Calcul haute performance

Rendu "photo-réaliste" avec illumination globale

Auteurs

Antoine Mas
Yoann Valeri

Encadrants

S. Moustafa
C. Makassikis

May 12, 2019

Contents

1 Introduction

1.1 Path-tracer

Le path-tracing est une méthode de calcul servant à effectuer le rendu d'une image de synthèse. Aussi appelé technique d'illumination globale, on calcul la quantité de lumière vers un certain point en utilisant un processus de Monte-Carlo. On lance un grand nombre de rayon qui rebondissent dans la scène afin d'obtenir une illumination moyenne. Cela simule naturellement plusieurs effets photos-réaliste.

1.2 Sujet du projet

Ce projet consiste donc en la parallélisation d'un "path-tracer". Ce procédé est assez long si effectué de manière séquentiel, c'est-à-dire si un seul processus s'occupe de faire tout les calculs et de faire le rendu. Ainsi, pour améliorer le temps global, il nous est proposé plusieurs moyen de paralléliser le code, que nous allons utiliser ensemble pour essayer de réduire au plus possible le temps de calcul global.

1. Le premier moyen abordé est OpenMPI, une API permettant d'effectuer un calcul sur plusieurs processeurs distants ou d'une même machine.

2. Le second est OpenMP, une API qui elle joue sur la création de threads, chacun effectuant un certain nombre d'opérations en parallèle d'autres threads.

3. Enfin, nous allons utiliser les instructions SIMD (OpenMP + fonctions intrinsèques), qui permettent de vectoriser des calculs, et donc de pouvoir effectuer plusieurs opérations sur un vecteur d'un seul coup.

1.3 Modification du projet

Dans ce projet, deux cas de test nous sont donnés : calculer une image de taille 320*200 avec 200 samples, et une seconde image de taille 3840*2160 avec 5000 samples. Nous avons calculé que pour faire la seconde image, il nous faudrait au minimum 150 heures. Ainsi, pour pouvoir effectuer des calculs à partir d'une image plus grosse, mais en terminant le temps de calcul séquentiel, nous avons décidé d'ajouter le calcul d'une image 1280*1024 avec 200 samples.

2 Parallélisation OpenMPI

2.1 Problématique

Pour effectuer nos tests, nous utiliserons les machines mises à disposition par la PPTI et nous limiterons nos programmes à utiliser les machines d'une seule salle soit au maximum 64 coeurs.

Le problème central de cette partie OpenMPI va donc être de faire communiquer ces 64 processus pour effectuer le travail le plus rapidement possible. Nous allons maintenant voir de manière pratique quels sont les problèmes qu'il faudra résoudre pour mener à bien ce projet.

2.2 Difficultées rencontrées

Création des tâches : Les images étant trop longues à calculer par un seul processus, il nous faut répartir les tâches entre différentes processus, c'est-à-dire créer un ensemble de plus petites tâches que chaque processus devra effectuer de son côté.

La répartition dynamique : Le temps de calcul de deux tâches peut être radicalement différent, il nous faut faire une répartition dynamique.

Donnees : Un des problèmes liés à ce type d'équilibrage de charge est la répartition des données : quel processus aura quelles données ?

Partage des tâches : Le problème suivant est de savoir à quel processus demander des tâches et combien de tâche donner. On cherche à éviter les communications inutiles entre des processus ne pouvant pas se donner de tâche et ne pas avoir les processus rapide n'aidant pas les processus lents.

Fin des calculs : On doit trouver un moment à partir duquel un processus suppose que tous les calculs (ou presque) ont été fait et arrête de chercher de nouvelles tâches.

Recuperation de l'image : Le dernier problème concerne la manière dont chaque processus va communiquer pour rassembler les données.

2.3 Solution choisie

Pour tout le projet, nous avons choisi d'utiliser les fonctions vu en cours et d'optimiser les communications entre les processus.

