Projet HPC

Polytech Sorbonne

C – MPI – OpenMP

Rendu photo-réaliste avec illumination globale

Auteurs :

Thizirie Ould Amer
Jordy Ajanohoun

Enseignants :

Charles Bouillaguet
Pierre Fortin

13 mai 2019

Abstract

Dans ce rapport nous présentons les différentes solutions que nous avons réussi à mettre en place pour paralléliser et optimiser le code sequentiel du path-tracing. Nous nous sommes préoccupés uniquement de la boucle principale qui calcule la valeur de tous les pixels. En effet, c'est cette partie qui constitue 97% du temps d'exécution total.

Nous avons mis en place un **équilibrage de charges dynamique** avec **MPI**, au moyen de deux algorithmes différents que nous allons présenter et comparer. Dans les deux cas, la **granularité est fine** : une charge correspond à un pixel donc les processeurs se répartissent les pixels à calculer de manière auto-régulée. Nous présenterons ensuite les résultats obtenus en s'aidant d'**OpenMP** en plus de MPI. Enfin, nous finirons avec les gains obtenus via la parallélisation **SIMD**.

L'ensemble des tests de performance présentés ici ont été réalisés sur un cluster de **16 nœuds homogènes**. Ces 16 ordinateurs sont **interconnectés au sein d'un même réseau** et sont dans une même salle de TP (salle 324, bâtiment Escanglon, Polytech Sorbonne). **Chaque nœud possède 4 cœurs physiques et chacun de ces cœurs physiques dispose d'un SMT à 2 voies**. Soit 8 cœurs logiques au total par ordinateur. La mémoire n'est pas partagée entre les nœuds, chacun a son propre espace mémoire.
CPU : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz.
L'ensemble des temps donnés sont en **seconde**.

Table des matières

1 Parallélisation MPI

Nous avons implémenté deux algorithmes d'équilibrage de charges dynamique, l'un que nous allons qualifier d'**original** (pensé par nous de toute pièce) et l'autre, que nous avons construit pendant le TD avec vous, de **classique**.

1.1 Algorithme original

Data: **w** (image's width), **h** (image's heigth), **p** (nombre de processus), **rank** (rang MPI du processus)
Result: **image** (l'image générée)
begin
  Répartition égale du nombre de pixels à calculer au départ;
  next ← indice du prochain pixel à traiter;
  fin ← indice du dernier pixel à traiter;
  round ← 0 ;
  fini ← faux ;
  while *!fini and next < fin* **do**
    debut ← next;
    while *next < fin* **do**
      CalculerPixel(next, image);
      next++;
      if *message "fin du round" reçu* **then**
        MAJ_Charge(debut, next, round, fini, nb_pixels_calculés);
      end
    end
    Envoyer "fin du round" à tous les autres processus;
    MAJ_Charge(debut, next, round, fini, nb_pixels_calculés);
  end
  if *rank!=0* **then**
    Envoyer les pixels que j'ai calculés au processus 0 round par round;
  end
  else
    Recevoir les pixels calculés par les autres processus round par round;
    Enregistrer l'image;
  end
end

MAJ_Charge(next, fin, round, fini) : Tous les processus s'échangent l'information de ce qu'ils ont fait au round qui vient de se terminer (combien de pixels calculés et lesquels). Ils connaissent donc tous l'état courant de la charge de travail globale et de chacun. A partir de là, la même fonction, entièrement déterministe, est exécutée par tous les processus pour répartir la charge de travail restante de manière égale et le round suivant peut commencer.

Condition d'arrêt : Un processus à qui il reste strictement moins de 2 pixels à calculer ne peut pas donner du travail dans la fonction de MAJ. Donc lorsque pour

tous les processus il reste un pixel ou moins à calculer, la fonction MAJ_Charge met la variable *fini* à vrai.

