Menaa Aness
Bouissonnié Florian

UNIVERSITÉ PARIS 8

Programmation temps réel
Paris 8
M1 informatique

Compilation et spécification :

Tous les tests on été réalisés sur un ordinateur ayant un Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz 4 coeurs et un système ubuntu 64 bits.
Pour compiler le projet un makefile a été réalisés, pour s'en servir il faut faire soit make bench pour créer le l'exécutable du benchmark ou bien make interact pour créer l'exécutable de l'interaction.
Pour exécuter les deux exécutables suivent le même modèle ./nomexecutable nombrethread nombreechantillon.

Partie 1 : Banc de test :

Methode de parallélisation :

La méthode de parallélisation que nous avons utilisée est la méthode de découpage de l'image en un certain nombre d'échantillon. Chaque thread du programme va donc s'occuper d'un échantillon, lorsqu'un thread finit de traiter un échantillon il va passer au prochain à traiter.

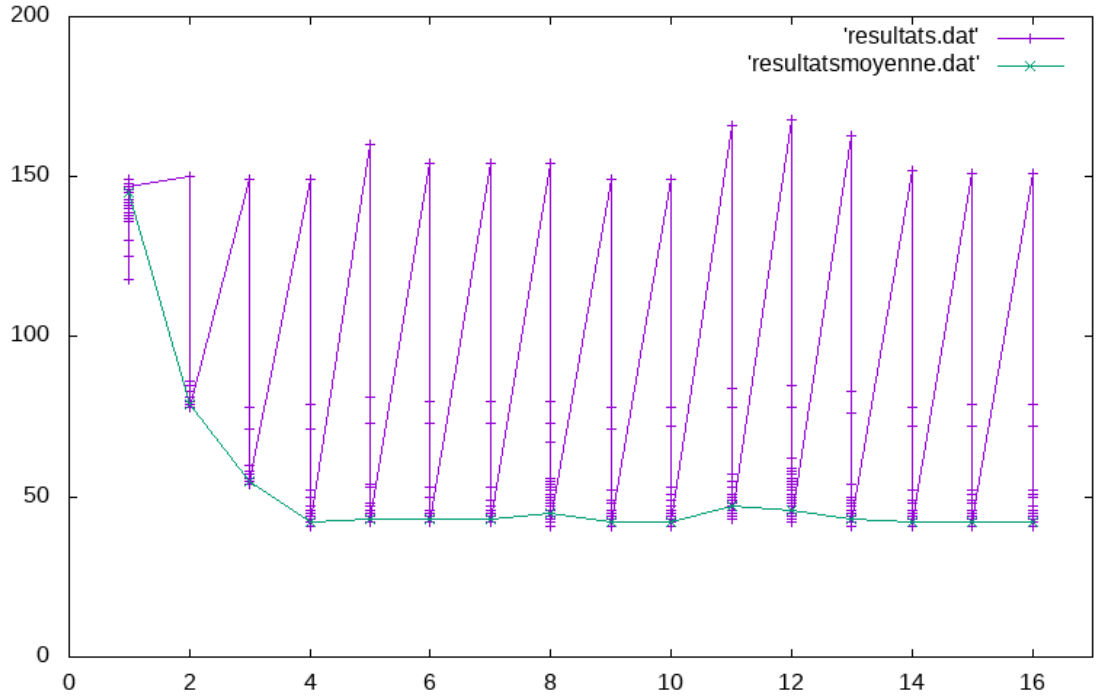
Pour l'utilisation du programme, on l'exécute avec deux paramètres : en premier le nombre de threads, avec lequel on veut que notre programme travaille, ainsi que le nombre d'échantillons, c'est à dire en combien de morceaux on va séparer le traitement de notre image.

Pour pouvoir faire notre banc de test on utilise un script shell qui va pour chaque nombre de threads de 1 a 16 et pour chaque nombre d'échantillons de 1 a 256, exécuter notre programme avec ces paramètres pour un nombre d'exécution totale de 16 * 256 exécutions et chaque exécution du programme calcule toutes les fractales avec chaque nombre imaginaires dans l'intervalle -1,1 avec des pas de 0.1. Tous les calculs de temps sont stockés dans un fichier resultats.dat et la moyenne des temps pour chaque thread est stockée dans un fichier resultatsmoyenne.dat.

Les fichiers resultats.dat et resultatsmoyenne.dat sont ensuite utilisés avec gnuplot afin de générer une courbe pour représenter nos résultats.

Nous avons choisi de faire nos tests sur 16 threads car nous disposons d'une machine avec un processeurs 4 coeurs.

Voici les résultats que nous avons obtenu :



En X nous avons le nombre de thread et en Y le temps d'exécution.
On peut voir que les meilleurs résultats se situent autour de 4 threads et que, plus tard à partir de 8 threads, on perd du temps. On peut penser que cette augmentation du temps d'exécution est due au temps que prend la création de thread.

Partie 2 : Interaction utilisateur :

Methode choisie :

Pour synchroniser les threads avec le processus principal, pour nous assurer de recalculer la fractale uniquement quand nécessaire tout en assurant une réactivité maximale, nous avons utiliser une variable conditionnelle pthread_cond_t.
Elle permettra de mettre en sommeil les threads lorsque le calcule de la fractale courante est terminé et d'éviter la consommation des ressources du processeur inutilement.
Ainsi , grâce à la fonction cvwaitkey() placée dans une boucle nous allons pouvoir attendre l'action de l'utilisateur sur le clavier qui permettra de recalculer une nouvelle fractale avec des valeurs différentes selon la touche appuyée.
De plus nous appelons la fonction d'affichage imshow() à chaque tour de boucle, ce qui nous permet de voir le calcul de la fractale en temps réel et de façon instantanée.

Voici les commandes disponibles :
z : permet de zoomer sur la fractale
d : permet de dézoomer sur la fractale
o : permet de décaler la fenêtre vers le haut
k : permet de décaler la fenêtre vers la gauche
m : permet de décaler la fenêtre vers la droite
l : permet de décaler la fenêtre vers le bas
c : permet de changer les couleurs de la fractale
n : permet de repasser en noir et blanc
s : permet de sauvegarder la fractale

r : permet de changer la partie réelle de la fractale
i : permet de changer la partie imaginaire de la fractale
q : permet de quitter le programme

Pour la colorisation de la fractale nous nous contentons de générer trois valeurs aléatoires c1, c2 et c3 avec lesquelles nous appelons la fonction color(c1, c2, c3) dans notre fonction julia.