Création des tâches : Une première idée que nous avons eu est de faire un découpage 1D par ligne. Cependant, cette solution peut s'avérer inefficace si le temps de calcul des lignes augmente. Nous avons donc choisi de faire des tâches de pixels continus, avec un nombre de pixel déterminé en fonction du nombre de sample. Chaque tâche a un temps de calcul d'environ une seconde sur les machines de la PPTI. Pour nos tests, nous utilisons :

$$\text{Nombre de pixel par tâche} = \frac{200}{320} * \text{Nombre de sample}$$

La répartition dynamique : L'équilibrage que nous allons utiliser est un équilibrage de charge dynamique de type auto-régulé. Chaque processus a des tâches attribuées au debut du programme. Les tâches attribuées pour un processus sont répartis dans toutes l'image pour éviter qu'une zone avec beaucoup de temps de calcul soit donnée au même processus. Ensuite, si un processus est plus rapide que les autres il pourra voler des tâches à un processus plus lent.

Donnees : Pour résoudre ce problème, nous faisons en sorte que chaque processus contienne un tableau d'information sur les tous les processus ainsi que deux buffers contenant le résultats pour le premier et la liste pour le second des tâches qu'il a calculé. Ce tableau d'informations contient la position dans l'arbre des processus.(voir arbre 2.3) Il contient aussi plusieurs informations qui changeront pendant l'execution : le nombre de tâche assignée et le nombre de tâche effectuée. Ces nombres sont des suppositions, sauf pour sa propre case qui contient les valeurs exactes. Ensuite, à chaque communication entre deux processus les tableaux des deux sont envoyées, et les valeurs des cases sont mise à jour.

Algorithm 1: Mise à jour des informations processus

```
input: T tableau initial d'informations des processus
       U tableau reçu d'informations des processus
       n le nombre de processus
for i ← 0 to n do
    if T[i].nbDone < U[i].nbDone then
        // nbDone = nombre de tâche effectuée
        // nbAssigned = nombre de tâche assignée
        T[i].nbDone ← U[i].nbDone;
        T[i].nbAssigned ← U[i].nbAssigned;
    end
end
```

Partage des tâches : Ce problème est résolu grâce au tableau d'informations processus décrit précédemment : lorsqu'un processus a peu de tâche à faire (en dessous d'un seuil), il va chercher dans le tableau le maximum de tâches encore à effecuter pour les autres processus. Actuellement notre seuil est le nombre de processus. Ensuite, si il trouve une victime, il lui enverra un message pour lui demander des tâches (en lui envoyant son propre tableau T). Le processus victime calculera le nombre de tâche à envoyer. Il renverra son tableau d'information avec plus ou moins de tâches à faire. Le voleur mettra à jour son tableau, ajoutera ou non des tâches et continuera son execution.

Algorithm 2: Calcul du nombre de tâche à envoyer

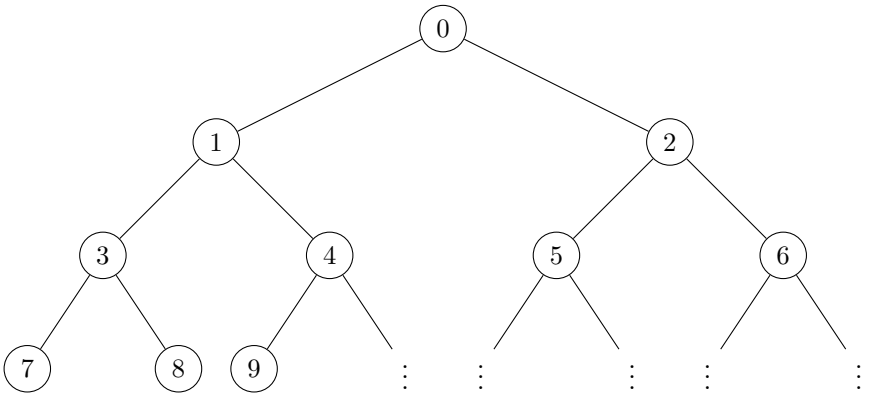
```
input: s le nombre de tâche assignée du processus voleur
       v le nombre de tâche assignée du processus victime
if v < 3 or v + 1 < s then
    return 0;
else
    n ← (v − s)/2 ;
    if n > 5 then
        return 5;
    else
        return n;
    end
end
```

Fin des calculs : Lorsqu'un processus n'a plus de tâche à faire et qu'il n'est pas en train d'en voler, il commence la partie récupération de l'image. Nous definissons un seuil de partage des tâches de façon à ne pas finir alors que les autres processus ont encore beaucoup de tâches.

