Elements logiciels pour traitement de données massives-Projet informatique

ENSAE 2021/2022

Mastère Spécialisé - Data Science

Michel Capot - Jeffrey VERDIERE

Parallélisation de l'ensemble de Julia Une étude appliquée à l'ensemble de Mandelbrot

Jeffrey Verdière

Michel Capot

Supervisors :Xavier Dupré Matthieu Durut

Table des matières

Résumé

L'ensemble de Mandelbrot est un ensemble mathématique susceptible de générer des figures fascinantes grâce à la manipulation de nombres complexes. Sa visualisation requiert néanmoins des calculs très coûteux d'un point de vue computationnel. Dans ce projet, nous voulons proposer une implémentation parallèle de cet ensemble grâce au compilateur Cython afin de réduire les temps de calculs observés sur une implémentation naïve dans un double objectif. D'une part, sur le plan théorique, nous souhaitons livrer une analyse comparative détaillée des performances propres à ces deux implémentations ; d'autre part, dans le cadre empirique d'une exploration graphique interactive de l'ensemble de Mandelbrot, nous souhaitons déterminer les paramètres optimaux permettant d'approcher la fluidité d'une vidéo.

1 L'ensemble de Mandelbrot

L'ensemble de Mandelbrot est un ensemble M de nombres complexes c tels que la suite $f_c(z) = z^2 + c$ ne diverge pas lorsqu'on prend pour condition initiale $z = 0$. Pour générer l'ensemble de Mandelbrot, on peut échantilloner le nombre complexe c et itérer la fonction f_c selon le mappage quadratique suivant :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \ \forall n \in \mathbb{N} \end{cases} \tag{1}$$

Tout nombre complexe c appartient à M si et seulement si, pour tout n inférieur ou égal à un nombre maximal d'itérations fixé, la condition suivante est respectée : $| \ z_n =| < 2$.

Chaque point échantilloné peut être représenté sur un plan en 2D en traitant sa partie réelle en abscisse et sa partie imaginaire en ordonnée. A chaque point du plan peut être associée une couleur (voir section suivante) ; dans l'exemple ci-dessous, les points qui appartiennent à l'ensemble sont jaunes tandis que les autres tendent vers le violet.

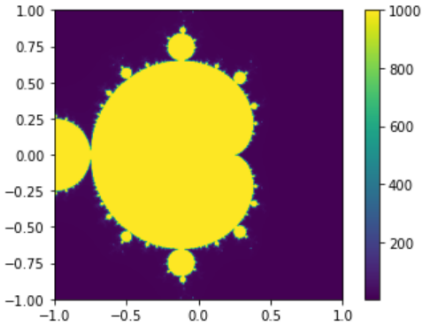

FIGURE 1 – Exemple de représentation graphique de l'ensemble de Mandelbrot

Notre projet est inspiré par les vidéos (disponibles en ligne) montrant de longs zooms hypnotisants sur différentes fractales[1]. Afin de permettre à tout utilisateur de Python d'élaborer sa propre exploration (potentiellement infinie) de l'ensemble de Mandelbrot, nous avons voulu construire une interface graphique permettant de se déplacer et de

1. Par exemple la vidéo *Eye of the Universe* par Maths Town : https ://www.youtube.com/watch ?v=pCpLWbHVNhkt=2283s

zoomer sur le plan à volonté : par souci d'ergonomie, nous avons décidé de rattacher ces commandes à des actions clavier. Ainsi, les flèches directionnelles déplacent la fenêtre visualisée dans les quatre directions, de même que les touches 'd' et 'q' permettent de zoomer et de dézoomer. Afin de créer l'illusion de la vidéo, le temps de calcul nécessaire pour afficher une portion donnée du plan ne devrait pas excéder un vinq-quatrième de seconde.

2 L'implémentation naïve de l'ensemble de Mandelbrot

2.1 Implémentation et pseudo-code

Il existe une multitude d'algorithmes pour dessiner l'ensemble de Mandelbrot. Puisque l'objectif de notre projet est de réduire les temps de calcul, nous optons pour son implémentation la plus simple : [1] ≪ The escape time algorithm ≫.

