Projet HPC

Parallélisation du rendu d'une image de synthèse

(1ère partie)

SORBONNE UNIVERSITÉ

CRÉATEURS DE FUTURS DEPUIS 1257

BAMBA Ibrahim &
Nitcheu Paul

2019

Présentation et contexte du projet

- **Entrée :** scène fictive décrite constituée d'objets (opaques, réfléchissants, transparents, colorés…), appareil photo virtuel positionné et orienté.
- **Objectif :** reproduire la photo de la scène fictive que prendrait l'appareil photo virtuel si la scène était vraiment présente dans le monde matériel
- **Caractéristique :** rendu photo-réaliste et simulant quelques effets optiques (non-triviaux)

Présentation et contexte du projet

- **Techniques utilisées :** ray-tracing, path-tracing
- **Condition de départ :** un code séquentiel effectuant le rendu d'une image de synthèse est fourni
- **Difficulté :** Pour obtenir un rendu de bonne qualité, il faut beaucoup d'échantillons par sous-pixel (ex : 5000), cela devient donc très long.

Présentation et contexte du projet

- **Notre but :** Paralléliser le code séquentiel afin de le rendre plus rapide
- **Outils/Techniques :**
- Parallélisation MPI dans un premier temps
- Parallélisation MPI+OpenMP et SIMD dans un second

Travail réalisé

1. Répartition de charge statique

- Le fichier */par-mpi/pathracer_mpi_statique.c* contient le code de la répartition de charge statique des tâches entre les processus.
- Répartition de charge statique par bloc
- Avant le début du travail, chaque processus calcule le bloc sur lequel il doit travailler :

  hypothèse de départ : h%NP = 0 (avec NP=Nbre de Processus et h=hauteur)

  taille de chaque bloc de travail : h' = h/NP

  *Le processus de rang r travaille sur : de r * h' à h' + r * h'*

Travail réalisé

1. Répartition de charge statique

- Chaque processus exécute la boucle principale *pour i de 0 à h'-1*
- Chaque processus, même '*root*', écrit le résultat de ses calculs dans une image locale temporaire
- Une fois les calculs du processus de rang *r* terminés, celui-ci envoie ses *3*h'*w* données calculés au processus root par MPI_gather (*avec w=largeur de l'image*)
- '*root*' reçoit les résultats dans un espace alloué pour l'image finale intégrale

Travail réalisé

1. Répartition de charge statique

Inconvénient : ce type d'équilibrage de charge n'est pas adapté pour ce type d'application :

- En affichant le temps total de calcul de chaque processus, on remarque que certains processus finissent beaucoup plus vite que d'autres (*r* peut finir à *5 mins* et root à *7 mins)*
  - ➔ certains processus sont au repos pendant que d'autres ont encore énormément de travail
  - ➔ la répartition des charges n'est pas équilibrée
  - ➔ Nécessité d'équilibrer les charges pour une meilleure performance

Travail réalisé

2. Équilibrage de charge dynamique

Nous mettons en place un équilibrage de charge dynamique de type auto-régulé en mode multi-processus avec MPI (dans le fichier */par-mpi/pathracer_mpi.c*) :

- L'utilisateur spécifie au lancement du programme le nombre de ligne par bloc (*nlines*)
- Hypothèse de départ : *nlines divise h*
- *nblocs (Nbre de bloc du process) = h / NP * nlines [+ 1 si rang < NP mod h/nlines]*
- Chaque processus possède un tableau de *nblocs* blocs de travail au départ

Travail réalisé

2. Équilibrage de charge dynamique

Principe :

Au départ :

Processus 1

Processus 2

Après un laps de temps :

Travail réalisé

2. Équilibrage de charge dynamique

Principe :

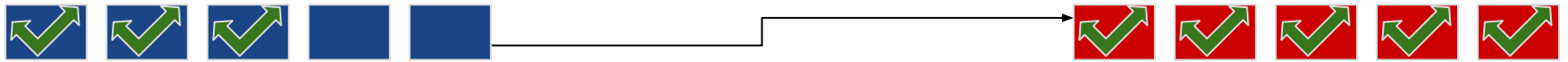
REQ :

Processus 1

Processus 2

Te restes-tu des blocs de tâche ?