Récupération de l'image : Une idée simple mais cependant inefficace avec notre modèle est de faire un simple gather à la fin. Le problème est qu'ainsi, tous les processus rapides vont attendre les lents. Nous avons choisi de réimplementer notre gather en faisant une récupération sous forme d'arbre binaire. Un soucis induit par ce mode de récupération est de savoir comment former l'arbre.
- si l'on choisit de manière statique alors si une feuille met beaucoup de temps à calculer ses lignes, tout les autres vont l'attendre, sans rien faire ;
- si l'on choisit un processus de manière dynamique, on a un grand nombre de communication pour former l'arbre.

Nous avons choisi de le faire de maniére statique et d'orienter notre programme autour. Notre arbre est ordonné de la manière suivante :

Arbre de recuperation final



Donc chaque processus aura un parent (sauf le processus 0) et au maximum deux fils. Il aura donc calculé un ensemble de lignes, et les "fusionnera" aux données que ses potentiels fils lui enverront. Ensuite, s'il a un parent, le processus lui enverra ses données. De cette manière, on assure qu'à la fin des envois/récupération de données, le processus 0 aura les données de tout le monde pour rendre l'image.

2.4 Analyse des performances

2.4.1 Mesure des performances

Pour mesurer les temps de calcul de nos programmes, nous avons tout d'abord fait la différence entre le temps au démarrage et le temps final. Cette méthode a quelques défaults, il se peut notamment que le programme mettent plus de temps à s'executer pour des raisons extérieur (autre processus en arrière-plan). Et ainsi, on obtenait des efficacités anormales. Certaines execution était deux fois plus longue que la normale. Cela posait aussi problème pour la grande image, il n'était pas possible d'executer le programme en séquentiel (temps supérieur à 150 heures).

Afin d'améliorer la qualité des tests, nous avons décidé de mesurer le temps passé dans le calcul et de le comparer avec le temps total d'execution. Cela ne prends pas en compte l'amélioration possible grâce aux caches processeurs mais nous permet d'avoir une meilleur visibilité sur nos tests.

Tous les temps et calcul sont disponibles en annexe A.

