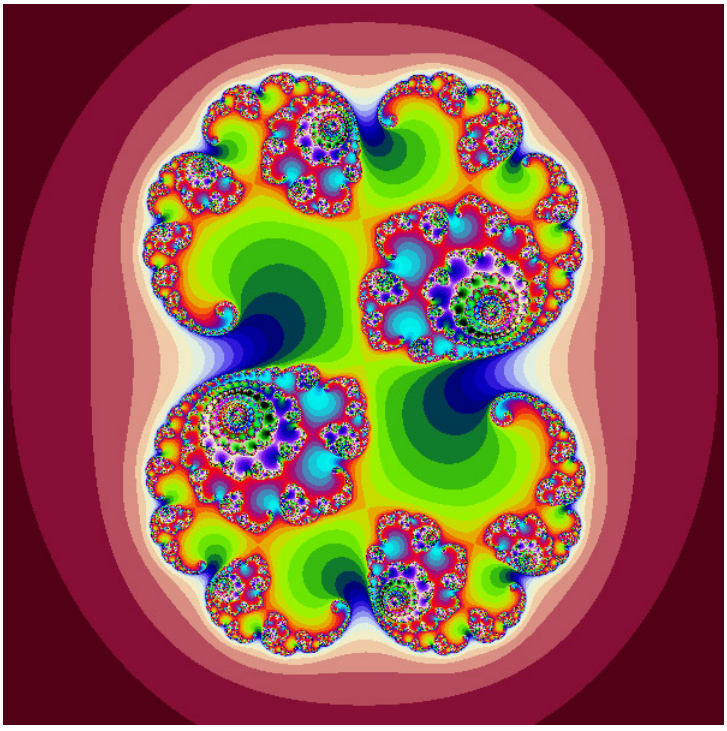
LSINF1252 : Systèmes informatiques
Rpport du projet "Fractales"

De Keersmaeker	François	7367-16-00
Gérard	Margaux	7659-16-00



Louvain-la-Neuve
11 mai 2018

1 Structure

Dans cette partie, nous expliquerons les différentes caractéristiques de notre programme. Nous avons pris la décision d'implémenter notre code sur base du principe du producteur et du consommateur. Le producteur s'occupe du fichier d'entrée qu'il va lire ligne par ligne grâce à des threads de lecture. Chaque ligne va être splitée pour ensuite être transformée en arguments pour créer une `struct fractal`. Cette fractale va être stockée dans un buffer dont la taille est connue a priori et est égale à `maxthreads`. Pour éviter que plusieurs threads placent deux fractales simultanément dans une même case du buffer, nous protégeons leur activité par un mutex. En ce qui concerne le remplissage du buffer, ce sont les sémaphores qui vont le gérer. Par exemple, elles empêcheront aux threads de lecture d'ajouter une fractale dans le buffer si celui-ci est déjà rempli.

Le consommateur va, lui aussi, être géré par des sémaphores et des mutex. Son rôle est de récupérer les fractales que le producteur aura placées dans le buffer, une par une, et d'effectuer un calcul sur celles-ci. Plus précisément, il s'agira de stocker la ou les fractales dont la moyenne des valeurs de chaque point est la plus élevée. Pour effectuer le calcul, nous avons instancié une variable globale `max` qui contiendra la valeur maximale de la moyenne de la meilleure fractale sur chaque pixel. A chaque itération du processus, la moyenne de la fractale sur chaque pixel sera comparée à la variable globale.

Après avoir calculé cette moyenne et l'avoir comparée à la variable globale pour chaque fractale, le consommateur va renvoyer la fractale donc la valeur moyenne sur chaque pixel est maximale. Si il y a plusieurs fractales ayant une valeur moyenne maximale, nous devons les stocker. Pour des raisons que nous mentionnerons dans la partie suivante de ce rapport, nous stockerons ces fractales dans une liste chaînée, une pile. Par la suite, il suffira au consommateur d'afficher les fractales contenues dans la pile grâce à la fonction `write_bitmap_sdl` qui nous est donnée dans l'énoncé. Après chaque affichage d'un noeud de la pile, ce noeud est supprimé et la mémoire est libérée.