REP :



RES

Travail réalisé

2. Équilibrage de charge dynamique

Algo :

*nblocs=h/(np*nlines) + (rank < (h/nlines) mod np) ? 1 : 0, ind=0*

Allocation d'un tableau d'entier de tailles nblocs (blocs)

*Pour i de 0 à nblocs-1, blocs[i] = NP * i + rank*

root alloue l'image intégrale

Initialisation d'une liste chaînée pour les images calculées localement

Tant qu'il reste globalement des blocs de travail (l'image intégrale n'est pas finie) :

Tant que le processus n'a pas fini SES blocs de travail faire :

*Allocation de l'image tampon de taille 3*w*nlines, de numéro ind*

Travail réalisé

2. Équilibrage de charge dynamique

blocs[ind] = -1 (Rend invalide le bloc courant)

ind = (ind+1)%nblocs (incrémentation vers le prochain bloc de travail)

Exécution de la boucle principale (boucle for) de 0 à nlines

- *Après chaque tour de la 2e boucle for, vérifier, si on a reçu une requête (demande de bloc de travail) d'un autre processus*
- *Si oui, lui envoyer la moitié des blocs qui nous reste (en vrai, seul les numéros de blocs concernés sont envoyés). Sinon, continuer*

Fin de la boucle principale

Ajouter le résultat de la portion d'image calculée à la liste chaînée des images calculées

Travail réalisé

2. Équilibrage de charge dynamique

Fin des blocs locaux (2e boucle tant que)

Pour i allant du processus de droite, i != rang, i = (i+1) mod NP :

Vérifier si tous les blocs globaux ont été traités

Si oui, sortir de la boucle

Sinon, vérifier, si on a reçu une requête d'un autre processus

Si oui, lui envoyer un tag de fin.

Régulation des charges : envoyer une requête au processus de droite. Si celui-ci nous envoie des blocs, retourner faire les calculs. Sinon, envoyer une requête au suivant à droite et ainsi de suite jusqu'à faire tous les processus

Travail réalisé

2. Équilibrage de charge dynamique

Fin pour

Si tous les processus nous ont envoyé un tag de fin, envoyer un tag de “Fin Toute” à tout le monde (afin d’éviter que les autres ne refassent le travail à leur tour)

Fin de tous les blocs

root reçoit les images résultats locaux des autres processus

Les autres processus envoient leurs images locaux calculées à root

Libération des tampons

root sauvegarde le résultat

Tests et résultats

- Avec ce mode de répartition, on constate que tous les processus finissent bien quasiment au même moment
  - ➔ Charges équilibrées : répartition de charge dynamique est bien adaptée à cette application
- Tests réalisés sur la petite et la grande image
- Résultats notés dans le tableau ci-dessous