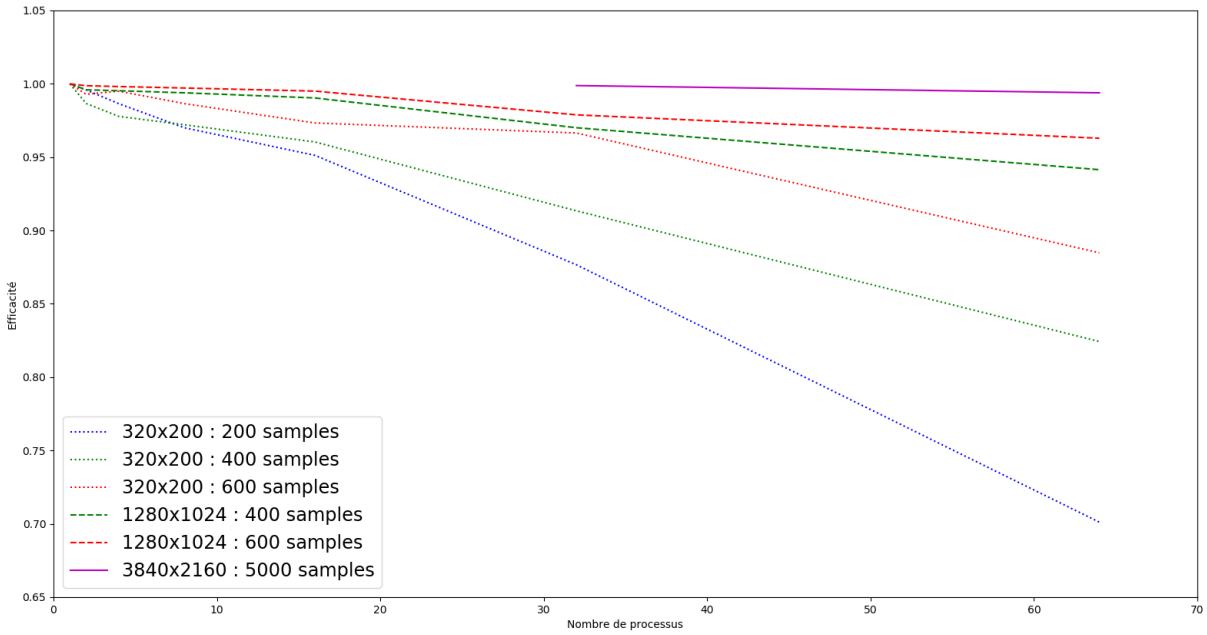


Figure 1: Courbe de l'efficacité en fonction du nombre de processus pour différentes tailles d'image

2.4.2 Analyse

Petite image : 320×200 Comme on peut le voir, on obtient une efficacité de plus de 95% pour tous les cas avec 20 processus ou moins. L'efficacité baisse ensuite dû au fait que le nombre de tâche initial de chaque processus est très faible et la plupart du temps est passé dans les communications finales. Ce temps final peut difficilement être réduit. On voit bien qu'il n'est pas nécessaire d'utiliser un grand nombre de processus pour une image trop petite.

Image moyenne : 1280×1024 Cette fois-ci l'efficacité reste toujours au-dessus de 95%, l'image est plus longue à calculer en séquentielle (plus d'une heure). On a donc un speed-up quasi-linéaire, plus de temps est passé dans la partie calcul que dans la partie finale qui n'est pas parallélisable.

Grande image : 3840×2160 Dans ce cas là, l'image étant tellement grande, l'efficacité diminue plus lentement, on a un speed-up linéaire.

2.5 Améliorations possibles

Dans le cadre du projet nous avons fait certains choix et implémentation et nous proposons ici plusieurs possibilitées d'amélioration possibles à notre programme.

Taille des tâches On pourrait changer le ratio auquel multiplier le nombre de sample ou utiliser une fonction qui prend en compte d'autres paramètres (taille de l'image, nombre de processus)

Implémentation du vol Dans notre implémentation actuelle, chaque processus ne communique qu'avec une seule victime ou un seul voleur. Augmenter cela pour le nombre de voleur améliorerait les performances et probablement de même pour le nombre de victime mais compliquerait beaucoup l'implémentation.

Préparer la récupération plus tôt Nous avons pensé à aussi modifier la condition de vol des lignes : nous introduisons un concept de pénalité. Cette pénalité aura pour effet de modifier le moment au cours duquel le processus demande des lignes ainsi que la quantité de ligne volée. La pénalité est calculée en fonction du niveau dans l'arbre.

De cette manière, les processus haut dans l'arbre auront un peu plus de lignes à calculer, ce qui permettra aux processus bas dans l'arbre de finir plus vite et d'envoyer leurs données à leurs parents pendant que leur parent fini ses calculs.

3 Parallélisation OpenMPI + OpenMP

3.1 Sections parallèles

Pour cette partie, nous avons d'abord cherché dans notre algortihme quelles étaient les parties parallélisables et corriger l'aléatoire n'étant pas thread-safe.

Dans l'algorithme du path-tracer, pour calculer la valeur finale d'un pixel, les valeurs de 4 sous-pixels sont calculées et moyennées. Les valeurs de ce 4 sous-pixels sont les moyennes des rayons qui sont considérés comme des samples.

Afin d'améliorer les performances du multithread, nous avons multiplié certaines parties du calcul des pixels sans en changer la véracité. Par exemple, à la fin du calcul d'un sample, on ajoute dans le résultat du sous-pixel la valeur calculée pour ce sample avec la formule $\frac{1}{\text{nombre de samples}} \times \text{valeur}$. Au lieu de faire cela, on ajoute dans des variables temporaires les valeurs calculées dans un tour de boucle, puis après la boucle, on divise par samples. Cela permet donc d'éviter de faire samples divisions, beaucoup plus coteuse qu'une addition.