Communications :

- 1 **MPI_Irecv** par processus au début de chaque round
- Test de la réception après chaque pixel calculé
- **1er processus qui termine ses pixels du round** :
  - **MPI_Send** "*fin du round*" à chaque processus
- **MPI_Allgather** : chaque processus "partage" 3 entiers (combien de pixels calculés pendant le round et duquel auquel)

Mesures de performance

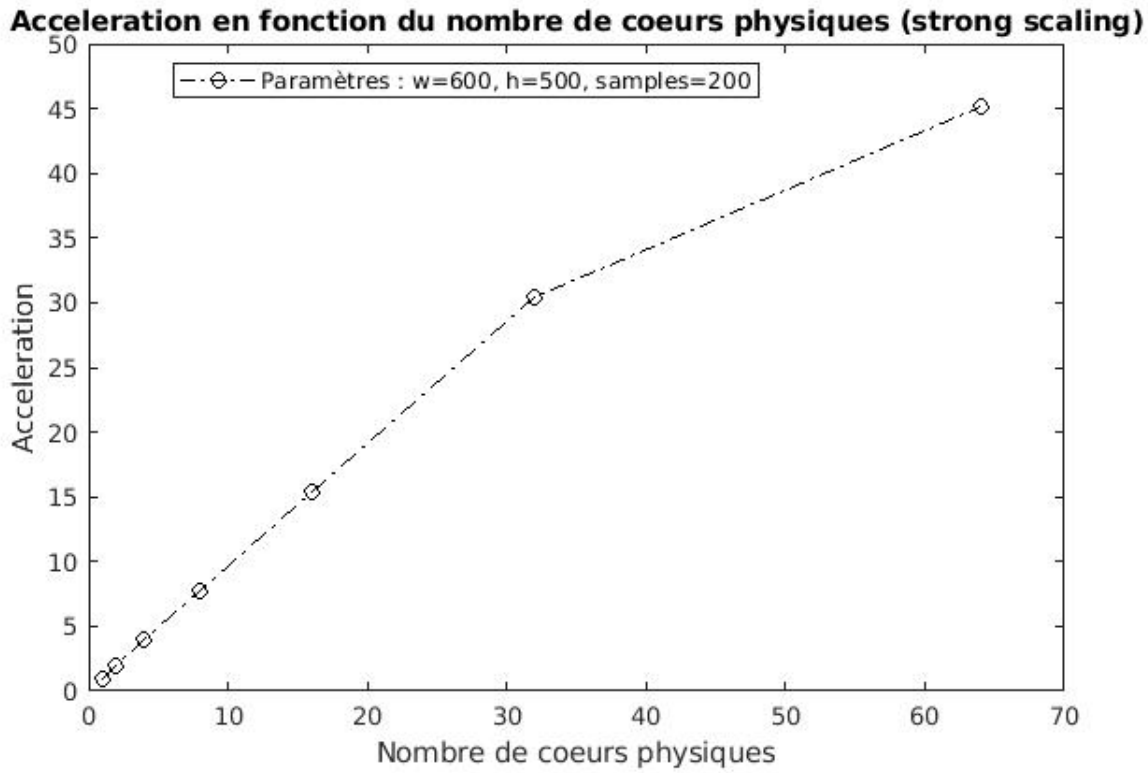

FIGURE 1 – Graphique de **l'accélération en fonction du nombre de cœurs physiques**, en mettant **un processus MPI par cœur physique**. La taille de l'image en entrée de l'algorithme est fixe : 600×500. On remarque que **l'accélération est linéaire jusqu'à 32 processus puis pour 64 processus elle est sublinéaire**. Ceci est dû au fait que lorsque le nombre de processus est grand, les communications prennent de plus en plus de temps. La boucle pour prévenir les autres processus de la terminaison du round prend plus de temps car il y'a plus de processus, et il en est de même pour les MPI_Allgather. Il est très difficile d'éviter ce problème de passage à l'échelle. Ces résultats obtenus avec notre algorithme personnel est tout de même satisfaisant pour nous car on ne peut mieux faire qu'une accélération linéaire. De plus, **le passage à l'échelle ne se comporte pas si mal, la baisse de performance n'est pas dramatique ni drastique**.

