TME 9

Objectifs pédagogiques :

- Manipulation de ForkJoin ;
- Introduction aux fractales ;

2 Mandelbrot

2.1 Definition

L'ensemble de Mandelbrot est un ensemble de points dans le plan complexe qui possède une structure fractale fascinante. Il a été popularisé par le mathématicien Benoît Mandelbrot dans les années 1970. Cet ensemble est défini par l'ensemble des nombres complexes c pour lesquels la suite (z_n) définie par la relation de récurrence suivante reste bornée :

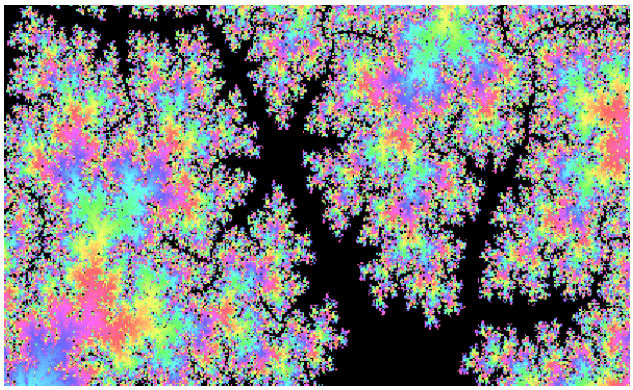
$$\begin{cases} z_{n+1} = z_n^2 + c \\ z_0 = 0 \end{cases}$$

Équation de l'ensemble de Mandelbrot.

Pour chaque point c du plan complexe, on calcule la suite z_n. Si cette suite reste bornée (c'est-à-dire qu'elle ne diverge pas vers l'infini) après un grand nombre d'itérations, alors le point c appartient à l'ensemble de Mandelbrot. Sinon, il n'en fait pas partie.

La vitesse de convergence de la suite (z_n) détermine la couleur attribuée à chaque point lors de la visualisation graphique de l'ensemble. Les points divergents sont colorés en noir, tandis que la couleur des autres points dépend de la vitesse de convergence.

Cette technique de coloration permet de révéler la complexité et la beauté fractale de l'ensemble de Mandelbrot.



2.2 Implémentation

On fournit une application qui sait déjà calculer une image en séquentiel. Sans tomber dans les détails, le code est défini dans une classe `MandelbrotCalculator`.

```
public class MandelbrotCalculator {
   public static void compute(BoundingBox boundingBox, int maxIterations, int[] imageBuffer)
        {
     long start = System.currentTimeMillis();
      // Iterate over each pixel
```

```
      for (int py = 0; py < boundingBox.height; py++) {
         for (int px = 0; px < boundingBox.width; px++) {
            int color = computePixelColor(boundingBox, maxIterations, px, py);

            // Set the pixel in the image buffer
            imageBuffer[py * boundingBox.width + px] = color;
         }
      }
      System.out.println("Rendered image in " + (System.currentTimeMillis() - start) + " ms"
          );
   }

   public static int computePixelColor(BoundingBox boundingBox, int maxIterations, int px,
       int py) {
   ...
  }
}
```

Un des points clés est que le calcul de la couleur d'un pixel prend au plus `maxIterations` itérations, mais peut aussi s'arrêter avant (pafois en un seul pas) si la suite diverge.

Question 1. A l'aide du framework ForkJoin, écrire une version parallèle `parCompute` de la méthode `compute` avec la même signature.

- On définira une nouvelle classe `MandelbrotTask` qui étend `RecursiveAction`.
- La tâche doit
  - Tester si la tâche est assez petite pour être calculée directement; pour celà on comparera le nombre de pixels à calculer à une constante `THRESHOLD` (e.g. 5000). Dans ce cas on fait un traitement séquentiel.
  - Sinon, on découpe l'image en deux parties (e.g. horizontalement) et on crée deux sous-tâches pour chaque partie.
- Il est donc nécessaire que la tâche possède des indicateurs (début/fin) pour définir la zone de l'image à traiter.
- L'opération `parCompute` doit simplement invoquer la tâche principale (e.g. en utilisant le `commonPool`) qui couvre toute l'image.

Question 2. Tester votre implémentation en comparant les temps d'exécution pour différentes parties de l'image et en faisant varier le nombre de threads, la taille de l'image, et maxIterations (on utilisera l'interface graphique fournie pour cette dernière partie).

2.3 QuickSort

Question 3. En vous basant sur le fichier fourni QuickSort.java, écrire une version parallèle de l'algorithme de tri QuickSort en utilisant ForkJoin.

Question 4. Essayez de dimensionner le nombre de threads plutôt que d'utiliser le pool par défaut.

Question 5. Faites varier le threshold de déclenchement d'un comportement séquentiel pour voir son impact sur les performances.