SCIENCES SORBONNE UNIVERSITÉ

UPMC SORBONNE UNIVERSITÉS

Projet HPC

Rendu « photo-réaliste » avec illumination globale

BAMBA Ibrahim
Nitcheu Paul

2018 - 2019

INTRODUCTION

La programmation séquentielle, il s'agit sans doute de la méthode la plus naturelle, de la base de la programmation. Lorsque l'ordinateur a fini de traiter une instruction, il exécute la suivante [1], puis une autre… jusqu'à la fin du code. La programmation parallèle permet d'exécuter plusieurs instructions à la fois et ainsi d'obtenir parfois un gain de temps considérable. Le début des années 60 a été marqué par l'apparition des multiprocesseurs. Dès lors, ces architectures parallèles ont été utilisées pour les applications nécessitant des besoins de calcul que les monoprocesseurs étaient incapables de fournir [2]. Toutefois, la loi de Moore dit : <<le nombre de transistors double tous les deux ans>>. Ainsi, à partir des années 80, on assistait à une progression exponentielle des performances des microprocesseurs, utilisés dans les PC et les serveurs multiprocesseurs. Ce qui limita l'utilisation massive du parallélisme aux très grandes applications de simulation numérique avec les architectures massivement parallèles [2]. Mais aux débuts des années 2000, à cause des contraintes énergétiques liées à la dissipation de la chaleur des processeurs, la limitation de la chaleur a complètement changé la donne. Ainsi, en 2016, les processeurs multi coeurs sont présents dans toutes les architectures matérielles. Compte tenu de la baisse de la puissance de calcul par coeur CPU au profit du nombre de coeur et des besoins en calcul toujours croissants de la recherche et de l'industrie, le besoin de parallélisation des codes se renforce.

Le parallélisme consiste à faire travailler simultanément plusieurs processeurs pour résoudre un même problème afin de diminuer le temps de calcul et d'augmenter la taille de problème traité. Parmi les moyens informatiques à notre disposition, MPI, OpenMP et la vectorisation. Le projet : "Parallélisation d'une image de synthèse", est une application parfaite pour le calcul parallèle. Une scène fictive est décrite; un appareil photo virtuel est positionné et orienté. Le code tente de reproduire la photo de la scène fictive que prendrait l'appareil photo virtuel si la scène était vraiment présente dans le monde matériel en utilisant les techniques du ray-tracing et du path-tracing. Toutefois, pour obtenir un rendu de bonne qualité, il faut beaucoup d'échantillons par sous-pixel (ex : 5000), cela devient donc très long en séquentiel [3]. Le but de ce projet est donc de paralléliser un programme séquentiel qui nous a été fourni qui résout ce problème. Dans la suite, nous présenterons dans un premiers temps nos algorithmes en MPI, MPI+OpenMP et MPI+OpenMP+SIMD utilisés et nos choix d'implémentations, dans un second temps nous montrerons les résultats que nous avons obtenus, leur analyse et la comparaison des différentes versions.

I. PRÉSENTATION DES ALGORITHMES ET CHOIX D'IMPLÉMENTATION

Dans toute la suite, sauf mention contraire, les tests ont été effectués par défaut avec une image moyenne dont les caractéristiques sont : w=2000, h=1500 et samples=500.

1. Parallélisation MPI pure

MPI (Message Parsing Interface), est une norme pour le passage de messages entre ordinateurs distants ou dans un ordinateur multiprocesseur [4]. Il permet de répartir une même tâche sur plusieurs processeurs. Dans notre code séquentiel, la boucle principale parcourt l'image de haut en large. Cette

boucle effectue un sur-échantillonnage 2x2 : chaque pixel de l'image est découpé en quatre sous-pixels, dont la couleur est calculée séparément [4]. Le calcul sur chaque pixel de l'image est donc indépendant. La boucle principal peut donc être parallélisée. Ainsi, pour paralléliser notre code en MPI pure avec la bibliothèque OpemMPI, nous avons décidé de traiter cette boucle de façon parallèle.
On procède par un découpage 1D statique de l'image c'est-à-dire un découpage sur la hauteur de l'image. On suppose alors que le nombre de processus divise la hauteur de l'image. Nous avons donc procédé en 3 étapes avec 3 algorithmes différents :

