POLYTECH SORBONNE - MAIN4

Présentation de nos algorithmes

Projet HPC : rendu photo-réaliste avec illumination globale

LIU Vincent
SAYAH Samir

Encadrant : Charles Bouillaguet
Pierre Fortin

2019

Table des matières

1 Partie I : Parallélisation MPI

Un équilibrage de charge dynamique est justifié pour cette application puisqu'en répartissant les charges de manière statique, on constate que la répartition n'est pas équilibrée. En effet, certains processus finissent bien avant les autres. L'avantage d'une répartition de charge dynamique est qu'elle permet de donner des calculs à faire aux processus ayant fini leur calcul.

Pour résoudre ce problème, deux possibilités s'offrent à nous :

— la méthode patron-ouvrier

— la méthode auto-régulé

La méthode patron-ouvrier consiste à avoir un processus indépendant des autres, appelé le patron. Il connaît les tâches à faire et c'est lui qui les distribue aux autre processus. Cette méthode présente des inconvénients :

— le patron ne travaille pas

— le patron peut devenir un goulet d'étranglement s'il y'a trop de processus

Il nous a été demandé d'implémenter une parallélisation du programme par une méthode auto-régulée pour pallier ces problèmes.

1.1 Présentation de notre programme

Nous avons alors pensé à une structure en anneau.
Au départ, les tâches de travail sont divisées équitablement sur chaque processus.

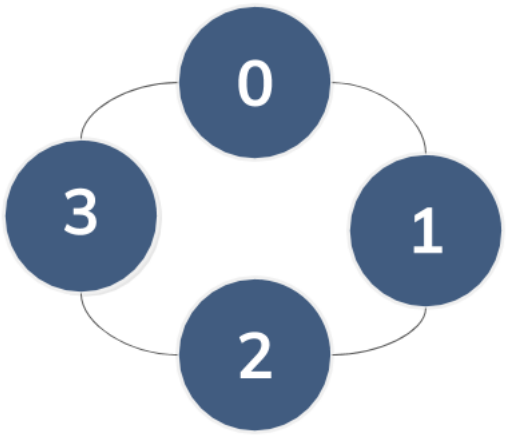

FIGURE 1 – Structure en anneau

Lorsqu'un processus finit les tâches qu'il avait à faire, il va demander du travail au processus qui a le rang suivant.

Dans le meilleur des cas, celui-ci va lui transmettre les tâches à faire.

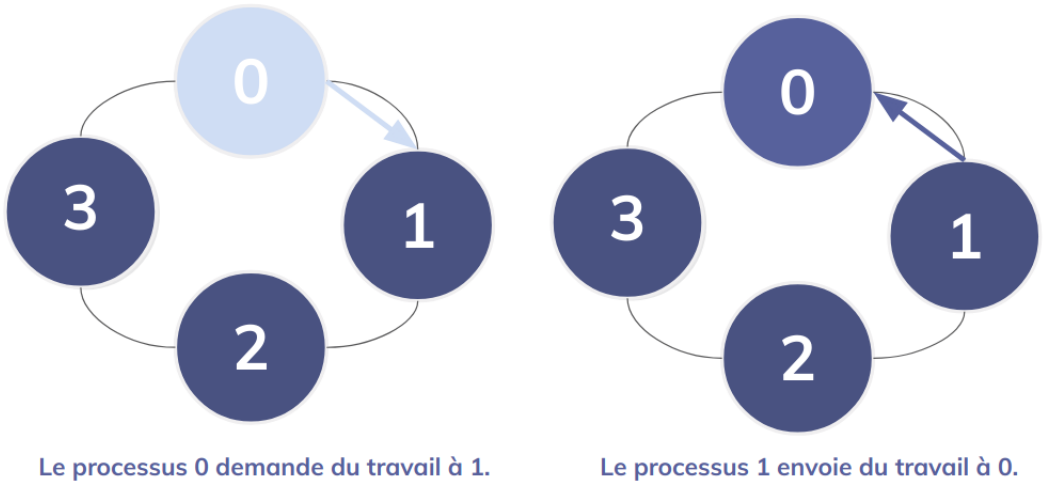

FIGURE 2 – Demande de travail - Un processus demande à son voisin, celui-ci lui envoie du travail

Sinon, le processus qui reçoit une demande de travail va transmettre la demande au processus suivant. Cela arrive lorsque celui-ci n'a pas assez de travail à donner.

