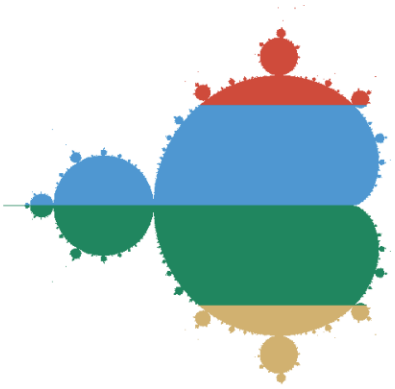
Un peu comme dans l'exercice précédent, il s'agit de découper une tâche globale en sous-tâches et d'affecter un thread à chaque sous-tâche. Puisque chacun des quatre threads se charge d'un quart de l'image, il doit savoir quelle zone de l'image lui est attribuée. C'est la raison d'être de l'attribut **`maZone`** qui prend une valeur entière entre 0 (le bas l'image) et 3 (le haut de l'image).

L'image est un carré de 500 lignes de 500 pixels. Le thread de la zone i sera chargé des lignes de l'image comprises entre $i \times 500/4$ et $(i+1) \times 500/4$. Il n'y a aucune difficulté particulière de codage, sauf si l'étudiant se hasarde sans raison à modifier le code de la méthode **`colorierPixel(i,j)`**, de la méthode **`mandelbrot(x,y,max)`** ou de la classe **`Picture`**. Au contraire, en suivant le modèle donné par l'exercice précédent, il suffit de modifier le **`main()`** et d'écrire une méthode **`run()`**.



```
public class Mandelbrot extends Thread{
  final static int taille = 500;   // nombre de pixels par ligne (et par colonne)
  final static Picture image = new Picture(taille, taille);
  final static int max = 100_000;

  final static int nbThreads = 4;
  private final int maZone;

  public Mandelbrot(int zone){
    maZone = zone;
  }

  public static void main(String[] args) throws Exception{
    final long début = System.nanoTime();

    // On crée et démarre les 4 threads de la classe Mandelbrot
    Mandelbrot[] mesThreads = new Mandelbrot[nbThreads];
    for (int i = 0; i < nbThreads; i++){
      mesThreads[i] = new Mandelbrot(i);
      mesThreads[i].start();
    }
    // On attend ensuite un par un qu'ils aient tous terminé
    for (int i = 0; i < nbThreads; i++){
      mesThreads[i].join();
    }

    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.println("Durée Totale = " + (long) durée /1000 + " s.");
    image.show();
  }

  public void run()  {
    final long début = System.nanoTime();
    for (int ligne = maZone*taille/4 ; ligne < (maZone+1)*taille/4; ligne++) {
      for (int i = 0; i<taille; i++){
        colorierPixel(i,ligne);
      }
      //synchronized(image){ image.show(); }
    }
    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.println("Durée de "+this.getName()+" = " + (long) durée /1000 + " s.");
  }
  ...
}
/*
   $ java -jar Mandelbrot.jar
   Durée de Thread-0 = 3 s.
   Durée de Thread-3 = 3 s.
   Durée de Thread-1 = 14 s.
   Durée de Thread-2 = 15 s.
   Durée Totale = 15 s.
   $
*/
```

On observe que les threads des zones 0 et 3, en haut et en bas de l'image, terminent beaucoup plus vite que les deux autres. La raison est simple si on explore le code de la fonction qui détermine si un pixel est blanc ou noir : il est plus facile de déterminer qu'un point est blanc, donc les lignes avec beaucoup de points blancs sont plus faciles à calculer. L'image étant symétrique, il est normal de constater une symétrie dans les temps de calcul des threads [3].

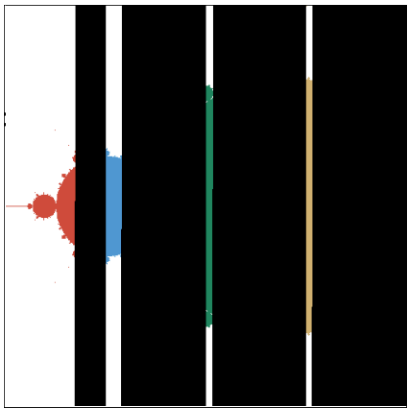
Pour aller plus loin Pour faire joli, même si ce n'est pas demandé dans l'exercice, chaque thread pourra être affecté d'une couleur distincte permettant de visualiser le travail réalisé par chacun. Il faudra alors définir de nouvelles couleurs et modifier la méthode `colorierPixel(i,j)` pour qu'elle tienne compte de l'attribut couleur du thread en train d'opérer.