<u>Etape 1</u> : Répartition statique des charges

Soit h la hauteur de l'image et np le nombre de processus. Avant le début des calculs, chaque processus calcule la portion de l'image sur laquelle il doit travailler. Soit h' la hauteur de la portion de l'image sur laquelle travailler : h' = h/np. Le processus de rang r travaille de r * h' à h' + r * h'. Chaque processus travaille donc sur 3*w*h' pixels. Les processus écrivent leur résultat dans une image locale temporaire. Après les calculs, les données sont regroupées dans l'image totale par le processus root avec MPI_Gather.
Bien qu'on constate un gain de temps par rapport à l'algorithme séquentiel, l'inconvénient avec cet algorithme est que le speed-up est sous-linéaire. Pour expliquer cela, nous avons affiché le temps de calcul (à la sortie de la boucle principale, avant le regroupement des données) de chaque processus. On constate que certains processus finissent leurs calculs avant d'autres (avec parfois une différence importante des temps de calcul). Le processus root est le plus lent. Cela implique que **certains processus sont au repos pendant que d'autres ont encore énormément de travail. La répartition des charges n'est donc pas équilibrée. D'où la nécessité d'un équilibrage de charge dynamique.**

<u>Etape 2</u> : Équilibrage de charge dynamique

Nous avons mis en place un équilibrage de charge dynamique de type auto-régulé en mode multi-processus avec MPI pour résoudre ce problème.
Principe *:* Avant les calculs, nous procédons à une répartition de charge statique avec un découpage 1D par bloc cyclique : chaque processus a une liste de bloc de travail. Le choix de la taille des blocs de travail est laissé à l'utilisateur. Mais par défaut, nous avons choisi 5 lignes par bloc. Soit *nlines* le nombre de ligne par bloc. Chaque processus a donc partie entière inférieure de *h/(np*nlines)* blocs de travail. Si np*nlines ne divise pas h, alors les processus de rang inférieur à *(h/nlines) mod np* ont un bloc en plus. Les processus travaillent alors sur leurs blocs de travail. À chaque bloc traité, le processus sauvegarde l'image temporaire correspondant dans une liste de résultats.
Nos choix : i. Nous avons implémenté une structure de liste simplement chaînée (dans les fichiers *list_res.h* et *list_res.c*) qui contient les portions d'images résultats calculés par le processus.
ii. Lorsqu'un processus termine ses blocs de travail, il fait une requête de demande de blocs aux autres processus ⇒ les processus doivent communiquer même durant le calcul. Pour cela, nous avons décidé que les processus testent leur buffer de communication avec MPI_Iprobe pour savoir s'ils ont reçu une requête après chaque tour de parcours de la boucle sur la largeur (la boucle for j).
iii. Nous utilisons 3 tags : TAG_REQ pour les requêtes de demandes de blocs, TAG_DATA pour les envois de données et TAG_END pour notifier les autres processus de la fin de tous les blocs.
Algorithme :

1. *nblocs=h/(np*nlines) + (rank < (h/nlines) mod np) ? 1 : 0, ind=0*
2. *Allocation de la listes des blocs (tableau d'entier blocs de tailles nblocs)*
3. *Initialisation de la liste (Pour i de 0 à nblocs-1, blocs[i] = NP * i + rank)*
4. *root alloue l'image intégrale*
5. *Initialisation d'une liste chaînée pour les images calculées localement*
6. *Calcul des pixels (exécution boucles for de 0 à nlines)*
7. *Après chaque tour de la boucle sur la largeur (for j de 0 à w), vérifier si on a reçu une requête (demande de bloc de travail) d'un autre processus avec MPI_Iprobe*
8. *Si oui, lui envoyer la moitié des blocs qui nous reste (les numéros de blocs concernés).*
9. *Ajouter l'image résultat temporaire dans la liste chaînée*
10. *Invalider les blocs déjà traités dans la liste des blocs de travail (blocs[ind]=-1)*
11. *Fin des calculs*
12. *Vérifier si on a reçu un TAG_END. Si oui ⇒ fin des blocs, aller dans gather*
13. *Sinon, Vérifier si on a reçu une requête d'un autre process. Si oui, lui répondre négativement.*
14. *Sinon, envoyer une requête de demande de bloc au processus de rang rank+1 et attendre la réponse. Si on reçoit une réponse négative, envoyer la requête au processus suivant et ainsi de suite jusqu'à ce que l'un des processus nous envoie des blocs, ou on aie parcouru tous les processus. Dans ce cas, envoyer un TAG_END à tous les processus.*
15. *root reçoit les images temporaires de chaque processus. Les autres envoient à root les images calculées dans la liste.*
16. *Sauvegarde de l'image par le root*