3.2 Section parallélisable

Sample Nous avons premièrement parallélisé la boucle qui génère les rayons samples.

Sous-pixels Nous avons ensuite parallélisé en donnant un sous-pixel par thread.

Pixels Nous avons aussi parallélisé la boucle en donnant un pixel entier à calculer par thread.

3.3 Changement d'aléa

À chaque besoin de nombre aléatoire, le programme utilise actuellement la fonction erand48(). Cependant, la fonction n'est pas thread-safe car elle modifie des variables globales et les lis en même temps. Il est donc nécessaire de changer la fonction utilisé. Pour cela, nous utilisons la fonction rand_r() qui génère un entier non signé aléatoire entre 0 et une valeur RAND_MAX. Pour obtenir ensuite un nombre entre 0 et 1, on divise le résultat par RAND_MAX.

D'autres options ont été envisagées, comme utiliser la librairie random, mais celle-ci semblait la plus rapide. Etant donnée que l'aléatoire n'est pas utilisé de manière cryptographique, nous sommes résté sur cette méthode.

3.4 Mesures et performances

Pour mesurer les qualités des parallélisations implémentées, nous allons comparer le temps passé dans la fonction comportant le multithread avec la fonction avec un seul thread.

Les communications ne faisant pas partie de notre partie parallélisable, on exécutera le programme que sur deux machines en variant le nombre de noeud et les dimensions et nombre de samples de l'image.

Les parallélisations open MP ont d'abord été testées sur le code séquentielles. Même sur le code séquentiel, nous avons pas obtenu une accélération de 2 ou supérieur avec 2 threads. Nous avons alors cherché la meilleure solutions parmis nos essais.

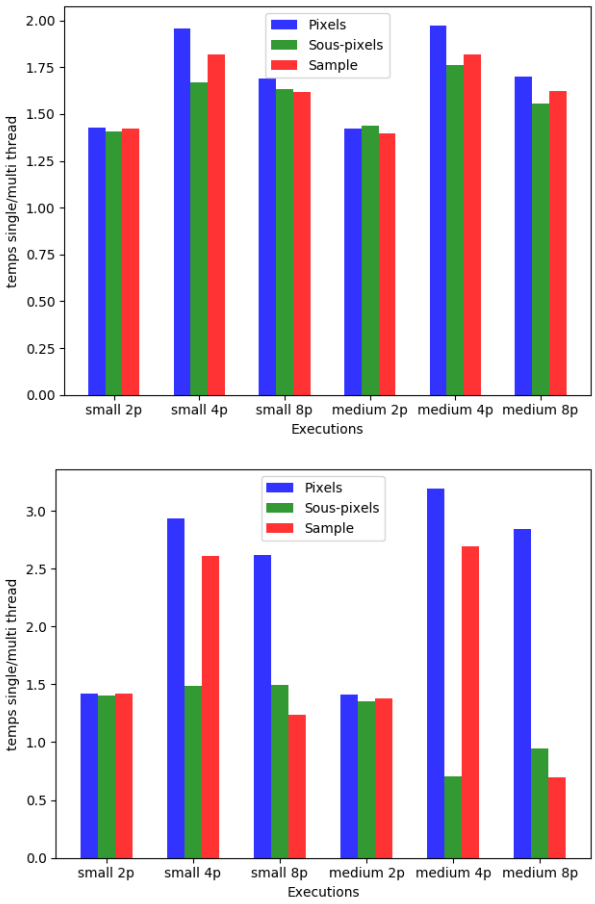


Figure 2: Rapport des temps entre une execution sans thread et une execution avec 2 threads puis avec une execution avec 4 threads pour deux images et different nombres de processus

3.5 Analyse

Remarques Comme on peut l'observer, le temps multithread peut être très différent d'une execution à une autre. Nous avons eu quelques remarques suite à nos mesures :