Tests et résultats

1. Tableau 1

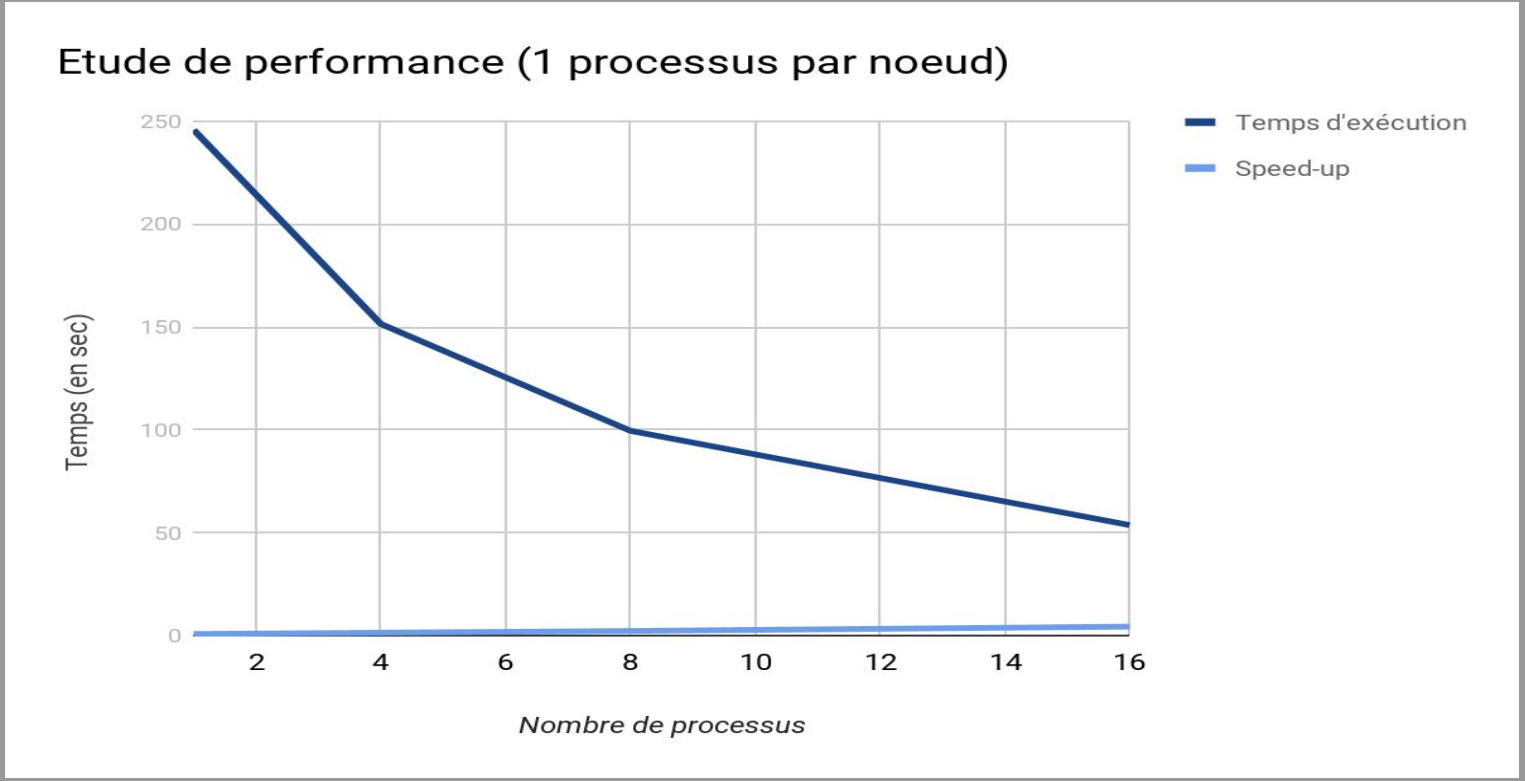
Taille image : w =320 , h=200, samples=200

(Configuration : 1 processus par noeud)

Nombre de processus	**1 (séquentiel)**	**4**	**8**	**16**
Temps d'exécution (*en sec*)	245.787	151.541	99.657	53.877
Speed-up	1	1.621	2.466	4.562
Efficacité (%)	X	40.5	30.8	28.5