Cet algorithme marche bien et nous donne des résultats beaucoup meilleurs que le séquentiel et le statique (cf partie II). Toutefois, on constate toujours que le speed-up est sous-linéaire et l'efficacité pas énorme (*cf slide "Parallélisation d'une image de synthèse.pdf"*). Cela est dû aux temps de communication. Nos processus mettent énormément de temps dans la communication après les calculs.

<u>Etape 3</u> : Équilibrage de charge dynamique avec anneau logique de processus

Après la soutenance sur la première partie et certaines consignes laissées par l'encadrant, nous avons choisi le mécanisme de l'anneau logique de processus pour nos communications. Les calculs et le mécanisme est pareil que précédemment. La différence intervient dans la manière dont les processus communiquent :

1. *Quand un processus a fini ses tâches, il envoie un message de demande de travail avec le tag TAG_REQ à travers l'anneau logique. Si un des processus récepteur a une file non vide, il renvoie directement au proc demandeur les numéros de la moitié des tâches qui lui reste avec le tag TAG_DATA.*
2. *Si le message de demande revient au proc demandeur, c'est que tous les processus ont fini (ou sont en cours de finition). Le proc demandeur envoie un message de terminaison avec le tag TAG_END.*
3. *Tous les procs qui reçoivent le message de terminaison ne demandent plus de travail, mais terminent quand leur file locale est vide. Ils propagent le message de terminaison sur l'anneau. Dès*

qu'un proc qui a envoyé un message de terminaison reçoit un message de terminaison, il peut terminer sans faire suivre le message.

4. *Le proc qui a propagé le message de terminaison attend de recevoir un message de terminaison avant de terminer.*

Cet algorithme marche bien et est efficace (voir partie II).

2. Parallélisation hybride MPI+OpenMP

Dans cette partie, nous mettons en place une parallélisation hybride MPI+OpenMP afin d'exploiter les processeurs multicoeurs dont nous disposons. Le modèle hybride utilise OpenMP pour distribuer la partie de calcul de chaque processus MPI entre les threads. Cette partie est celle qui nous a le plus posé de problème. En effet, ajouté une simple directive *#pragma omp parallel for* modifiait parfois considérablement le comportement de notre algorithme. Par exemple, nous avons repris à l'identique notre code en MPI qui marche parfaitement bien. En y ajoutant la directive *#pragma omp parallel for schedule(runtime)* à la boucle sur les échantillons (for s), notre temps d'exécution augmentait considérablement lorsqu'on utilise énormément de processus (>= 32 avec 14 machines). L'explication que nous avons trouvé à cela, c'est que dans cette boucle, les coûts des communications entre les threads étaient importants. On a aussi essayé de détacher un thread de communication lorsque le processus reçoit une requête de demande afin que le thread principal continue les calculs. Pour cela, lorsque MPI_Iprobe renvoie true, on détache un thread avec :

```
#pragma omp parallel num_thread(2)
#pragma omp task nowait
#pragma omp single
[Communication]
```

Ainsi, le thread 1 faisait la communication pendant que le thread 0 continuait les calculs. Mais les résultats n'étaient pas satisfaisant. Le temps de communication augmentait. L'explication qu'on a trouvé est que la communication durant les calculs n'a pas un coût élevé (il y a lieu peu de communication). Il est donc plus rapide de le faire en séquentiel que de créer des threads (car la création des threads a un coût). Nous avons donc abandonné cette piste.