- Dans aucun des test nous n'avons eu une amélioration pour 2 processus en passant de 2 à 4 threads, alors qu'on peut en observer une dans les mêmes conditions pour 4 et 8 processus.
- Comme dit précédemment, nous n'avons pas d'accélération linéaire en passant en multithreads. Nous pensons que cela est dû à la création de thread qui peut être lent sur les machines.
- Les temps de calcul pour la partie sans thread augmente quand on augmente le nombre de thread alors que cela ne devrait pas l'affecter. Cela est probablement dû au système qui parallélise la destruction de thread.
- Étant donné que nous avons effectué nos calculs sur la fin du projet, beaucoup d'autres d'autres les faisaient aussi en mme temps, ce qui peut fausser plusieurs calculs et exécutions.

Sample Le problème de cette méthode est qu'elle dépend beaucoup du nombre de sample et avec les paramètres actuels un sample est très rapide à calculé. Il n'est pas rentable d'en faire une tâche, et répartir un thread par itération est moins performant que la parallélisation sur les pixels.

Sous-pixels On a cette fois-ci un problème de concurrence lorsqu'on veut le faire en tâche qui ralenti l'execution et avec un for, l'amélioration est moins importante que les pixels. Une optimisation est surement possible mais il faudrait alors changer certaines variables et demanderait beaucoup de test.

Pixels Comme notre algorithme créer des tâches avec un nombre variable de pixel, pour les test que nous avons fait, c'est la meilleur méthode. Dans le cas d'une grande image avec plusieurs threads, on pourra augmenter la taille des tâches.

4 Parallélisation SIMD

4.1 Via OpenMP

Nous avons d'abord cherché quelles étaient les parties que l'on pouvait vectoriser sans modifier sensiblement le code initial, part mettre des tableaux en aligné et déclarer des fonctions en SIMD. Malheureusement, le code de calcul de base n'est pas vraiment "vectorisable". Nous avons quand même tenté d'utiliser des pragma omp simd (for), et les résultats ne sont pas significatifs, on ne gagne que quelques secondes. Sur les trois parties identifiées, le SIMD via OpenMP ne change que très peu les résultats. Pour cette parallélisation, nous avons également testé sur les différentes parties identifiées précédemment.

4.2 Via les fonctions intrinsèques

Nous avons ensuite étudié l'utilisation de fonction intrinsèques. On peut voir qu'il n'est pas du tout optimal d'utiliser les fonctions intrinsèques sur plusieurs samples en même temps, un sample rentre un nombre aléatoire de fois dans la fonction radiance. La seule option est donc d'optimiser les calcul en mettant les coordonnées dans un avx soit 3 doubles. On pourrait alors accélérer alors 3 fois. Cependant, les calculs n'étant pas toujours simples, on doit utiliser beaucoup de "techniques" pour les faire en avx (permutation d'avx).

Dans plusieurs des fonctions que l'on doit modifier en avx, on se rend compte qu'il faut faire des opérations en plus, notamment pour le produit en croix ou autres. Cela entraîne aussi une modification conséquente du code. Nous avons modifié tout en avx dans la boucle sample sauf la fonction radiance qui castera les avx en double et inversement. Cela créer beaucoup de chargement que l'on pourrait modifier pour améliorer. Il y a aussi le fait que le code repose beaucoup sur des modifications de variable grâce aux pointeurs, ce qui ne correspond pas à une utilisation des fonctions intrinsèques.

A Test de performance OpenMPI

Pour les calculs des Acceleration (Speedup) et Efficacite (Efficiency), nous avons utilise les formules suivantes :
Soit p le nombre de processus, $T_k(n)$ le temps de d'execution de resolution du probleme de taille n avec k processus, $S(n,p)$ l'acceleration et $E(n,p)$ l'efficacite.

$$S(n,p) = \frac{T_1(n)}{T_p(n)}, E(n,p) = \frac{S(n,p)}{p}$$

Tous nos tests ont ete effectue sur les machines de la salle 14-406 de la ppti. Les temps actuels ont été mesuré en additionnant les temps de calcul.

