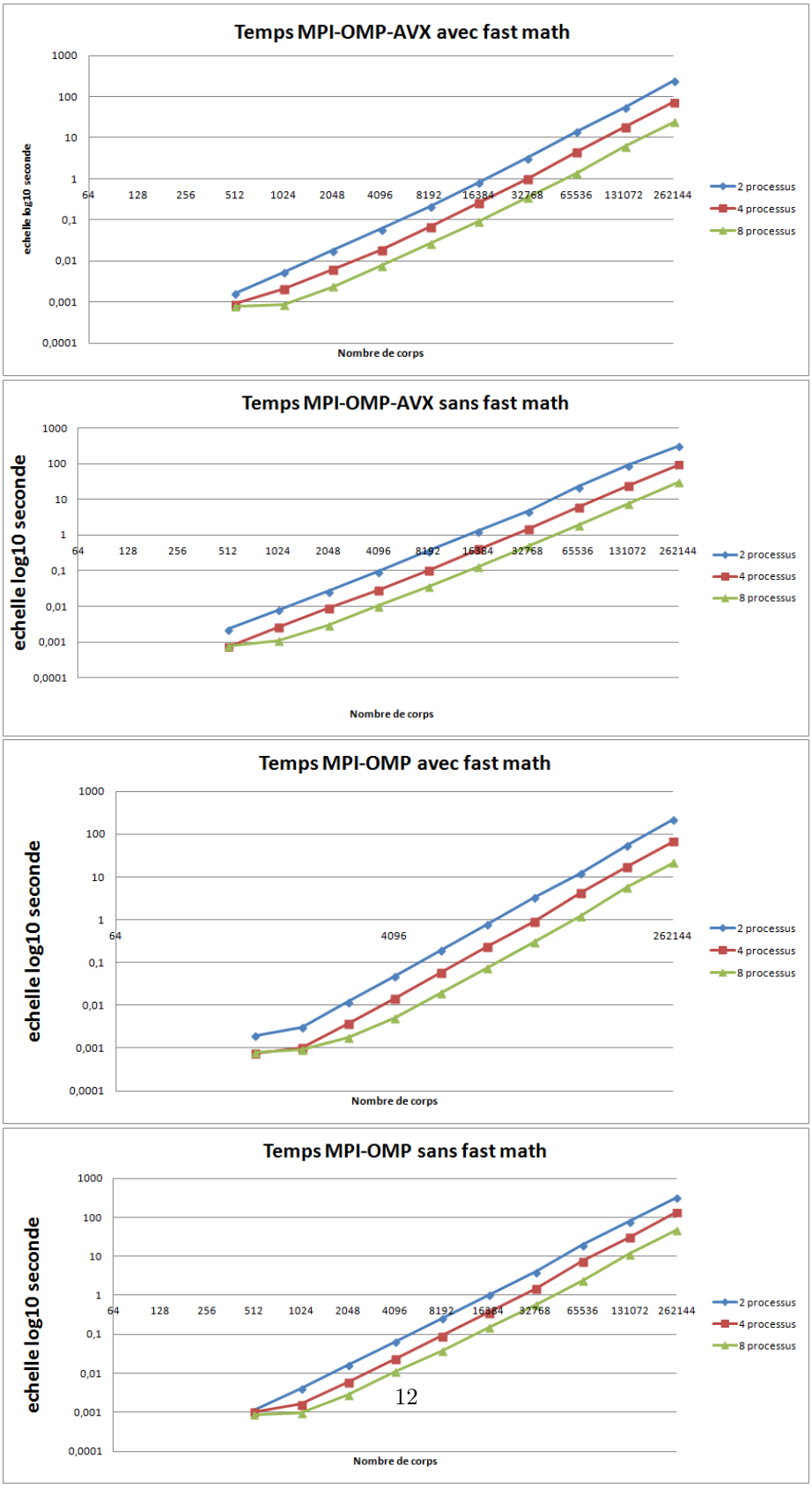

FIGURE 6 – Temps obtenus pour les implémentations hybrides (4 thread) sans ou avec -fast-math, avec ou sans vectorisation AVX. Echelle log2 pour les abscisses et log10 pour les ordonnées

Il est claire grâce à ces graphiques que les temps de calcul suivent une loi en $10^{a log_2(N)+b} = \alpha N^{\beta}$ à partir d'un certain nombre N_0 de corps qui varie entre 500 corps avec OMP et 1000 sans OMP (les cas sans OMP n'ont pas été représenté pour ne pas alourdir le rapport). Le paramètre b dépend du nombre de processus et des options de parallélisation choisies. Mais le coefficient a ne dépend pas du nombre de processus, en tout cas dans les trois cas analysés ici, il ne dépend que des options que l'on a choisies. On remarque que l'option de compilation -fast-math diminue légèrement le temps de calculs. Cette influence se voit mieux sur une échelle linéaire en ordonnée. On peut se poser la question de son influence par rapport aux options de parallélisation choisies.