Pour la parallélisation avec openMP, on ne pouvait paralléliser la boucle principale (boucle sur la longueur for i) ni celle sur la largeur (boucle for j) car à la fin de la dernière, nous testons si le processus a reçu une requête de demande de travail auquel cas il entame une communication. Le test et la communication ne pouvant être réalisés qu'une fois (donc par un seul thread), ces boucles ne sont donc pas parallélisables. On a alors parallélisé la boucle sur les échantillons (for s in samples) mais aussi celle du processus *root* où il reçoit les résultats des autres processus. En effet, en mpi, le processus *root* parcourt tous les processus de 1 à *np-1* et reçoit leurs données. Ce qui implique que si le processus 7 finit par exemple en premier et est prêt à envoyer ses données, il doit attendre que *root* reçoit d'abord les données de tous ceux qui le précède. En parallélisant cette boucle, son temps d'attente est divisé par le nombre de thread.

Le cas erand48 : En consultant le manuel de erand48, il était marqué que cette fonction n'est pas thread-safe car si on l'appelle simultanément à partir de plusieurs threads, les nombres aléatoires peuvent ne pas être très aléatoires. Cela est dû au fait que erand48 garde en mémoire les valeurs successives calculées. **Cette fonction n'est donc pas adaptée en mode multi-thread**. Dans le même

manuel, il est aussi marqué que la fonction *random* pouvait nous intéresser. En jetant un coup d'oeil à son manuel, il est marqué que cette fonction était thread-safe. Super! Toutefois, il est aussi mentionné qu'elle diminuait la performance du programme.pas vrai ça! En continuant la recherche sur random,nous avons été renvoyé vers la fonction r*and()*. Et dans le manuel de *rand*, nous avons aussi la fonction ***rand_r*** qui est marqué thread-safe et efficace en mode multi-threads. Youpii ! Néanmoins un problème demeure.En effet, *erand48* prend en entrée *unsigned short** et renvoie un *double* entre 0.0 et 1.0 alors que *rand_r* prend en entrée *unsigned int** et renvoie un *int* entre 0 et *RAND_MAX*. Pour résoudre ce problème, Nous avons donc dû changer le type de la variable *PRNG_state* de *unsigned short[3]* à *unsigned int** et convertir la sortie en *double* avant de le diviser par *RAND_MAX*.

3. Parallélisation MPI+OpenMP+SIMD et SIMD

Dans cette partie, nous apportons de la vectorisation à notre algorithme. Nous avons vectorisé le code séquentiel en utilisant les fonctions intrinsèques dans les calculs vectoriels dans un premier temps,et dans un second temps, nous avons laissé la vectorisation au compilateur en lui signifiant les parties vectorisable de notre code avec la directive *#pragma omp simd* ajouté à la boucle sur les échantillons et la déclaration de toutes les fonctions traitant de calcul vectoriel vectorisable avec la directive *#pragma declare simd*. Pour finir, nous avons vectorisé en openmp notre code hybride. Cela nous permet ainsi de mesurer les différents gains de performances et de les comparer.

II. RÉSULTATS, ANALYSES ET COMPARAISONS

1. Résultats

Dans cette section, nous présentons les résultats des mesures de performance effectuées sur chaque version de notre code. Notez que tous nos tests ont été effectuées sur les machines de la salle **ppti-14-408** et souvent celles de la salle **ppti-14-406(**pour les tests avec 64 coeurs**).**
Taille image (moyenne) : w=2000 h=1500 samples=500
Temps d'exécution en séquentiel : 17 205.09 secondes

- **Mesures de performance MPI pure** (codes sources dans *par_mpi/*) :

Nombres de processus	4	8	16	32	64
temps d'exécution (*secondes*)	4168.52	2079.634	1035.63	570.195	334.001
speed-up	4.13	8.27	16.61	30.17	51.51
efficacité (%)	103.25	103.37	103.81	94.28	80.48