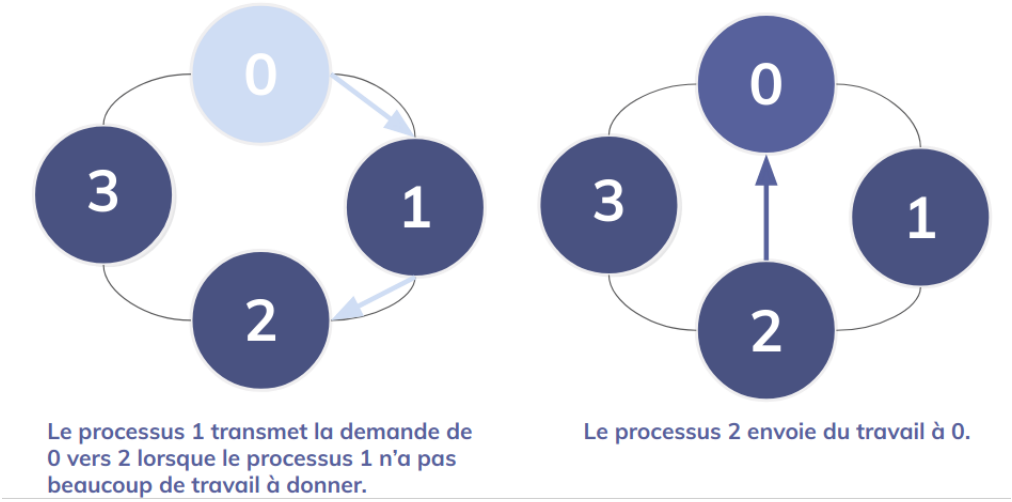

FIGURE 3 – Transmission de la demande de travail

Lorsqu'un processus indique à son voisin qu'il cherche du travail, il se met en attente. C'est-à-dire qu'il va transmettre toutes les demandes de travail des processus car il n'a lui même pas de travail à donner.

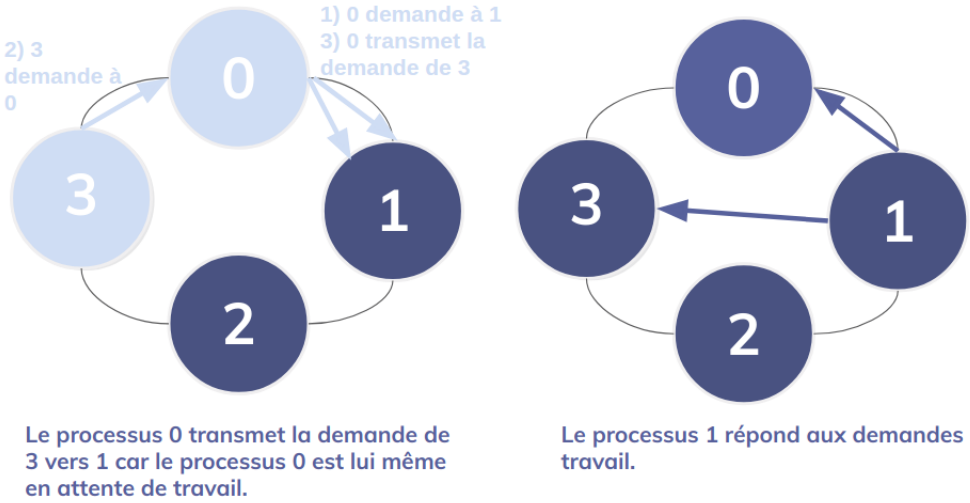

FIGURE 4 – Transmission de la demande de travail - Un processus en attente transmet les demandes de travail

Lorsqu'un processus reçoit sa propre demande de travail, cela signifie qu'il n'y a plus de travail à demander. Pour assurer la terminaison de l'algorithme, les processus en cours de calcul doivent être informés de la fin.

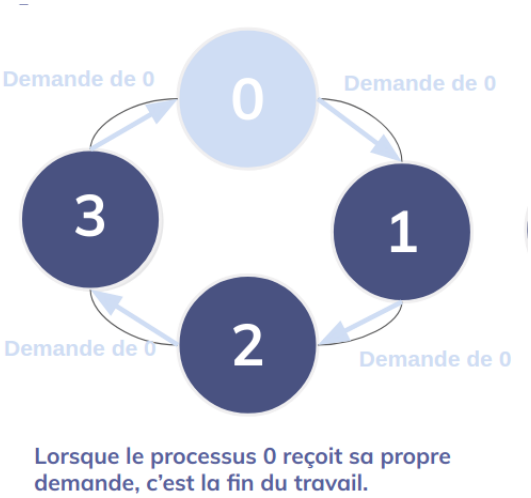

FIGURE 5 – Fin du travail

1.2 Terminaison de notre programme

Nous avons essayé deux méthodes pour la terminaison du programme.
La première consiste à annoncer la fin des tâches à tous les processus lorsque un processus se rend compte de la fin des tâches.

