# Laboratoire de programmation concurrente

## Calcul de la figure de Mandelbrot

### Cyril Ruedin et Alexandre Serex

### Inf2 dlm-a

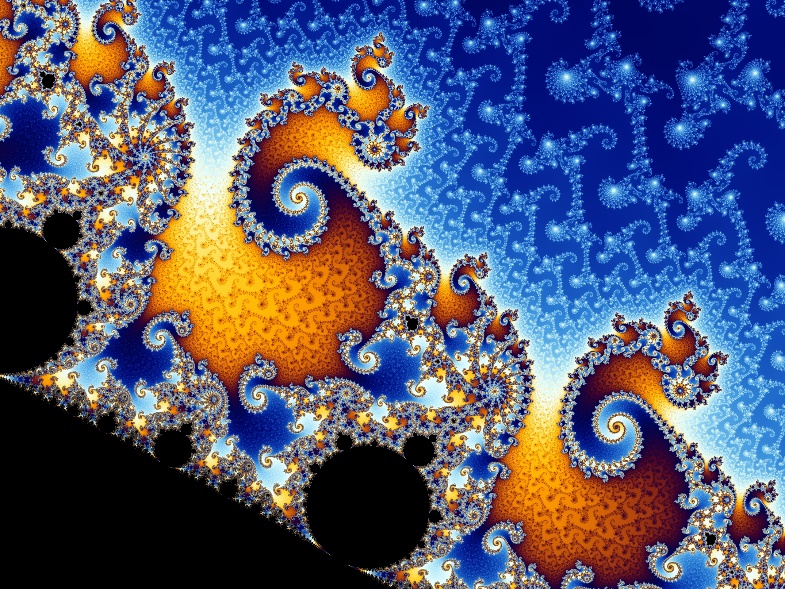


Table des matières

[Introduction 3](#_Toc439855814)

[But et contraintes 3](#_Toc439855815)

[Spécifications matérielles et logicielles 3](#_Toc439855816)

[Structure du programme 3](#_Toc439855817)

[Analyse des tests réalisés 4](#_Toc439855818)

[Premier cas : 1 thread, 1 bloc 4](#_Toc439855819)

[Second cas : 8 threads, 8 blocs 4](#_Toc439855820)

[Troisième cas : 8 threads, 256 blocs 5](#_Toc439855821)

[Dernier cas : 128 threads, 512 blocs 5](#_Toc439855822)

[Conclusion 5](#_Toc439855823)

## Introduction

L’ensemble de Mandelbrot (du mathématicien Benoît Mandelbrot, 1924 – 2010) est un ensemble fractal dans le plan complexe. Il représente chaque point du plan pour lesquels une suite de nombre complexe définie par récurrence converge (ou diverge). En fonction de la vitesse de convergence, on dessinera les points du plan de couleurs différentes (en noir s’ils divergent).

Chaque point du plan peut demander énormément d’opérations avant de pouvoir lui associer une couleur, en plus du fait que le plan possède plusieurs millions de pixels où, pour chaque pixel, il faut effectuer cette longue suite d’opération.

Afin d’accélérer le processus, nous procéderons à une parallélisation des calculs de convergence du plan complexe. Nous bénéficierons ainsi des architectures multi-cœurs pour calculer plus rapidement la couleur de chaque pixel. Un nombre défini d’opération est fixé afin de savoir si, après autant d’opérations, le point du plan complexe converge ou non, et quelle couleur lui donner.

## But et contraintes

Le but de l’application est de calculer et afficher une partie de l’ensemble de Mandelbrot. Le programme fonctionnera de manière concurrente, en séparant en blocs le calcul de l’image à afficher. Chacun de ces blocs sera calculé par un thread unique, mais un thread seul peut calculer plusieurs blocs. Il ne peut y avoir moins de blocs qu’il n’y a de threads.

L’efficacité du programme variera en fonction de la machine sur laquelle l’application tourne ainsi qu’en fonction du rapport threads – blocs. Ces modalités sont traitées dans le chapitre « Tests réalisés et efficacité ».

## Spécifications matérielles et logicielles

Le programme est réalisé en langage C sur le framework Code Blocks. La librairie graphique utilisée pour l’affichage de l’ensemble de Mandelbrot est la Simple DirectMedia Layer, souvent abrégée SDL. Quant au matériel utilisé, les tests ont été réalisés sur une machine comportant un CPU à 4 cœurs physiques et 8 cœurs logiques. ( !!!! CONFIRMATION !!!!)

## Structure du programme

Le programme suit l’architecture « Boss – worker » selon laquelle un thread Master, instancié par le Main, va lui-même être chargé de créer les threads Slaves qui calculeront et afficheront l’image. Le nombre de threads qui seront créés par le Master est déterminé par l’utilisateur du programme grâce à une simple entrée clavier. De la même manière, l’utilisateur doit spécifier le nombre de blocs de calculs qui seront exécutés par les threads.