Tests et résultats

2. Courbe 1



Tests et résultats

3. Tableau 2

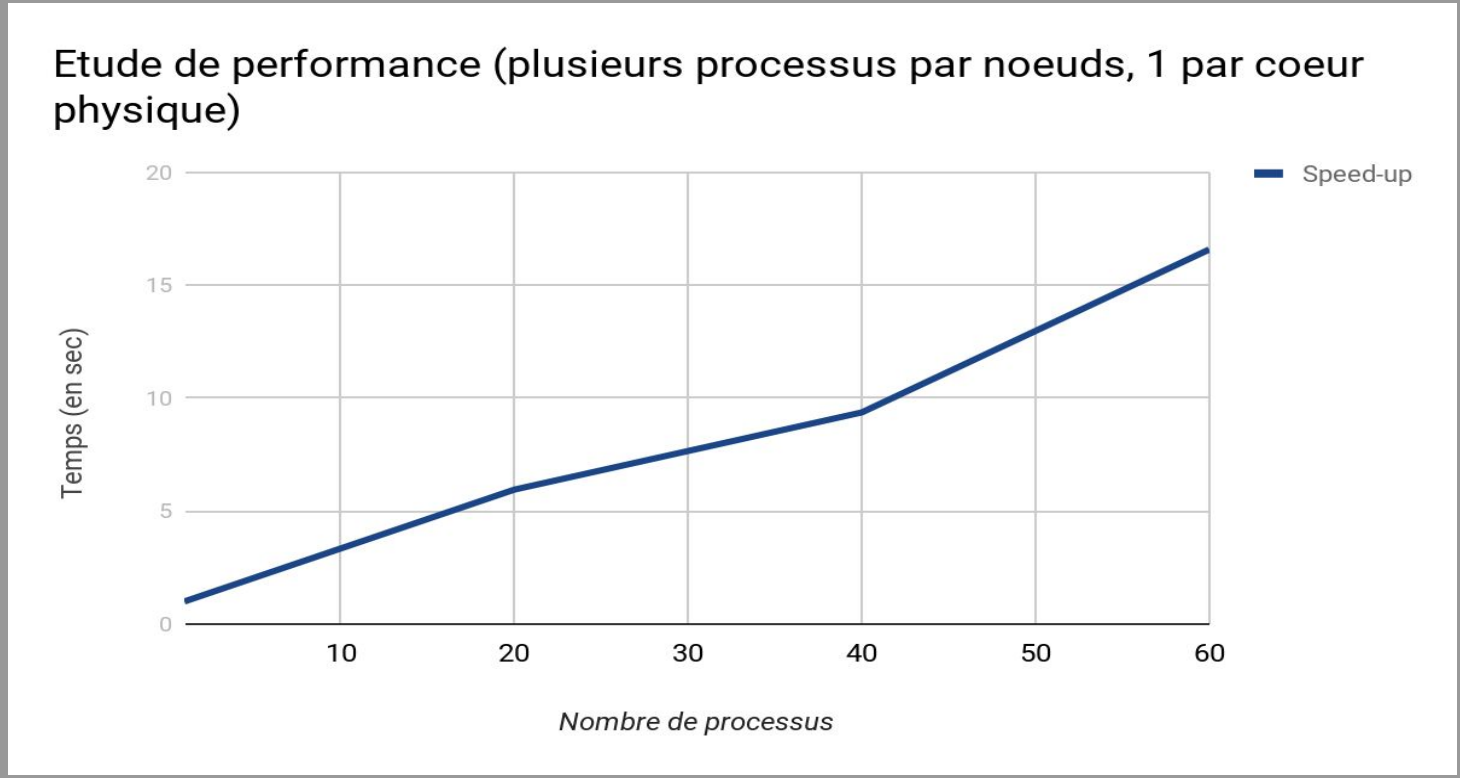
Taille image : w =1500, h=1200, samples=500

(Configuration : 4 processus par noeud, 1 par coeur physique)

Nombre de processus	**1 (séquentiel)**	**20**	**40**	**60**
Temps d'exécution (*en sec*)	env. 5h	3020.73	1920.05	1084.49
Speed-up	1	5.956	9.375	16.59
Efficacité(%)	X	29,78	23,43	27,65