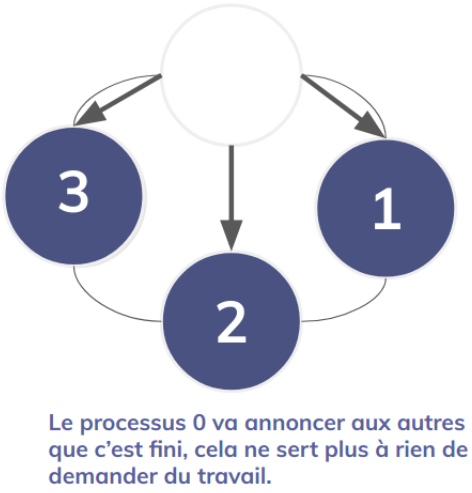

FIGURE 6 – Fin des communications

Le problème avec cette méthode est qu'elle ne garantit pas l'exclusion mutuelle. En effet, il se peut très bien que deux processus se rendent comptent en même temps de la terminaison.

La deuxième méthode consiste à faire circuler un dernier message (technique du jeton) indiquant la fin des calculs. L'exclusion mutuelle est ainsi garantie.

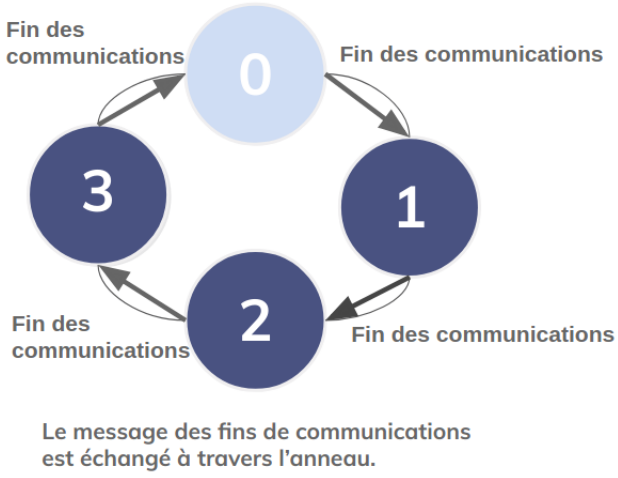

FIGURE 7 – Fin des communications - Version anneau

Enfin, pour rassembler les données, nous faisons appel à l'opération collective **MPI_Reduce**.

1.3 Pseudo Code

Algorithme 1 : Partie 1 : parallélisation MPI

```
1  Initialisation MPI ;
2  tant que ce n'est pas la fin faire
3  |   tant que on a des calculs faire
4  |   |   pixel = next_pixel() ;
5  |   |   si on a recu un msg alors
6  |   |   |   traiter le msg ;
7  |   |   fin
8  |   |   Calcul(pixel) ;
9  |   fin
10 |   demander_travail(suivant) ;
11 |   tant que on est en attente de travail faire
12 |   |   si on a recu un msg alors
13 |   |   |   traiter le msg ;
14 |   |   fin
15 |   fin
16 fin
17 MPI_Reduce() ;
```

1.4 Étude des performances

Afin d'évaluer les performances de notre programme, nous avons comparé deux versions pour deux grains de calcul différents :

- par blocs de lignes
- par blocs de pixels

Nous avons également opté pour deux configurations différentes :

- 1 processus par noeud
- plusieurs processus par noeud avec 1 processus par coeur physique
- plusieurs processus par noeud avec 1 processus par coeur logique.

Les conditions d'expérimentation ont été les suivantes :

- nous avons testé nos programmes sur les ordinateurs 4 coeurs de la salle 327
- Les processeurs sont des Intel(R) Core(TM) $i7-6700$ CPU @ 3.40GHz
- les images calculées ont été en 720p avec 200 échantillons.