A.1 Petite image

A.1.1 200 samples

processus	Temps (en s)	Acceleration	Efficacite
1	177.90	$\frac{177.87}{177.90} = 1.00$	$\frac{1.00}{1} = 100\%$
2	88.89	$\frac{177.03}{88.89} = 1.99$	$\frac{1.99}{2} = 100\%$
4	44.92	$\frac{177.26}{44.92} = 3.95$	$\frac{3.95}{4} = 99\%$
8	22.82	$\frac{177.07}{22.82} = 7.76$	$\frac{7.76}{8} = 97\%$
16	11.66	$\frac{177.40}{11.66} = 15.21$	$\frac{15.21}{16} = 95\%$
32	6.46	$\frac{181.16}{6.46} = 28.04$	$\frac{28.04}{32} = 88\%$
64	4.14	$\frac{185.70}{4.14} = 44.86$	$\frac{44.86}{64} = 70\%$

A.1.2 400 samples

processus	Temps (en s)	Acceleration	Efficacite
1	360.20	$\frac{360.17}{360.20} = 1.00$	$\frac{1.00}{1} = 100\%$
2	179.94	$\frac{355.05}{179.94} = 1.97$	$\frac{1.97}{2} = 98\%$
4	90.64	$\frac{354.49}{90.64} = 3.91$	$\frac{3.91}{4} = 98\%$
8	45.55	$\frac{354.20}{45.55} = 7.78$	$\frac{7.78}{8} = 97\%$
16	23.07	$\frac{354.41}{23.07} = 15.36$	$\frac{15.36}{16} = 96\%$
32	12.34	$\frac{360.75}{12.34} = 29.23$	$\frac{29.23}{32} = 91\%$
64	7.00	$\frac{369.39}{7.00} = 52.77$	$\frac{52.77}{64} = 82\%$

A.1.3 600 samples

processus	Temps (en s)	Acceleration	Efficacite
1	537.23	$\frac{537.20}{537.23} = 1.00$	$\frac{1.00}{1} = 100\%$
2	269.75	$\frac{535.65}{269.75} = 1.99$	$\frac{1.99}{2} = 100\%$
4	133.61	$\frac{531.78}{133.61} = 3.98$	$\frac{3.98}{4} = 100\%$
8	67.36	$\frac{531.62}{67.36} = 7.89$	$\frac{7.89}{8} = 99\%$
16	34.15	$\frac{531.78}{34.15} = 15.57$	$\frac{15.57}{16} = 97\%$
32	17.50	$\frac{541.33}{17.50} = 30.93$	$\frac{30.93}{32} = 97\%$
64	9.78	$\frac{553.80}{9.78} = 56.63$	$\frac{56.63}{64} = 88\%$

A.2 Moyenne image

A.2.1 400 samples

processus	Temps (en s)	Acceleration	Efficacite
1	10282.23	$\frac{10281.55}{10282.23} = 1.00$	$\frac{1.00}{1} = 100\%$
2	4496.40	$\frac{8955.99}{4496.40} = 1.99$	$\frac{1.99}{2} = 100\%$
4	2002.48	$\frac{7971.99}{2002.48} = 3.98$	$\frac{3.98}{4} = 100\%$
8	1002.56	$\frac{7971.26}{1002.56} = 7.95$	$\frac{7.95}{8} = 99\%$
16	502.05	$\frac{7955.12}{502.05} = 15.85$	$\frac{15.85}{16} = 99\%$
32	259.70	$\frac{8061.56}{259.70} = 31.04$	$\frac{31.04}{32} = 97\%$
64	138.22	$\frac{8328.34}{138.22} = 60.25$	$\frac{60.25}{64} = 94\%$

A.2.2 600 samples

processus	Temps (en s)	Acceleration	Efficacite
8	1516.89	$\frac{12100.28}{1516.89} = 7.98$	$\frac{7.98}{8} = 100\%$
16	759.57	$\frac{12092.56}{759.57} = 15.92$	$\frac{15.92}{16} = 100\%$
32	390.87	$\frac{12242.99}{390.87} = 31.32$	$\frac{31.32}{32} = 98\%$
64	222.28	$\frac{13697.52}{222.28} = 61.62$	$\frac{61.62}{64} = 96\%$