7.4 Comparaison versions MPI+OMP et MPI+OMP+AVX

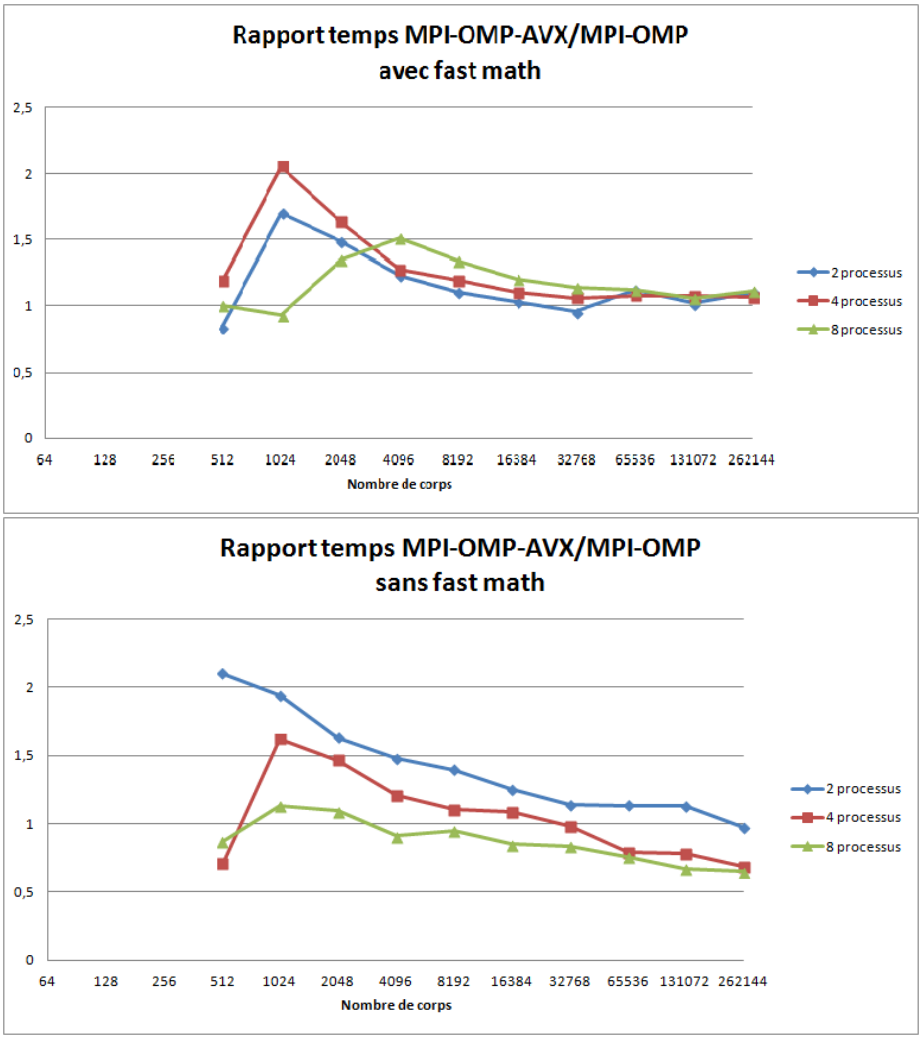

FIGURE 7 – Rapport entre les temps obtenus pour les implémentations hybrides (4 threads) avec ou sans vectorisation AVX.

On s'aperperçoit que l'option de compilation -fast-math a une influence sur les performances Etonnamment, la vectorisation fait chuter les performances. Nous avons découvert laborieusement que ce phénomène est dû à l'option -fast-math du complilateur. En effet, sans -fast-math, la version vectorisée est légèrement plus rapide que la version scalaire (Figure 6).