Algorithm 1: Escape Time algorithm

for *each pixel (Px, Py) on the screen* **do**_

$x_0 \leftarrow$ scaled x coordinate of pixel $y_0 \leftarrow$ scaled y coordinate of pixel $x \leftarrow 0.0$ $y \leftarrow 0.0$ $iteration \leftarrow 0$ **while** $(x \times x + y \times y < 2 \times 2$ *AND* $iteration < max\_iteration)$ **do**

_ $x_{temp} \leftarrow x \times x - y \times y + x_0$ $y \leftarrow 2 \times x \times y + y_0$ $x \leftarrow x_{temp}$ $iteration \leftarrow iteration + 1$

$color \leftarrow palette[iteration]$ $plot(Px, Py, color)$

Où $z = x + iy$, $c = x_0 + iy_0$, $x = Re(z^2 + c)$ et $y = Im(z^2 + c)$.

2.2 Complexité algorithmique de l'implémentation naïve

Dans cette partie, nous tentons d'approcher le coût de calcul de l'implémentation naïve de l'algorithme escape time dont la structure algorithmique est restituée ci-dessus. Pour simplifier le problème, supposons que l'on génère des figures carrées de hauteur et de largeur de N pixels. L'algorithme est alors appelé N^2 fois.

Ce coût théorique peut être majoré en considérant le pire scénario où chaque complexe n'est ni dans le cardoïde ni dans le bulbe de période 2 jusqu'au maximum d'itérations, et où l'image n'est pas centrée sur l'axe des abscisses (pas de symétrie qui permette de diviser les temps de calcul par deux).

Pour calculer ce coût, on considère uniquement les opérations suivantes sur chaque itération :

Table 1 – Coût de chaque opération

Opération	**Augmentation de la complexité par**
Additions, soustraction, comparaison	1
Multiplication, division par 2	4

Plus particulièrement, dans l'algorithme ≪ escape time ≫, on a :

TABLE 2 – Répartion des tâches de l'algorithme

Opération	**Tâches**
Affecter une valeur à c	3 multiplications, 3 additions, 2 soustractions
Dans chaque itération	7 multiplications, 3 additions, 1 soustraction

Par conséquent, le coût du traitement d'une image $N * N$ est donné par la formule suivante :

$$Coût = (17 + 32 * max_iteration) * N^2 \tag{2}$$

Le coût du traitement est donc en $O(N^2)$ pour une valeur fixée de maxiteration.

Remaque : Il faut noter que le calcul de coût de l'équation (2) ne prend pas en compte le coût de la détermination de la couleur de chaque pixel et du stockage de la matrice contenant les valeurs de temps d'échappement comme une image. Ainsi, $Coût(stockage - image) = o(N^2)$ et $Coût(couleur - pixel) = o(N^2)$. Pour finir, si on fixe la taille d'une image, le coût de calcul augmente linéairement en fonction du nombre maximal d'itérations. Pour illustrer, ces différentes configurations, on trace les figures suivantes :

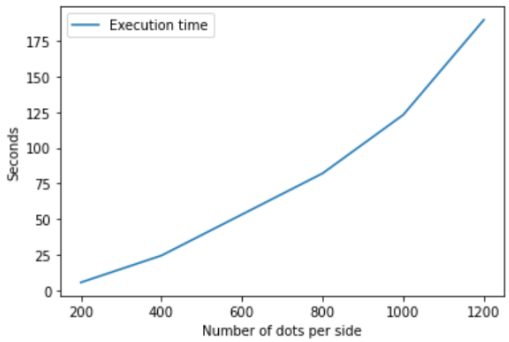

FIGURE 2 – Temps d'exécution en fonction du nombre de pixels

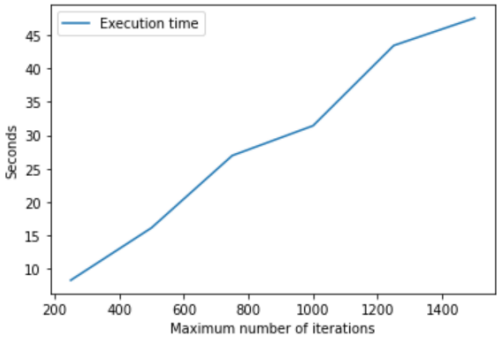

FIGURE 3 – Temps d'exécution en fonction du nombre maximal d'itérations

On constate donc bien que le temps d'execution de l'algorithme est quadratique du nombre de pixels et linéaire du nombre d'itérations.