Tableau 1 : Mesure de performance MPI pure

- **Mesures de performance MPI-OpenMP hybride** (codes sources dans *par_mpi-omp/*) :

Nombres de processus (p) / Nombres threads(th)	8p - 2th (16 procs)	8p - 4th (32 procs)	16p - 2th (32 procs)	16p - 4th (64 procs)	32p - 2th (64 procs)
temps d'exécution (*secondes*)	496.232	454.265	420.06	250.370	268.992
speed-up	34.671	37.87	40.95	68.71	63.96
efficacité (*%*)	216.69	118.34	127.96	107.35	99.93

Tableau 2 : Mesure de performance MPI-OpenMP

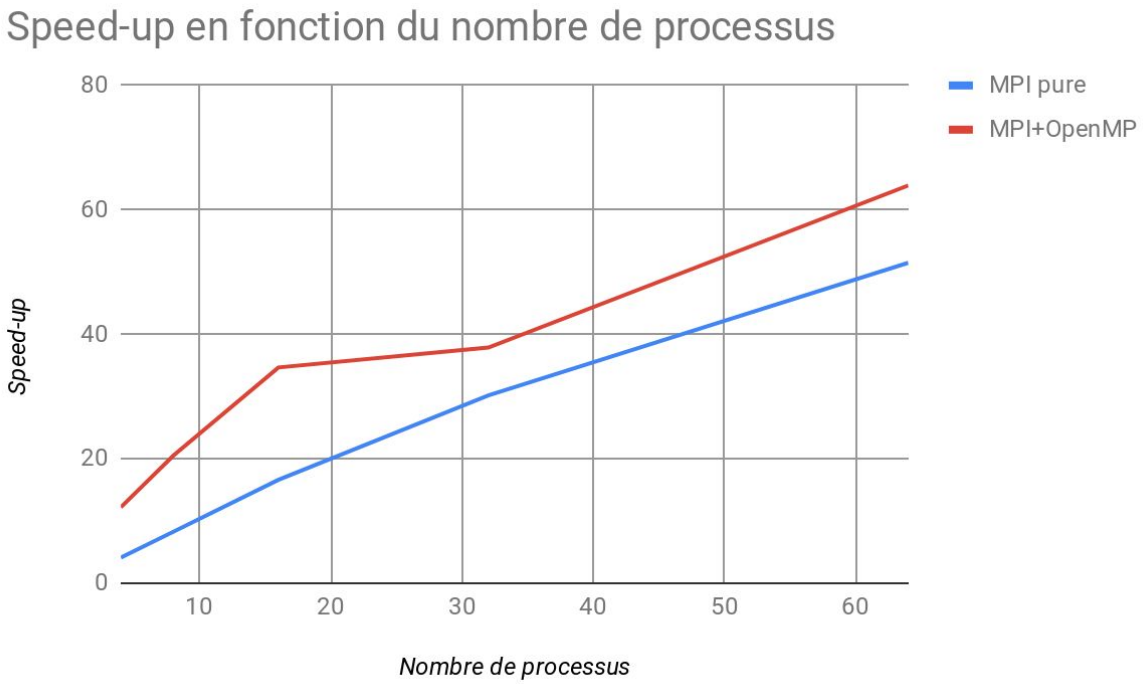

figure 1 : Speed-up en fonction de processus MPI et MPI+OpenMP

- **Mesures de performance MPI-OpenMP et SIMD**(sources : *par_mpi-omp-simd/ et simd/*) **:**

Taille image (petite) : w=300 h=200 samples=200

Temps d'exécution en séquentiel : 245.787 secondes

	Vectorisation intrinsèque	Vectorisation omp - simd	Meilleur algo avec unités simd (mpi+openmp+simd)
Temps d'exécution (secondes)	457,493	192,586	33,83
Gain de performance (Temps seq / Temps d'exécution * nombres processus)	0,537	1,276	0,908

Tableau 3 : Mesure de performance MPI-OpenMP-simd

- **Mesures de performance MPI Pure vs OpenMP Pure vs Hybride (**codes sources omp dans *par_omp/* **) :**

Dans un souci comparatif, nous avons décidé de relever les mesures de performance de chaque outil en solo.