2 Choix de conception

Dans le cadre de ce projet, nous avons effectué plusieurs choix de conception qu'il est important de mentionner. Tout d'abord, pour la partie du code qui gère le calcul de la fractale dont la moyenne des valeurs de chaque point est la plus élevée, nous avons décidé d'utiliser une liste chaînée et plus précisément, une pile. En effet, utiliser une pile plutôt qu'un tableau nous permet de pouvoir stocker plusieurs fractales qui auraient une valeur en chaque point la plus élevée sans pour autant réserver une zone mémoire inutile. Ensuite, en ce qui concerne le buffer qui est rempli par le producteur, il nous semblait légitime de créer un tableau et non une liste chaînée puisque nous connaissons le nombre de threads de calcul maximal, qui sera aussi la taille de notre tableau.

Abordons maintenant les tests que nous avons effectué. Nous avons testé chacune de nos fonctions de manière indépendante pour s'assurer qu'elles fonctionnaient correctement. A priori l'ensemble des fonctions exécutent le code correctement et retourne ce que nous souhaitons. Nous avons également écrit des tests CUnit mais nous n'avons malheureusement pas eu le temps de tous les tester.

3 Evaluation qualitative et quantitative

Dans cette partie, nous évoquerons les différents points positifs mais aussi négatifs que contient notre code. Pour commencer, notre code affiche la solution que nous devions obtenir : des fractales. Cependant, notre code n'est pas parfait. Nous n'avons par exemple pas pas implémenté le fait que deux fractales ne peuvent pas avoir le même nom. Une autre faille est que notre code fonctionne uniquement pour une seule fractale. Nous avons également eu plusieurs problèmes lors de l'implémentation de ce programme, notamment beaucoup de *segmentation fault*. Heureusement, nous avons réussi à les déceler et à résoudre la plupart par la suite.

Donnons tout de suite un exemple d'un problème que nous avons rencontré et qui nous a donné du fil à retordre. Nous avions imposé une condition sur les threads de calcul au sein de la fonction `void* consommateur (int* d)`. Cette condition était dans une boucle while et elle permettait d'arrêter l'activité lorsque le buffer dans lequel le thread va chercher la fractale à calculer était vide. Le problème apparaîssait lorsque la condition n'etait plus vérifiée. Le thread de calcul ne devait plus entrer dans la boucle et pourtant, la condition restait bonne jusqu'à ce que ce thread soit entré dans la boucle après quoi elle devenait mauvaise. Malheureusement, le thread étant déjà entré dans la boucle, il provoquait un deadlock et faisait planter le programme. Pour régler ce problème, nous avons testé l'ensemble de la fonction et nous nous sommes rendus compte que notre condition n'était pas tout à fait correcte. En effet, il ne faut pas imposer une seule mais bien deux conditions sur les threads de calcul. Nous avons donc défini une variable globale `lecture` qui donnent le nombre de threads de lecture ouverts et chaque thread de calcul va, avant de calculer la valeur d'une fractale, vérifier deux conditions qui sont les suivantes :

- Il y a des threads de lecture ouverts
- Le buffer n'est pas vide

Si les deux conditions ne sont pas respectées, il faut arrêter le calcul et empêcher le thread d'enter dans la boucle.

Pour terminer, nous pensons être parvenus à coder des fonctions censées qui s'exécutent correctement et notre raisonnement nous semble pertinent. Etant donné le manque de temps, nous n'avons implémenté qu'une grande partie de ce qui était demandé dans l'énoncé et il reste encore des problèmes dans notre code mais avec plus de temps, nous aurions aimé améliorer notre programme. Nous aurions, par exemple, pu faire en sorte que la fonction principale prennent en compte plus d'arguments et que notre programme soit plus efficace.