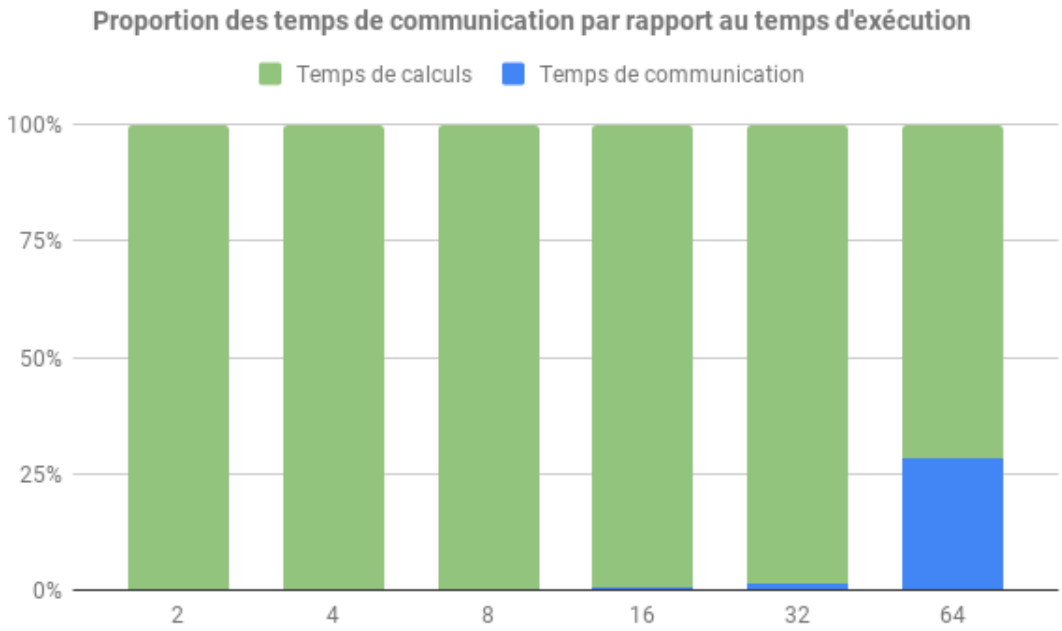

FIGURE 2 – **Part des temps de communication dans le temps d'exécution total en fonction du nombre de processus (1 processus par cœur physique. Paramètres : w=600 h=500 samples=200**. On confirme ce qui a été observé et déduit avec le graphique précédent. En effet, la part de communication avec 64 processus devient assez importante.

Dans toutes les mesures de performance on compte le temps de récupération de tous les pixels calculés. Dans cet algorithme, **nous récupérons les pixels calculés via une boucle for sur le nombre de rounds qu'il y a eu** et pour chaque round on fait un **MPI_Gatherv** où chaque processus envoie, au processus qui récolte, les pixels qu'il a calculé au round en question. **Ceci joue aussi dans le fait que lors du passage à l'échelle, les communications prennent de plus en plus de place dans l'exécution du programme**.

Améliorations possibles

— Dans cette version, les communications ne sont pas recouvertes par du calcul.
— Optimisation de la mémoire utilisée.

1.2 Algorithme classique

Il s'agit de l'algorithme basé sur l'idée du **jeton** mais **sans exclusion mutuelle**. Nous l'avons pensé avec vous en TD à l'avant dernière séance.

Mesures de performance

Nous allons le **comparer** à notre algorithme original précédent.

