LSINF1252 - Groupe 60

Systèmes Informatiques

Travail de groupe : Création de fractales

Rapport 2 - Implémentation

Auteurs :
Julien Calbert - 33211500
Pierre Lamotte - 65441500

10 mai 2018

Table des matières

1 Architecture

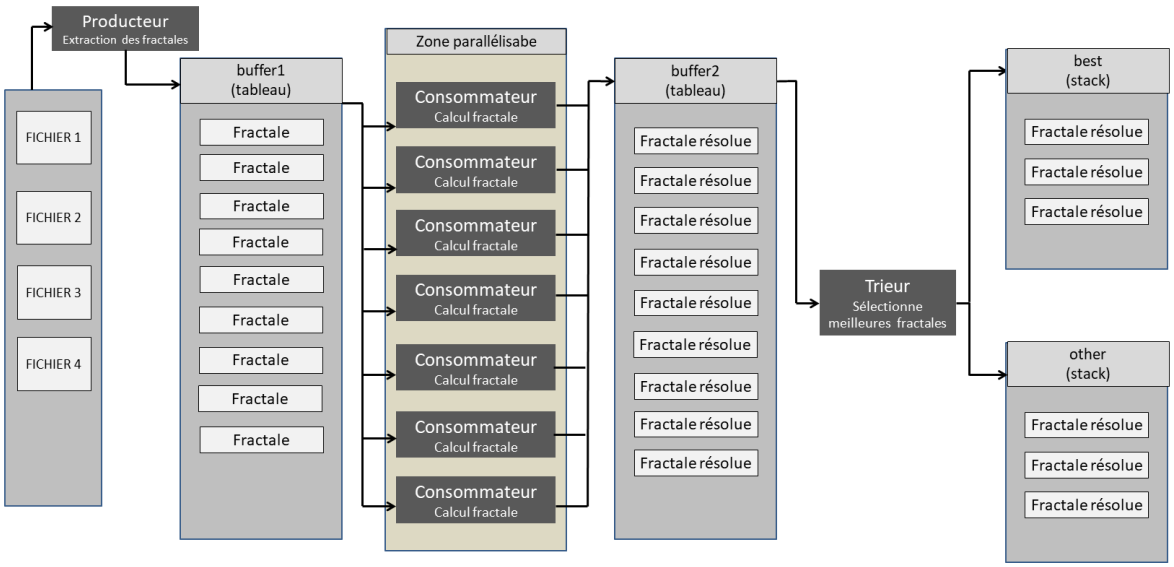

FIGURE 1 – Structure globale du programme

1.1 Types de threads et leurs implémentations

— Producteur : nous avons décidé de faire 1 producteur s'occupant de tous les fichiers, Il saute les lignes vides et lignes commencant par : '#' correspondant à un commentaire. Toutefois, mise à part ces 2 cas, notre programme suppose que le format d'entrée est tel que spécifié dans l'énoncé.

— Consommateurs : ce sont les threads qui vont travailler dans la zone parallélisable, ils prennent un pointeur vers une structure fractale se trouvant dans le buffer1 rempli par le producteur et calculent la fractale. Une fois la fractale complètement calculée, un pointeur vers cette dernière est ajouté dans le buffer2 (tableau). Le thread peut alors répéter l'opération sur une autre structure du buffer1.

— Trieur : ce thread est unique. Il maintient dans un troisème buffer (implémenté sous la forme d'une stack) la ou les meilleurs fractales.

  — Si [-d] n'est pas spécifié : le trieur extrait une fractale du buffer2, si celle-ci est de valeur moyenne plus faible que l'actuelle meilleure, il la free(), si elle est égale, il l'ajoute dans best(stack) et si elle est meilleure, on dépile la stack, libérant toutes les structures ainsi dépilées et la place dans la stack (dans ce cas, other n'est pas utilisé).

  — Si [-d] est spécifié : le comportement est semblable au cas précédent exepté le fait qu'on ne free() par les fractales de valeurs moyennes plus faible, mais on les push dans other. Ainsi, à la fin de l'exécution, on se retrouve avec toutes les fractales de valeur la plus élevée dans best et les autres dans other.

2 Caractéristique de notre implémentation

— Avant toute chose, nous tenons à préciser que nous n'avons pas implémenté la lecture sur l'entrée standard. Ayant repoussé cette impléméntation pour la finalisation du programme, nous avons manqué de temps pour coder cette partie de sorte qu'elle ne présente aucun souci de compilation ou de fonctionnement du programme.

— L'avantage d'utiliser des tableaux par rapport à des listes chainées pour les buffers 1 et 2 est leur taille fixe. En effet, étant donné que le buffer1 ne peut accepter qu'un nombre limité de fractales, le producteur ne pourra

donc pas générer un nombre trop important de fractales et risquer de saturer la mémoire en pré-allouant trop de structures.