En résumé, pour l'implémentation hybride, -fast-math scalaire est plus rapide que -fast-math vectorielle, elle-même plus rapide que la version AVX sans fast-math. Et la version sans fast-math scalaire étant sans surprise la plus lente. On suppose que l'option -fast-math gère mal la compatibilité avec la combinaison MPI + OMP + AVX.(Figure 8)

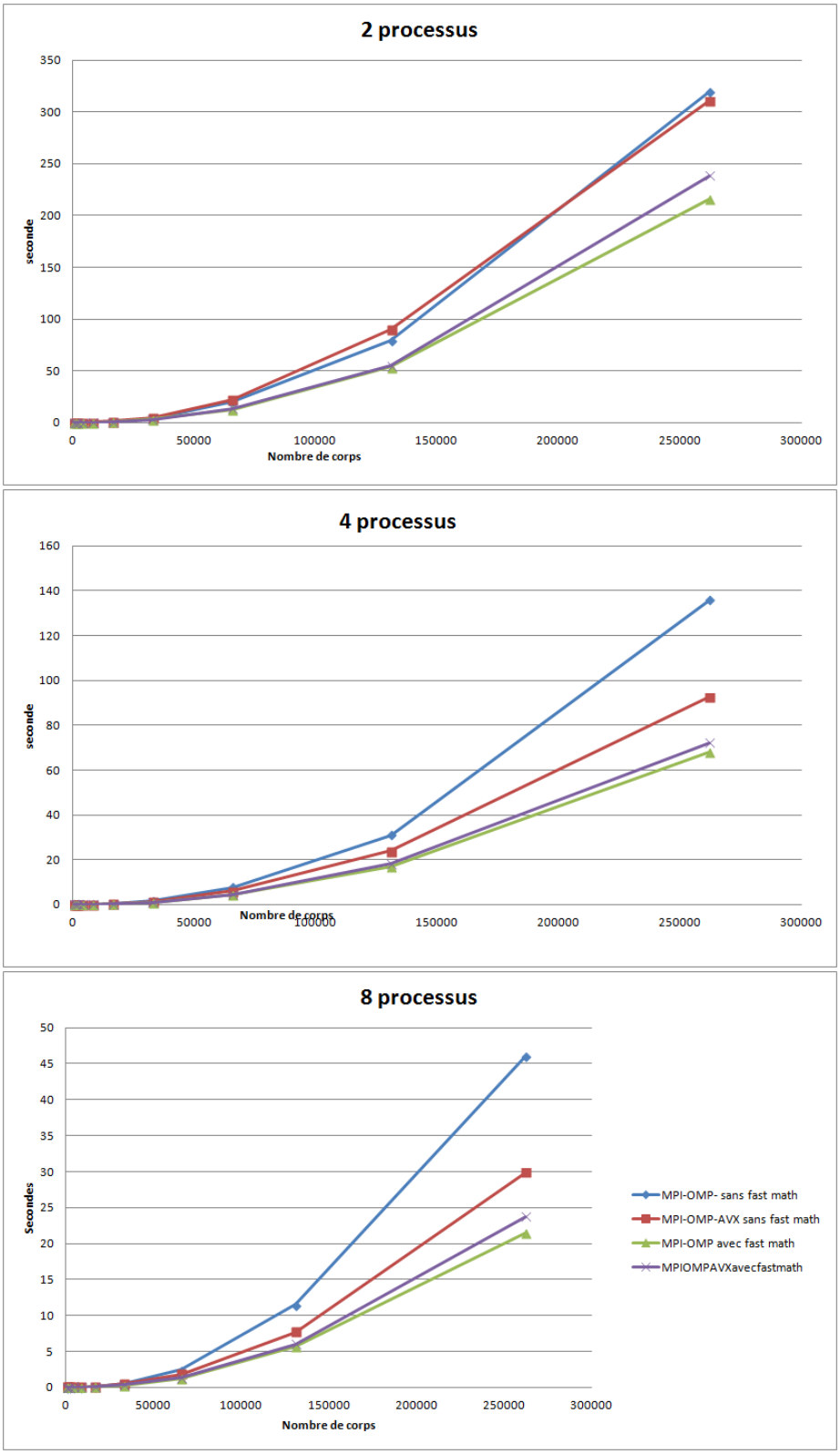

FIGURE 8 – Temps de calcul pour les différentes versions

7.5 Comparaison MPI et MPI-AVX

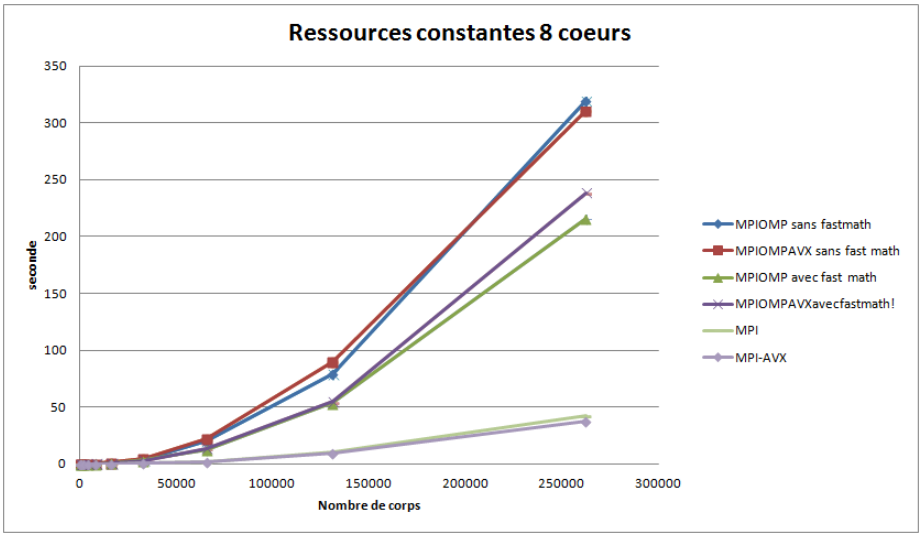

FIGURE 9 – Comparaison des temps entre toutes les versions avec des ressources constantes.

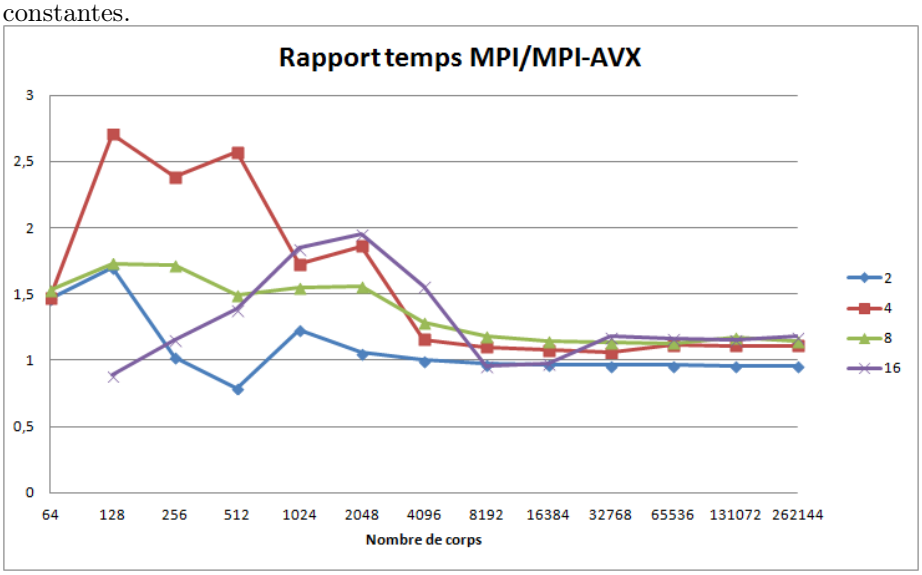

FIGURE 10 – Comparaison entre la version MPI-AVX et la version MPI pure

La version MPI pure vectorisée est notre meilleure version(Figure 9). Toutefois, les gains par rapport a la version MPI scalaire ne sont pas à la hauteur de l'attente, de x8 espéré idéalement.(Figure 10

8 Conclusion

On s'aperçoit que cumuler les types d'optimisation n'améliore pas forcément performances comme on pourrait s'y attendre. Et même quand il y a amélioration, celle-ci n'est pas forcément importante. Ceci est dû aux interactions entre les options d'optimisation et les méthodes implémentées.
On aurait pu tester chaque amélioration indépendamment les unes des autres mais dans l'optique d'avoir les meilleurs performances possibles, nous avons choisi une approche incrémentale des ajouts d'optimisations. En outre, la répartition des charges dans nos tests OMP n'est pas suffisante, il aurait fallu faire plus de tests pour trouver la meilleur taille de chunk.
Nous n'avons pas non plus tester les nombre de corps qui ne sont pas des puissances de 2. Nos implémentations ne peuvent le gérer. Cependant, notre implémentation avec MPI seule gère un nombre de corps qui est multiple d'un nombre de processeur fixé (que ce nombre soit pair ou impair).