A.3 Grande image

processus	Temps (en s)	Acceleration	Efficacite
32	17352.68	$\frac{554608.03}{17352.68} = 31.96$	$\frac{31.96}{32} = 100\%$
64	8936.92	$\frac{568448.44}{8936.92} = 63.61$	$\frac{63.61}{64} = 99\%$

B Test de performance OpenMPI + OpenMP

Les calculs des rapports sont pondérés par le nombre de tâche terminée que nous n'affichons pas dans ces tableaux mais qui sont disponible dans le dossier temps.

B.1 Sample

B.1.1 Small image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	846.66	596.56	$\frac{846.66}{596.56} = 1.42$
2	4	846.63	596.72	$\frac{846.63}{596.72} = 1.42$
4	2	861.15	474.06	$\frac{861.15}{474.06} = 1.82$
4	4	881.40	337.46	$\frac{881.40}{337.46} = 2.61$
8	2	970.17	600.77	$\frac{970.17}{600.77} = 1.61$
8	4	1093.48	889.78	$\frac{1093.48}{889.78} = 1.23$

B.1.2 Medium image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	1880.24	1348.96	$\frac{1880.24}{1348.96} = 1.39$
2	4	1885.66	1367.35	$\frac{1885.66}{1367.35} = 1.38$
4	2	1927.77	1060.36	$\frac{1927.77}{1060.36} = 1.82$
4	4	1986.12	738.04	$\frac{1986.12}{738.04} = 2.69$
8	2	2180.95	1344.71	$\frac{2180.95}{1344.71} = 1.62$
8	4	2485.71	3580.70	$\frac{2485.71}{3580.70} = 0.69$

B.2 Sous-pixels

B.2.1 Small image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	896.46	637.15	$\frac{896.46}{637.15} = 1.41$
2	4	851.30	606.50	$\frac{851.30}{606.50} = 1.40$
4	2	1007.31	605.14	$\frac{1007.31}{605.14} = 1.66$
4	4	1266.19	854.71	$\frac{1266.19}{854.71} = 1.48$
8	2	1052.16	645.38	$\frac{1052.16}{645.38} = 1.63$
8	4	1255.64	845.01	$\frac{1255.64}{845.01} = 1.49$

B.2.2 Medium image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	2078.70	1447.78	$\frac{2078.70}{1447.78} = 1.44$
2	4	1986.92	1469.30	$\frac{1986.92}{1469.30} = 1.35$
4	2	2002.06	1137.44	$\frac{2002.06}{1137.44} = 1.76$
4	4	3673.02	5242.87	$\frac{3673.02}{5242.87} = 0.70$
8	2	2162.99	1392.78	$\frac{2162.99}{1392.78} = 1.55$
8	4	2491.76	2633.93	$\frac{2491.76}{2633.93} = 0.95$

B.3 Pixels

B.3.1 Small image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	845.79	593.40	$\frac{845.79}{593.40} = 1.43$
2	4	844.33	594.07	$\frac{844.33}{594.07} = 1.42$
4	2	857.59	439.71	$\frac{857.59}{439.71} = 1.95$
4	4	877.48	298.80	$\frac{877.48}{298.80} = 2.94$
8	2	967.96	574.03	$\frac{967.96}{574.03} = 1.69$
8	4	1056.97	405.75	$\frac{1056.97}{405.75} = 2.60$

B.3.2 Medium image

processus	thread	temps single thread	temps multi thread	Rapport
2	2	1887.11	1326.92	$\frac{1887.11}{1326.92} = 1.42$
2	4	1878.08	1332.27	$\frac{1878.08}{1332.27} = 1.41$
4	2	1923.49	975.08	$\frac{1923.49}{975.08} = 1.97$
4	4	1973.38	619.04	$\frac{1973.38}{619.04} = 3.19$
8	2	2157.21	1271.22	$\frac{2157.21}{1271.22} = 1.70$
8	4	2388.96	840.17	$\frac{2388.96}{840.17} = 2.84$