Taille image (petite) : w=300 h=200 samples=200

Temps d'exécution en séquentiel : 245.787 sec

	Temps d'exécution (*en sec*)			**Speed-up**			**Efficacité** (*en %*)		
Nombre de coeurs cpu (procs / threads)	MPI	OpenMP	MPI + OpenMP	MPI	OpenMP	MPI + OpenMP	MPI	OpenMP	MPI + OpenMP
2	77,4916	194,612	77,4916	3,17	1,26	3,17	158,5	63	158,5
4 (2 machines, 2 threads en hybride)	35,6391	99,8775	51,93	6,89	2,46	4,73	172,25	61,5	118,25
8 (4 machines 2 threads en hybride)	18,601	55,6019	8,1	13,21	4,42	30,34	165,25	55,25	379,25

Tableau 4 : Comparaison mpi, openmp et hybride

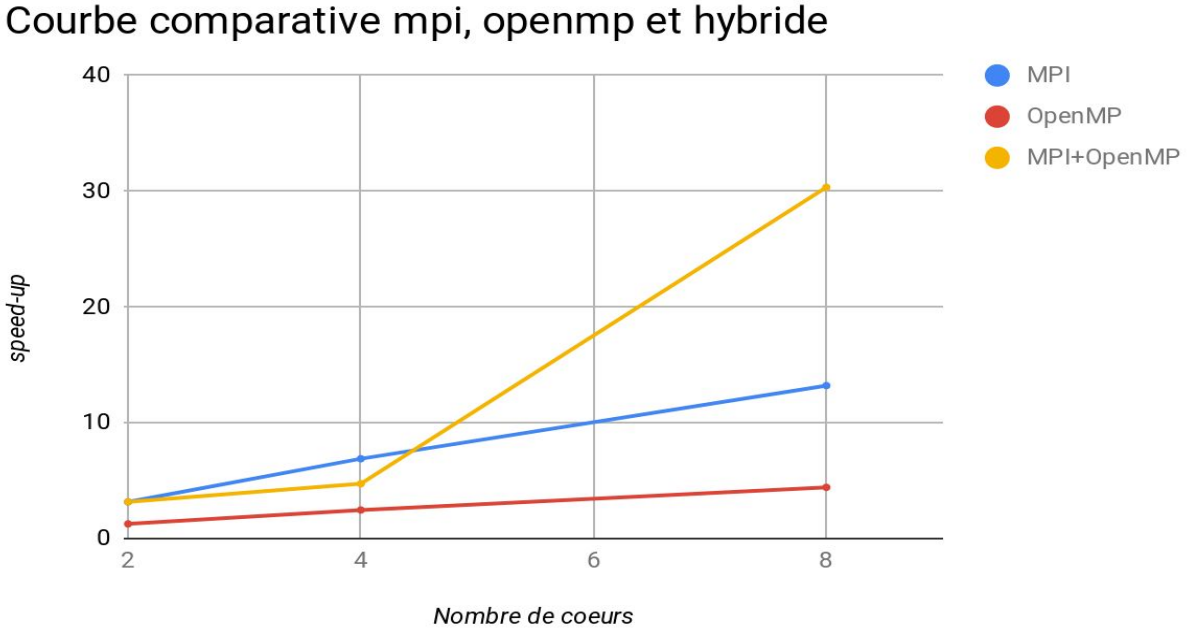

figure 2 : Speed-up en fonction de processus MPI,OpenMP et MPI+OpenMP

2. Analyses et comparaisons

Le calcul des performances pour l'algorithme en mpi pure (tableau 1) montre que le speed-up est linéaire (le temps de calcul est proportionnelle a le nombre de processus). Le benchmark du speed-up