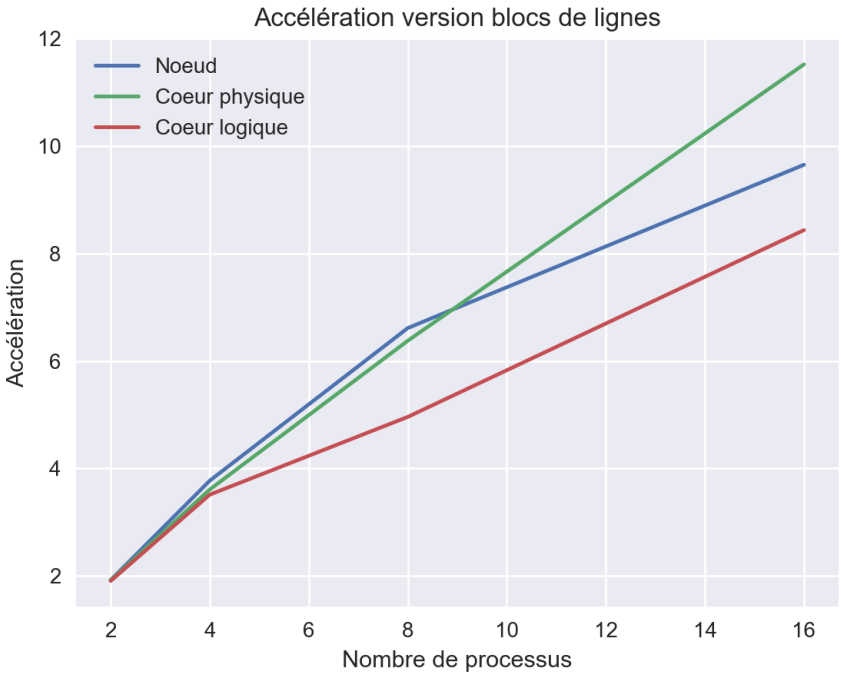

FIGURE 8 – Courbe d'accélération pour une granularité de calcul en blocs de lignes

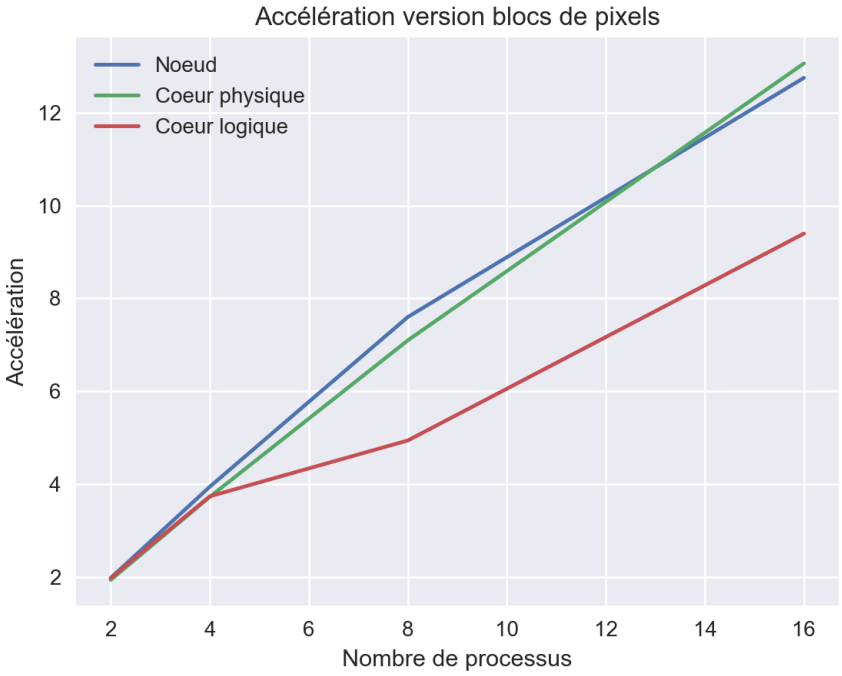

FIGURE 9 – Courbe d'accélération pour une granularité de calcul en blocs de pixels