2.3 Du calcul du speedup théorique à une implémentation parallélisée

On souhaite étudier la possibilité d'améliorer notre code naïf en parallélisant notre problème sur plusieurs coeurs. La loi d'Amdhal permet de calculer un speed up maximum qui peut être espéré en parallélisant notre code naïf. Cette loi est exprimée par la formule suivante :

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} = \frac{T_1}{T_p} \tag{3}$$

où α est la fraction de code non-parallélisable, p le nombre de processeurs, T_1 le temps d'execution sur 1 processeur, T_p le temps d'execution sur p processeurs.

Dans notre configuration, avec une résolution de 500*500 points, une norme maximale de 16 et un nombre d'itérations limité à 1000, nous obtenons un temps d'exécution total de 32.57 secondes, dont 32.50 secondes correspondant à la portion parallélisable du code, soit une valeur d'alpha de 0.002 pour un facteur de speed-up théorique de 3.97.

3 L'implémentation parallélisée de l'ensemble de Mandelbrot

Pour l'implémentation parallélisée de notre code, nous reprenons la même stratégie fondée sur la méthode "escape-time". Nous implémentons une nouvelle fois cet algorithme en Cython grâce à la syntaxe cdef et l'instruction "nogil", qui permet de relâcher le Global Interpret Lock afin d'autoriser les calculs multi-threads lors de l'appel de cette fonction. Nous réécrivons ensuite la fonction "mandelbrot", dans laquelle la parallélisation est effectuée en remplaçant la fonction Python usuelle "range" par la fonction Cython "prange", qui permet de spécifier manuellement le nombre de threads utilisés et la taille de chunk (qui détermine quelle fraction des calculs restant doit être allouée à chaque thread libre). Les fonctions d'affichage ("display", ainsi que l'interface graphique Tkinter) restent inchangées, à ceci près que les calculs sont effectués en appelant la fonction "mandelbrot" parallèle que nous venons de décrire.

4 La comparaison des performances des implémentations naïve et parallélisée de l'ensemble de Mandelbrot

Nous avons réalisé plusieurs visualisations (disponibles dans le notebook) pour étudier l'évolution du speed-up en fonction de la variation de différents paramètres (le nombre de threads utilisés, la taille de chunk et la résolution de l'image à afficher), que nous pouvons résumer par les figures 4 et 5.

Nous constatons systématiquement que le speed-up empirique est d'un ordre de grandeur supérieur au speed-up théorique que nous avons calculé précédemment grâce à la loi d'Amdahl.

Ayant identifié dans cette section les valeurs optimales des arguments "num_threads" et "chunksize" (Respectivement 2 et 10) , nous relançons l'interface graphique avec ces paramètres pour obtenir la meilleure fluidité possible dans notre exploration interactive de l'ensemble de Mandelbrot .

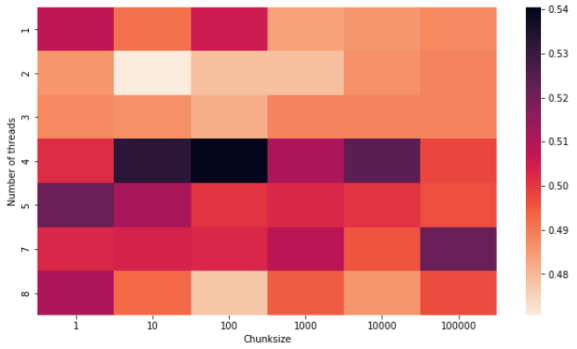

FIGURE 4 – Speed-up observé (colormap) en fonction du nombre de threads (lignes) et de la taille de chunk (colonnes)

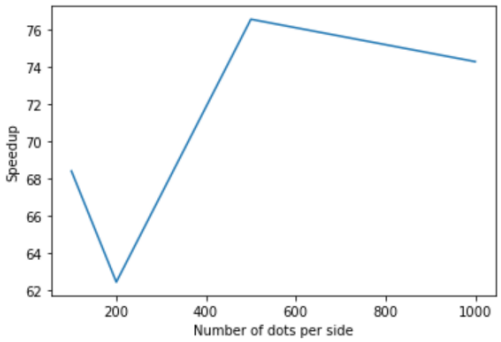

FIGURE 5 – Speed-up observé en fonction de la résolution (2 threads, taille de chunk de 10)

Références

[1] “How to plot the mandelbrot set”. In : (2020).

Liste des tableaux

Table des figures