en fonction du nombre de processus pour cet algorithme (figure 1) confirme cela. Nous avons bien une courbe linéaire. L'efficacité est aussi au rendez-vous puisque on est à 100% d'efficacité jusqu'à 32 processus. Nos processus sont donc tous occupés à 100 %. Le temps de communication avec le mécanisme de l'anneau logique de processus est 2*NP avec NP le nombre de processus. Le temps de communication augmente donc linéairement avec le nombre de processus. Mais la bonne répartition des tâches entre les processus remédie à cette augmentation. Toutefois, à partir de 32 processus, on constate un léger baisse de l'efficacité. Au delà de ce nombre de processus, plus le nombre de processus augmente, plus on perd en efficacité. On peut en déduire que le seuil du nombre de processus pour des gains de performance et une exécution efficace est 32. Au delà, le coût de la communication entre les processus devient important.

Le calcul des performances pour l'algorithme hybride (MPI+OpenMP) (tableau 2) est quant à elle très intéressante. On constate un gain de performance plus important que dans l'algorithme en mpi pure. En effet, pour 16 coeurs cpu, on constate que l'algorithme en hybride (8 processeurs et 2 threads) a un speed-up et une efficacité largement supérieure (environ 2 fois plus) à celle en mpi pure (16 processus). De plus, avec 32 coeurs cpu, là où on perd en efficacité avec l'algorithme mpi pure (32 processus), on conserve toujours notre efficacité en hybride (16 processus et 2 threads / 8 processus et 4 threads). Le speed-up est toujours largement supérieur à celui de l'algo en mpi pure. On fait aussi la même remarque pour 64 coeurs cpu. **Ainsi, pour un même nombre de coeurs cpu, l'algorithme en hybride est plus performant et plus efficace que l'algorithme en mpi pure.** Il est toutefois intéressant de noter que la performance de l'algorithme en hybride varie considérablement selon le choix du nombre de processus et du nombre de thread et ceux, même pour un même nombre de coeur cpu. Par exemple, comme relevé dans le tableau, pour 32 coeurs cpu, il est plus performant et plus efficace d'utiliser 16 processus et 2 threads que d'utiliser 8 processus et 4 threads pendant que pour 64 coeurs cpu, il est plus efficace et plus performant d'utiliser 16 processus et 4 threads que 32 processus et 2 threads. En effet, on a expliqué précédemment qu'en mpi, on commençait à perdre en efficacité à partir de 32 processus. Même en ajoutant des threads, la baisse de performance de la partie du code hybride en mpi entraîne une baisse de performance de tout l'algo. De plus, on sait que la communication entre les threads est moins coûteux que celle entre les processus mpi. Voilà pourquoi il est plus performant et plus efficace de se limiter à 16 processus en augmentant le nombre de threads. On en déduit également que le speed-up dépend de la configuration entre threads et processus. Cela expliquerait alors pourquoi le speed-up de cet algorithme ne soit pas linéaire (figure 1).

Quant à la mesure du gain de performance pour les différentes versions (tableau 3), elle nous permet de constater qu'avec la vectorisation avec les fonctions intrinsèques, on a plutôt une perte de performance par rapport à l'algorithme séquentiel alors qu'avec la vectorisation en openmp (directives #pragma pour laisser la vectorisation au compilateur) on a un gain de performance important. **On peut en déduire que le compilateur effectue une meilleur vectorisation que nous.** Toutefois, nous perdons ce gain lorsqu'on décide d'apporter de la vectorisation (openmp-simd) à notre meilleur algorithme parallèle (mpi + openmp). En effet, l'utilisation des structures en mémoire partagée peut induire une forte limitation de la scalabilité. L'utilisation de la vectorisation dans ce cas peut alors causer une diminution de performance. Lorsque plusieurs processeurs manipulent des données différentes mais adjacentes en mémoire, la mise à jour d'éléments individuels peut provoquer un

chargement complet d'une ligne de cache, pour que les caches soient en cohérence avec la mémoire. Tout ceci peut avoir un impact sur la performance.