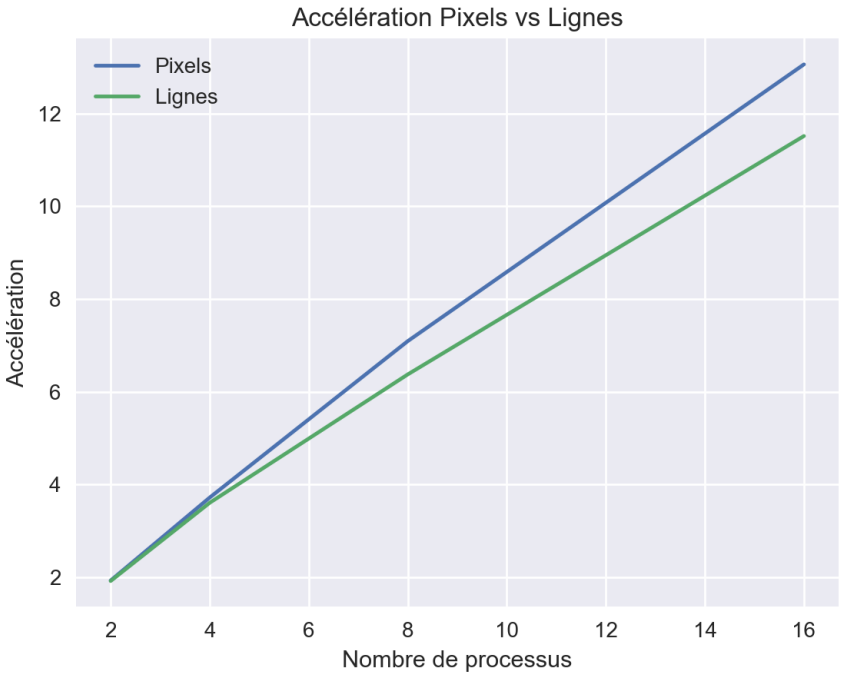

FIGURE 10 – Comparaison des performances des deux versions du programme

On observe que lorsqu'on met un processus par coeur physique, l'accélération est beaucoup plus rapide. En effet,

- Lorsqu'un on met seulement un processus par noeud, les temps des communications réseaux deviennent assez coûteux.
- Lorsqu'on met un processus par coeur logique, comme nous ne disposons pas d'un vrai coeur physique, le calcul est partagé entre deux coeurs logiques, nous utilisons les mêmes ressources.

On observe également que la version où on a affiné le grain de calcul est légèrement plus rapide. Cela peut s'expliquer par le fait qu'une ligne de calcul peut prendre relativement du temps, c'est pourquoi en affinant le grain de calcul, on peut diviser le calcul d'une ligne pour être beaucoup plus rapide.

2 Partie 2 : Parallélisation MPI+OpenMP et SIMD

2.1 Parallélisation MPI+OpenMP

A priori, la version hybride présente certains avantages :

— réduction du nombre de communications

— mise en commum du thread dédié aux communications

2.2 Pseudo Code

Algorithme 2 : Partie 2 : Parallélisation MPI + OpenMP

```
1  Initialisation MPI ;
2  tid = omp_get_thread_num() ;
3  tant que ce n'est pas la fin faire
4  |  si tid == 0 alors
5  |  |  si Si on a recu un msg alors
6  |  |  |  traiter le msg ;
7  |  |  fin
8  |  |  si Demande == 1 alors
9  |  |  |  demander du travail ;
10 |  |  |  Demande = 0 ;
11 |  |  fin
12 |  sinon
13 |  |  p = get_pixel() ;
14 |  |  si p est valide alors
15 |  |  |  Faire le calcul de p ;
16 |  |  sinon
17 |  |  |  Demande = 1 ;
18 |  |  fin
19 |  fin
20 fin
21 MPI_Reduce() ;
```

2.3 Étude des performances

Afin d'évaluer les performances de notre implémentation MPI + OpenMP, nous avons mesuré l'efficacité de deux configurations différentes :

— threads par noeud, c'est à dire un par coeur physique. Dans cette version le thread qui s'occupe des communications tourne tout seul sur un coeur physique.

— 5 threads par noeud, un par coeur physique + le thread qui gère les communications.

Nous avons ensuite comparé les performances avec l'implémentation MPI pur avec un processus par coeur physique.

Les conditions d'expérimentations ont été les suivantes :

— nous avons testé nos programmes sur les ordinateurs 4 coeurs de la salle 327

— Les processeurs sont des Intel(R) Core(TM) $i7-6700$ CPU @ 3.40GHz

— les images calculés ont été en 720p avec 200 échantillons.

Nous avons choisi de répartir un thread par coeur physique afin de maximiser nos performances.