— Nous n'utilisons qu'un seul producteur et un seul trieur car nous considérons que leurs vitesses d'exécution est bien supérieure à celle des consommateurs.

— L'information communiquée entre les différents threads sont des pointeurs vers des structures fractales. Nous avons choisi d'implémenter best et other sous la forme d'une stack et non sous la forme d'un tableau car nous ne savons pas à l'avance le nombre de fractales dont la valeur moyenne est la plus élevée.

— De plus, notre programme optimise aussi la gestion de l'espace mémoire. En effet, notre programme free() la mémoire (si [-d] est non spécifié) au fur et à mesure de son exécution.

— Afin d'implémenter la consigne suivante : chaque fractale doit avoir un nom unique dans les fichiers. Si plusieurs fractales ont le même nom, seule la première doit être considérée et les autres doivent être négligées. Pour cela, on maintient une structure chainée contenant les noms distincts des fractales déjà en cours de calcul. Ainsi, quand le producteur lit sur le fichier une fractale à calculer, il parcourt la liste chainée pour vérifier si ce nom n'est pas déjà présent, si ce n'est pas le cas, elle l'ajoute, sinon elle est négligée.

— Nous avons également implémenté une solution au problème du rendez-vous pour les consommateurs afin de correctement gérer la terminaison de ceux-ci et faire en sorte qu'ils soient bloqués lorsqu'il n'y a plus de fractales à calculer, avant d'être libérés en chaîne par le dernier consommateur.

3 Evaluation

Nous avons effectué de très nombreux tests avec Valgrind pour controler les fuites de mémoire. Nous n'avons jamais détecter d'erreurs.
Nous avons évalué le temps réel de calcul de notre programme pour la configuration suivante :

— les buffers 1 et 2 sont de tailles 10.

— il y a 3 fichiers, contenant un total de 30 fractales de tailles variables. Le détail de ces fractales est en annexes.

— -d, non spécifié.

Nous avons progressivement augmenté le nombre de thread en maintenant inchangé les fichiers d'entrées. Nous avons effectué cette analyse sur une machine à 4 coeurs dont voici un graphique de performances.

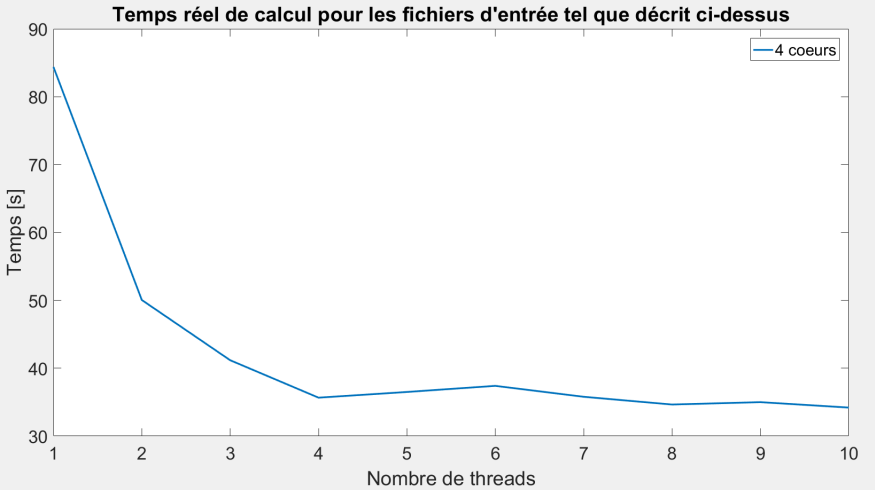

FIGURE 2 –

Premièrement, nous constatons bien une diminution du temps de calcul avec l'augmentation du nombre de threads (consommateurs). Cet amélioration des performances du pro- gramme est permise grâce à l'agrandissement de la zone parallélisable. De plus, les résultats sont cohérents, en effet, on vérifie que la courbe bleue ne descend plus à partir de 4 threads étant donné qu'il s'agit d'un processeur 4 cœurs.

4 Annexe

Voici le détail des fractales calculées lors du test d'évaluation.

Liste des fractales			
nombre	dimensions	partie réelle	partie imaginaire
7	1920×1080	-0.52	0.57
15	1920×1080	1	-1
1	500×300	-0.52	0.57
4	800×800	-0.8	0.4
1	400×400	-0.8	0
1	400×400	-0.8	0.7
1	400×400	0.5	0.5

Il y a donc un total de 30 fractales et $48.8092 \cdot 10^6$ pixels.
Remarque, pour un même nombre de pixels, certaines fractales nécessitent plus de temps de calcul (par exemple : la fractale (-0.52 ;0.57) est plus longue à calculer que (1 ;-1)).