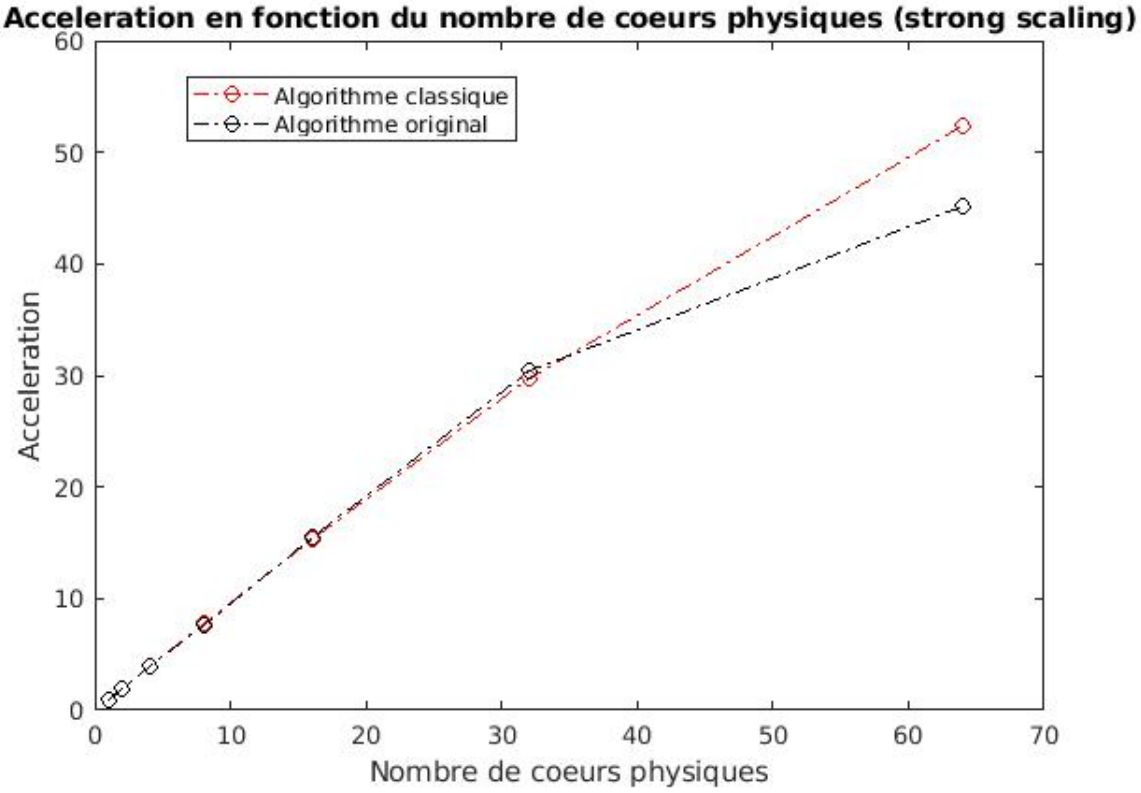

FIGURE 3 – **Comparaison des accélérations en fonction du nombre de cœurs physiques pour les deux algorithmes**. On a placé **un processus MPI par cœur physique**. **Paramètres : w=600 h=500 et samples=200**. On constate que l'algorithme classique est meilleur sur les tests effectués pour 64 processus. Pour un nombre de processus entre 1 et 32 on peut considérer qu'ils sont équivalents en terme de performance. **L'algorithme plus classique passe donc mieux à l'échelle que le notre**. Cependant **l'accélération est toujours linéaire jusqu'à 32 processus et sublinéaire pour 64 processus**. Pour l'algorithme classique, c'est surement les **communications à l'approche de la terminaison** qui sont plus nombreuses lorsque le nombre de processus est grand. Les messages qui circulent dans l'anneau pour s'échanger les derniers pixels à calculer et détecter la terminaison circulent plus longtemps dans l'anneau donc c'est cohérent. De plus, il y a la**réduction** qui est faite à la fin pour qu'un processus récupère toute l'image et c'est une **routine de communication collective**. Plus il y a de processus plus cela prend du temps également.

Améliorations possibles

— Dans cette version, les communications ne sont pas recouvertes par du calcul.
— Optimisation de la mémoire utilisée.