Quelques erreurs fréquentes

1. Si l'on souhaite visualiser en temps réel l'évolution de l'image, on incorpore l'instruction `image.show()` après le dessin de chaque ligne dans la méthode `run()`. Il faut néanmoins protéger cette instruction par un verrou car elle n'est pas « threadsafe » [4]. Sinon, une exception risquera de planter un des threads et de conduire au final à une image incomplète. Cette erreur sera signalée par une exception.
2. Il faut veiller à opérer un `join()` sur chaque thread avant d'afficher le temps de calcul et l'image obtenue. Sinon, l'image apparaîtra là aussi incomplète.
3. Il faut veiller à couper horizontalement. Par exemple, une méthode

```
public void run()  {
  for (int i = maZone*taille/4 ; i < (maZone+1)*taille/4; i++) {
    for (int j = 0; j<taille; j++){
      colorierPixel(i,j);
    }
  }
}
```

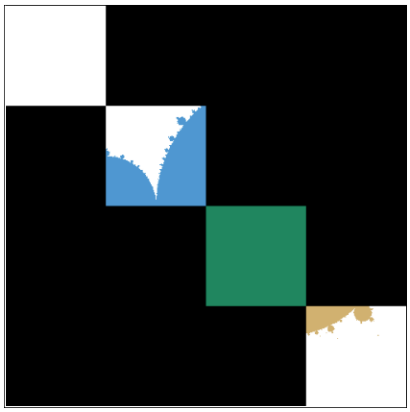
conduira à quelquechose comme ça :



4. Il faut veiller à ce que chaque thread effectue sa part de travail. Par exemple, une méthode

```
public void run() {
  int première = maZone*taille/4;
  int dernière = première + taille/4;
  for (int i = première ; i < dernière; i++) {
    for (int j = première; j < dernière; j++){
      colorierPixel(i,j);
    }
  }
}
```

donnera quelquechose comme ça :

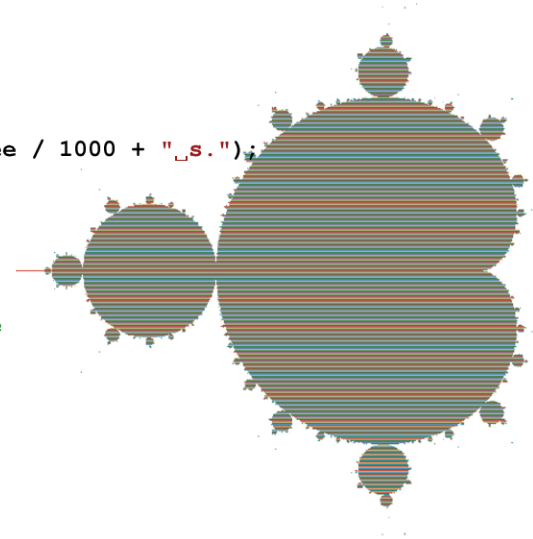


3. Si ce n'est pas le cas, c'est en général dû à une séparation verticale et non horizontale de l'image en quatre parties.
4. Normalement, un seul thread est chargé de la gestion des composants et du rafraichissement d'une fenêtre avec Swing.

Afin d'améliorer les performances, il faut mieux répartir la tâche de calcul sur les quatre threads. Plutôt que d'attribuer a priori une série de lignes à chaque thread, de manière statique préalablement au calcul, nous voulons à présent nous assurer que tous les threads participent tous à la tâche globale tant que celle-ci n'est pas achevée. Pour cela, un compteur partagé et initialisé à 0 va indiquer quelle est la prochaine ligne à dessiner. Lorsqu'un thread a terminé une ligne, il prendra en charge la prochaine ligne à calculer, sauf si celle-ci dépasse la taille de l'image, et incrémentera ce compteur.