De tout ce qui précède, on peut conclure que la version la plus performante est la parallélisation hybride mpi+openmp. Mais, pour approfondir notre compréhension et mieux expliquer ce résultat, nous avons décidé de prendre les mesures séparées de l'exécution du code avec une parallélisation openmp pure (codes sources disponible dans le répertoire par_omp/), mpi pure et d'une exécution hybride mpi+openmp avec un même nombre de coeur cpu. Les résultats sont consignés dans le tableau 4. Ce tableau nous montre que la version openmp pure a un speed-up sous-linéaire (figure 2) et une efficacité maximale de 63% atteint avec 2 threads. Au delà de ce chiffre, plus le nombre de thread augmente, plus on perd en efficacité. Les performances de notre machine ne sont pas exploitées au maximum avec cette version bien que nous utilison la clause *schedule(runtime)* dans la directive for (répartition de charge dynamique). En effet, en mpi, chaque processus écrit ses résultats dans une image temporaire avant de les envoyer au processus root après les calculs. Il n'y a donc pas accès concurrent à la mémoire (lors de l'écriture dans l'image). Ce qui n'est pas le cas avec openmp pure. L'accès en mémoire de manière concurrente par plusieurs threads peut avoir un impact sur la performance. C'est la raison pour laquelle notre version openmp pure est moins efficace que la version mpi pure. Le modèle hybride reste quant à lui plus performant que les deux modèles précédents. En fait, ce modèle nous offre une optimisation du temps de communication car les threads openmp communiquent via des zones mémoire partagée ce qui rend leur temps de communication léger mais aussi une optimisation de la consommation de mémoire totale grâce à l'approche mémoire partagée OpenMP, un gain au niveau des données répliquées dans les processus MPI et de la mémoire utilisée par la bibliothèque MPI elle-même.

CONCLUSION

A travers ce projet, nous avons pu voir les possibilités et l'importance de la programmation en parallèle. En effet, même pour des calculs simples, s'ils deviennent vraiment nombreux, la différence de temps de calcul peut vite devenir un paramètre à ne pas négliger. Dans ce projet, nous avons relevé et examiné les performances d'une parallélisation MPI pure, MPI+OpenMP et l'apport d'une vectorisation. Mais nous avons aussi comparer les performances MPI, OpenMP et MPI+OpenMP sur des machines multiprocesseurs. MPI pure est plus performant et plus efficace que OpenMP mais les deux unissant leur force, sont meilleurs que tout autre version dans ce modèle de calcul que nous avons traités. La vectorisation quant à elle apporte un gain de performance par rapport au code séquentiel lorsqu'il est bien géré sinon on perd en performance. Mais celle-ci ne s'associe pas aisément avec l'hybride compte tenu de la perte de performance que nous constatons. Toutefois, même s'il n'était pas le sujet de ce projet, il est bien de savoir que chaque processus MPI alloue de la mémoire supplémentaire pour gérer les communications et l'environnement MPI. Plus le nombre de processus augmente, plus la mémoire utilisée augmente. Cependant, les threads OpenMP eux utilisent moins de mémoire. Pas de mémoire supplémentaire grâce à la mémoire partagée. Ainsi, la version hybride MPI+OpenMP est plus performante, plus efficace et utilise moins de mémoire. Voilà pourquoi nous privilégions définitivement cette version.

Bibliographie

[1] Restinel. **Introduction à la programmation parallèle**,date de mise en ligne 06/12/2013 date de consultation 09/05/2019. Disponible https://openclassrooms.com/fr/courses/1559481-html5-web-workers-le-monde-parallele-du-javascript/1559586-introduction-a-la-programmation-parallele

[2] Jean-Paul SANSONNET. **Introduction au parallélisme et aux architectures parallèles** date de mise en ligne 10/08/2017 date de consultation 11/05/2019 Disponible à https://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/architectures-materielles-42308210/introduction-au-parallelisme-et-aux-architectures-paralleles-h1088/

[3]Sorbonne Université, Projet M1 Info SFPN - MAIN4 / HPC (Mars 2019)

[4] Frédéric Gava et Gaétan Hains:notes du cours **Introduction à la Programmation parallèle: MPI,** Repéré dans http://lacl.univ-paris12.fr/gava/cours/M2/PSSR/cours_mpi.pdf