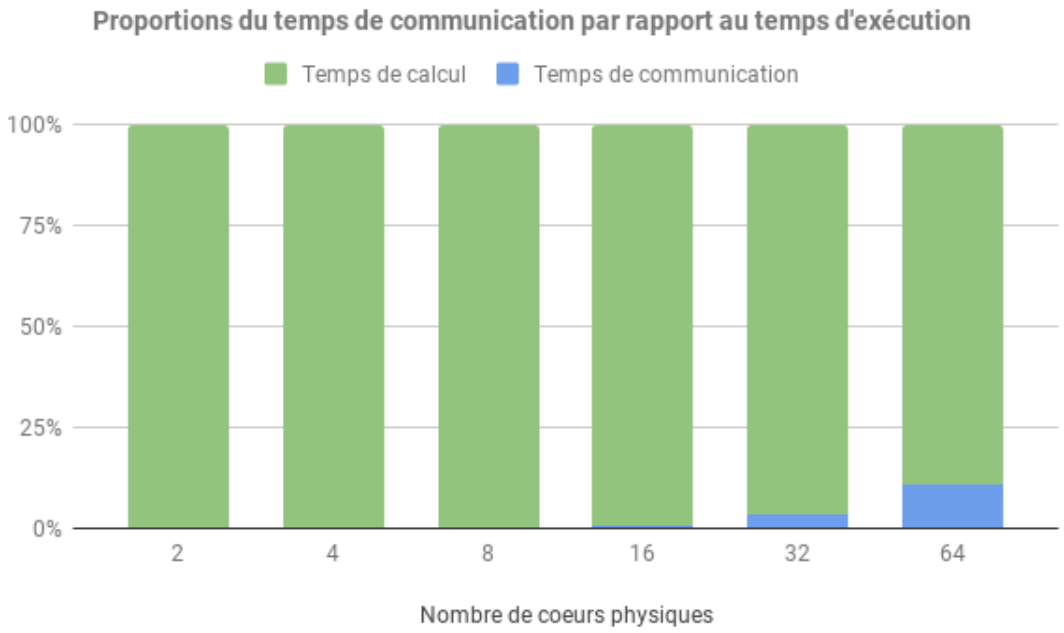

FIGURE 4 – **Part des temps de communication, avec l'algorithme classique, dans le temps d'exécution total en fonction du nombre de processus (1 processus par cœur physique). Paramètres : w=600 h=500 samples=200**. Les temps de communications augmentent moins brutalement comparé à l'algorithme original.

2 Parallélisation MPI + OpenMP

Nous avons implémenté la version MPI + OpenMP de l'algorithme classique, c'est l'algorithme que nous avons construit avec vous en TD. **Les communications sont toutes gérées par le thread 0 et les autres threads calculs**. Il est sensiblement identique à la version MPI seul à la seule différence du thread de communication et des synchronisations entre les threads, sinon le principe est exactement le même.

Mesures de performance

On compare pour **un même nombre de noeuds** :

— le gain obtenu en exploitant au mieux ces noeuds avec MPI seul : **8 processus MPI par noeud** car il y a 8 coeurs logiques par noeud

— le gain obtenu en exploitant au mieux ces noeuds avec MPI + threads : **1 processus MPI + OpenMP (qui génère 8 threads) par noeud**