Le compteur **prochaineLigne** sera déclaré **volatile** pour respecter la consigne présentée lors du premier cours. En effet, ce compteur est manipulé par plusieurs threads. En outre, ses incrémentations seront réalisées naturellement en exclusion mutuelle, à l'aide d'un verrou commun à tous les threads. Ce verrou est simplement le verrou intrinsèque d'un objet static, appelé **unVerrou**, partagé par tous les threads de cette classe.

```
public class Mandelbrot extends Thread{
  final static int taille = 500;   // nombre de pixels par ligne (et par colonne)
  final static Picture image = new Picture(taille, taille);
  final static int max = 100_000;

  final static int nbThreads = 4;
  private static volatile int prochaineLigne = 0;
  private static Object unVerrou = new Object();

  public static void main(String[] args) throws Exception{
    final long début = System.nanoTime();

    Mandelbrot[] mesThreads = new Mandelbrot[nbThreads];
    for (int i = 0; i < nbThreads; i++){
      mesThreads[i] = new Mandelbrot();
      mesThreads[i].start();
    }
    for (int i = 0; i < nbThreads; i++){
      mesThreads[i].join();
    }

    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.println("Durée␣Totale␣=␣" + (long) durée / 1000 + "␣s.");
    image.show();
  }

  public int ligneSuivante(){
    int ligne;
    synchronized(unVerrou){ // Incrémentation atomique
      ligne = prochaineLigne;
      prochaineLigne++;
    }
    return ligne;
  }

  public void run()  {
    final long début = System.nanoTime();

    int ligne = ligneSuivante() ;
    while(ligne < taille){
      for (int i = 0; i<taille; i++) colorierPixel(i,ligne);
      // synchronized(image){image.show();}
      ligne = ligneSuivante() ;
    }

    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.println("Durée␣de␣"+this.getName()+"␣=␣" + (long) durée / 1000 + "␣s.");
  }
  ...
}
```



qui produira l'exécution suivante :

```
$ java -jar Mandelbrot.jar
Durée de Thread-0 = 9 s.
Durée de Thread-2 = 9 s.
Durée de Thread-3 = 9 s.
Durée de Thread-1 = 9 s.
Durée Totale = 9 s.
$
```

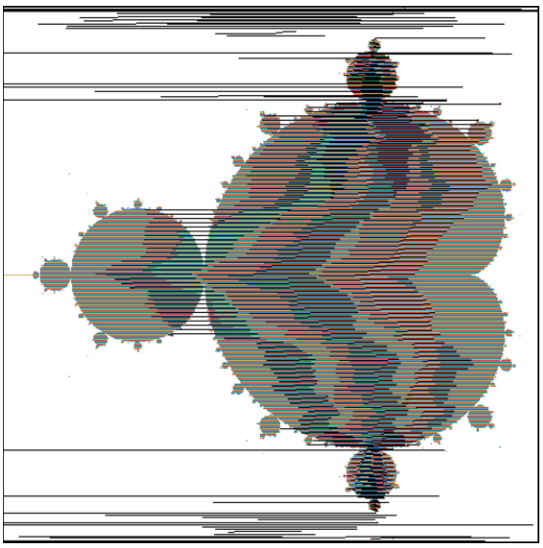
On observe ici que le temps de calcul global (9 s. pour chacun des 4 threads) est bien moindre qu'avec un découpage statique (15 s. pour le thread le plus lent) car la charge de travail est mieux répartie.

Quelques erreurs fréquentes

1. Il faut veiller à ce que chaque thread utilise une *variable locale* dans laquelle il conserve le numéro de la ligne dont il se charge. Par exemple, une méthode

```
public void run()  {
  while(prochaineLigne < taille){
    for (int i = 0; i<taille; i++){
      colorierPixel(i, prochaineLigne);
    }
    synchronized(unVerrou){
      prochaineLigne++;
    }
  }
}
```

   donnera quelquechose comme ça :



2. Il faut à incrémenter de manière atomique le compteur partagé, à l'aide du verrou. Par exemple, une méthode

```
public int ligneSuivante(){
  int ligne = prochaineLigne++;
  return ligne;
}
```

   pourra donner lieu à l'omision d'une ligne et à quelquechose comme ça :