Sur la figure 11, on peut constater que l'utilisation de 5 threads est bien meilleure que celle de 4, et ce quelque soit le nombre de processus/noeud/CPU utilisés. On choisit donc de garder 5 threads pour la comparaison avec MPI pur

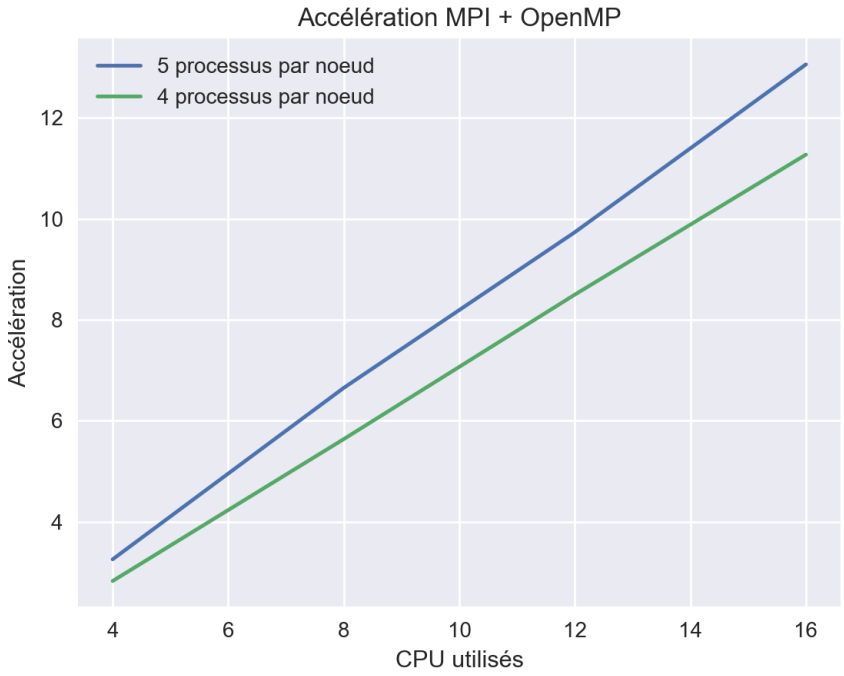

FIGURE 11 – Courbes d'accélération MPI + OpenMP avec 4 et 5 threads

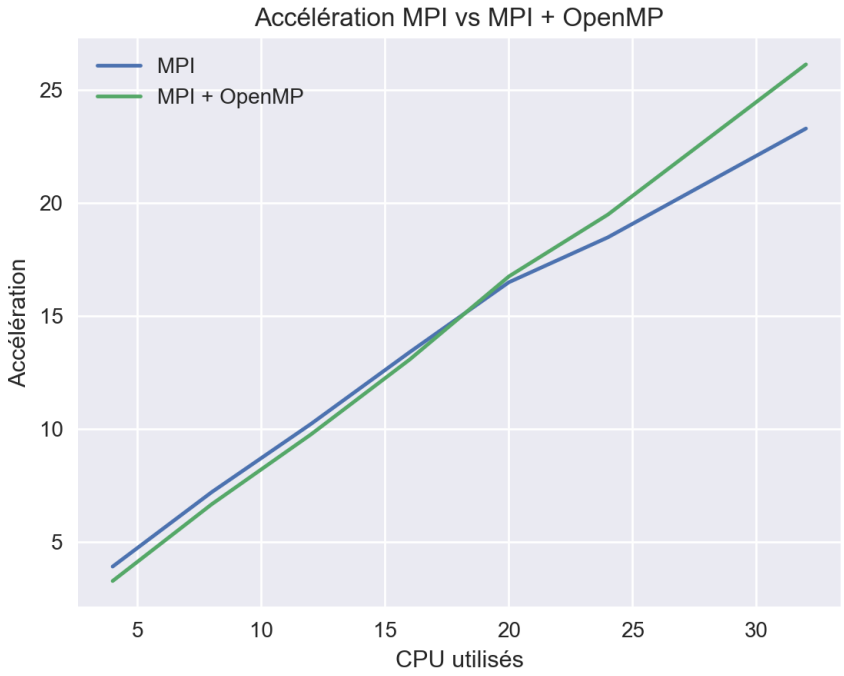

FIGURE 12 – Courbe d'accélération MPI pur vs MPI + OpenMP

Lorsqu'on utilise un faible nombre de processus/noeuds/CPU, l'implémentation MPI est la plus efficace. En effet, le thread en plus dédié aux communications limite les ressources d'un des threads de calcul.

Au delà de 20 CPU utilisés, c'est à dire 20 threads, 4 processus sur 4 noeuds différents, on remarque que l'accélétation de l'implémentation MPI + OpenMP prend le dessus. En effet, au delà de ce seuil les communications deviennent trop coûteuses pour l'implémentation MPI pur et pénalisent celle-ci au profit de l'implémentation MPI + OpenMP.