On peut schématiser l’architecture de la manière suivante :

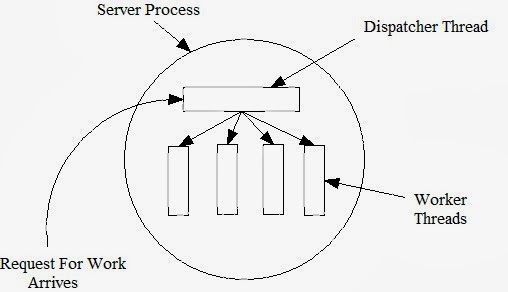


Figure 1 : architecture boss-worker

Le process Main, à l’exécution, s’occupe de créer le thread Master (ci-dessus, « Dispatcher Thread ») qui lui-même séparera le calcul de l’image en blocs. Le Master crée ensuite les threads qui seront chargés de calculer chaque bloc un a un, en concurrence.

L’opération « Request For Work Arrives » sur la figure 1 symbolise les deux entrées que l’utilisateur doit fournir, c’est-à-dire le nombre de threads et de blocs voulus.

## Analyse des tests réalisés

En tout, seuls 4 tests sont importants pour bien comprendre et mesurer l’efficacité du programme. Ces trois tests, détaillés ci-dessous, sont :

* Le cas 1 thread, 1 bloc, situation triviale.
* Le cas 8 threads, 8 blocs, combinaison optimale.
* Le cas 8 threads, 8+ blocs (p. ex. 256 blocs).
* Le cas 8+ threads (p. ex. 128) threads, 8+ blocs (p. ex. 512 blocs).

Ici, on note 8 threads car le CPU qui a servi à faire les tests possède 8 cœurs logiques. Cependant, si elle devait en comporter 12 par exemple, les 8 (et leurs multiples) seraient remplacés par des 12 (et leurs multiples). Pour généraliser, si une machine comporte N cœurs logiques, alors la combinaison optimale est de N cœurs, N blocs.

### Premier cas : 1 thread, 1 bloc

Ce cas-ci est le cas trivial. Il n’existe pas de concurrence, les threads n’apportent rien par rapport à un programme séquentiel classique. La durée d’exécution du programme n’est pas du tout optimale, elle est même assez importante en comparaison du cas multi-threadé. Si on se réfère à l’utilisation du CPU lors de l’exécution du programme, celui-ci n’est utilisé qu’au huitième de sa performance totale, car une seul de ses cœur logique est mis à contribution.

### Second cas : 8 threads, 8 blocs

Ce deuxième cas est le cas optimal en prenant en compte l’architecture de la machine sur laquelle les tests ont été effectuée. De cette manière, la totalité du CPU est mise à contribution et chaque cœur logique s’occupe d’un seul est unique thread. Une fois que chaque thread a fini son travail sans interruption, l’image est affichée au complet.

### Troisième cas : 8 threads, 256 blocs

Dans cette configuration threads – blocs, chaque cœur logique du CPU n’aura à s’occuper que d’un thread à la fois, tout comme dans le cas optimal. Cependant, chaque thread devra calculer plusieurs blocs, en l’occurrence 8. Ainsi, Ainsi quand un thread a fini de calculer un bloc, il devra être assigné au prochain bloc en attente, ce qui demande quelques opérations supplémentaires de la part de la machine.

Cette situation diffère légèrement de la situation optimale car quand chaque thread aura fini sa première tâche, l’image ne sera encore affichée que partiellement. Chaque bloc à afficher demande moins d’opérations que dans la situation optimale vu qu’il couvre une moins grande surface de l’image, et est donc plus rapide. En revanche, le temps total pour afficher l’image entière est plus important qu’avec la combinaison optimale, car il faut compter, en plus du calcul de chaque bloc, le changement de bloc entre celui qui vient d’être terminé et le prochain à calculer et ce, pour chaque thread.

### Dernier cas : 128 threads, 512 blocs

Cette situation ci est loin d’être optimale et ce pour plusieurs raisons. Premièrement, tout comme dans le troisième cas, chaque thread aura plusieurs blocs à calculer. On aura donc un temps supplémentaire provenant du fait que chaque thread ayant fini un bloc demande quelques opérations avant de pouvoir se remettre à travailler sur le bloc suivant.

Deuxièmement, le nombre de threads est plus important que le nombre de cœurs logiques du CPU. Comme chaque cœur ne peut travailler que sur un thread à la fois, le système d’exploitation va devoir interrompre un thread en exécution afin de laisser la place à un autre thread. Il va donc effectuer une commutation de contexte. Plus le nombre de threads est important et plus les commutations seront fréquentes. Chaque commutation demande un certain temps à être complétée, du temps durant lequel aucun bloc n’est en train d’être calculé par ce cœur vu qu’aucun thread n’est en exécution dessus. C’est donc du temps gâché pour permettre au surnombre de thread de pouvoir s’exécuter en semi-parallélisme.

Ainsi, même si le CPU est entièrement mobilisé pour exécuté le programme, toutes les opérations ne serviront pas à dévoiler une partie de l’ensemble de Mandelbrot, contrairement au deuxième cas qui lui est optimal.

## Conclusion

Les tests ont bien démontré l’avantage des architectures multi-cœur en permettant d’accélérer sensiblement le calcul de l’image. Ceci nous permet également de tester les limites de la concurrence sur nos machines actuelles lorsque le nombre de threads / blocs ne correspond pas à la machine sur laquelle tourne le programme.