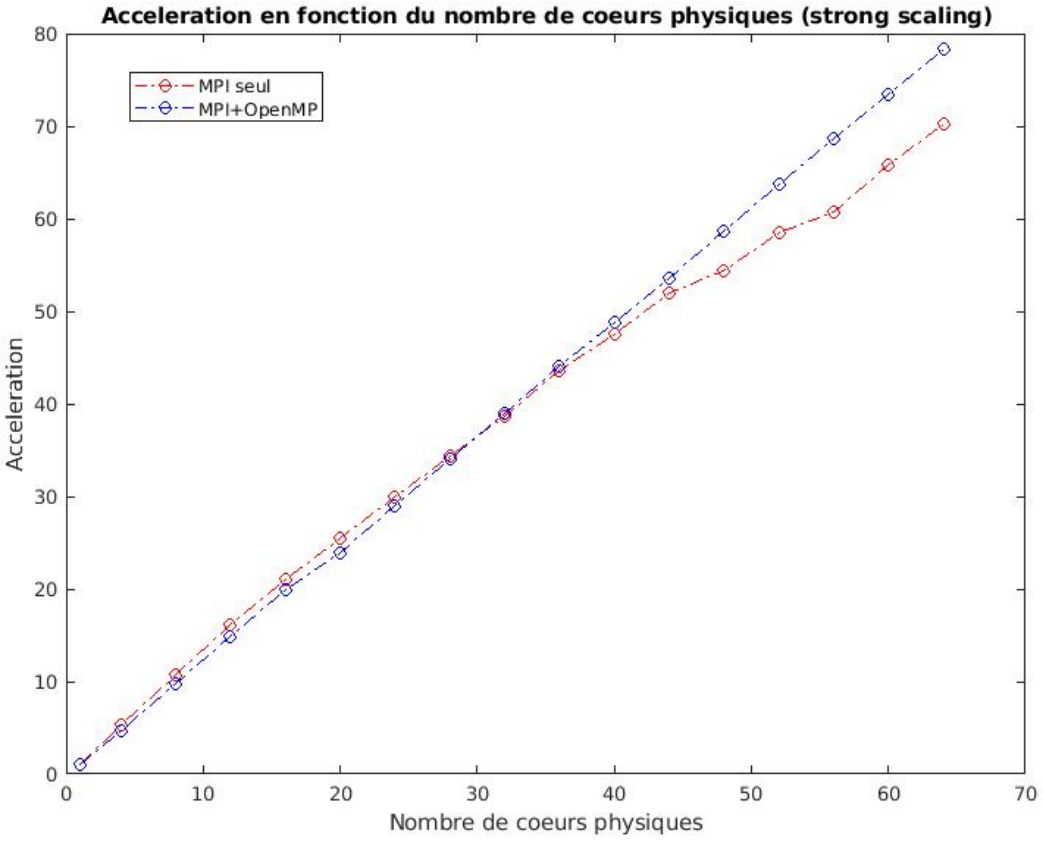

FIGURE 5 – **Comparaison des accélérations de MPI seul et de MPI + OpenMP en fonction du nombre de cœurs physiques (1 thread par coeur logique). Paramètres : w=600 h=500 et samples=200**. De 4 à 28 coeurs physiques MPI seul est plus performant que MPI+OpenMP. Puis de 28 à 36 coeurs physiques les deux sont équivalents en terme d'accélération. Enfin, MPI+OpenMP finit par prendre le dessus en étant plus efficace à partir de 40 coeurs physiques. **MPI+OpenMP passe donc mieux à l'échelle que MPI seul pour l'algorithme classique**.

On constate sans surprise qu'**en utilisant l'ensemble des coeurs logiques on obtient un gain supérieur à celui obtenu en placant 1 processus par coeur physique**. Le gain en exploitant les coeurs logiques est **supralinéaire**. **Nous avons testé différentes configurations pour MPI+OpenMP** en faisant varier le nombre de threads pour chaque processus MPI et le nombre de processus placés sur 1 noeud. Les meilleures performances que nous avons obtenues sont avec la configuration décrite ci-dessus et celle avec laquelle les tests ont été faits.

3 Parallélisation SIMD

Nous avons comparé dans cette partie uniquement les **gains obtenus avec les différentes options de compilation**.

Options de compilation	Temps d'exécution
Aucune	88.574459
-O3 -mavx2 -fopt-info	23.886434
-O3 -mavx2 -funroll-loops	23.651078
-O3 -mavx2 -funroll-loops -ffast-math	21.754654
-O3 -mavx2 -funroll-loops -ffast-math -march=native	20.798558

FIGURE 6 – **Code séquentiel - Paramètres : w=150 h= 100 samples=100**

La **vectorisation automatique** avec gcc a bien fonctionnée car le compilateur a été capable de vectoriser la majeure partie des boucles du micro blas et plus encore.

Le gain le plus conséquent a été obtenu avec -O3 et -mavx2 comme on peut le constater sur le tableau. Les autres optimisations apportent un gain de performance plutôt minime.

Nous avons bien sûr toujours vérifier, lors de tous les tests, que l'image générée était bien correcte.

Merci pour votre lecture !