Tests et résultats

4. Courbe 2



Tests et résultats

5. Tableau 3

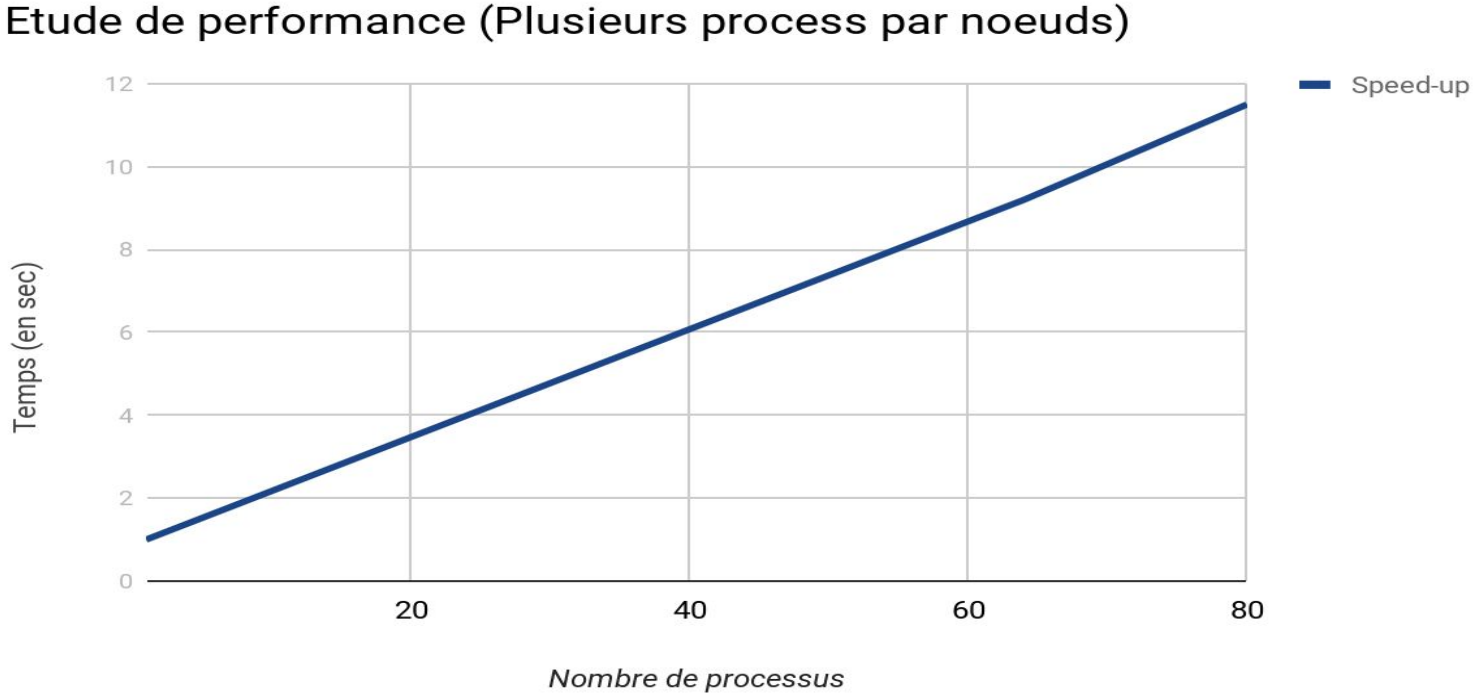
Taille image : w =3840, h=2160, samples=5000

(Configuration : plusieurs processus par noeud, 1 par coeur logique)

Nombre de processus	**1 (séquentiel)**	**64**	**80**
Temps d'exécution	environ 23h	env. 2h30	env. 2h
Speed-up	1	9,2	11,5
Efficacité(%)	X	14,3	14

Tests et résultats

6. Courbe 3



Analyse des résultats

- La courbe du speed-up en fonction du nombre de processus (courbe 1, courbe 2) semble décrire une trajectoire linéaire (-> accélération linéaire) ce qui signifie que nos processus sont occupés à 100 % et donc on a une bonne répartition de charge.
- Le tableau 1 montre qu'à partir de 16 processus, l'augmentation du nombres de processus n'apporte plus de gain de performance.
- Les tests effectués sur la grande image (w=3840, h=2160,samples=5000), permettent de mettre en exergue la complexité liée à ce calcul(plus de 23h pour 1 processus) mais une meilleur amélioration lorsqu'on lance sur plusieurs 80 processus d'où le réel intérêt de la parallélisation dans l'optimisation des codes.