Exercice D.1 Exécuteur d'une fractale Comme suite au premier exercice du premier TP, il s'agit aujourd'hui d'utiliser un réservoir de threads via un exécuteur pour coder plus facilement une version parallèle du programme.

Question 1. Récupérez tout d'abord le fichier `Mandelbrot.java` de l'archive `TP_A.zip`. Puis ajoutez à la classe **`Mandelbrot`** une sous-classe **`TraceLigne`** qui implémente **`Runnable`** et dont la méthode **`run()`** assure le tracé d'*une seule ligne*, déterminée par un attribut de ces objets.

Question 2. En reprenant le modèle de programme vu en cours et disponible dans l'archive `TP_C.zip`, créez ensuite un réservoir de threads comportant 4 threads et affectez lui le calcul des 500 lignes de l'image, vues chacune comme un objet de la sous-classe **`TraceLigne`** introduite.

Question 3. Testez le programme obtenu, en clôturant le réservoir à l'aide d'un appel à **`shutdown()`** puis en détectant sa terminaison de l'exécuteur via **`isTerminated()`** comme dans le cours, avant de conclure le **`main()`** par un affichage de l'image à l'aide d'un appel à la méthode **`show()`**.

Question 4. Modifiez votre programme pour afficher l'image *sans clôturer le réservoir de threads*, en utilisant *un loquet* (de la classe **`CountDownLatch`**), initialisé à 500 et décrémenté par chacune des 500 tâches.

Question 5. Modifiez à nouveau votre programme pour afficher l'image sans clôturer le réservoir ni utiliser de loquet, mais en vous appuyant sur un *service de complétion* comme dans le modèle du cours.

Exercice D.2 Parallélisation du tri rapide Le programme `TriRapide.java` de la figure 17 est disponible dans l'archive `TP_D.tgz` sur le site de l'UE. Il implémente le tri rapide[2] d'un tableau de longueur **`taille`** et formé d'entiers choisis aléatoirement entre **`-borne`** et **`+borne`**.

La stratégie employée suit une approche du type « *diviser pour régner* » ; elle consiste à placer un élément du tableau, appelé *pivot*, à sa place définitive, en permutant tous les éléments du tableau de telle sorte que :

— tous ceux qui sont inférieurs au pivot soient à sa gauche,
— tous ceux qui sont supérieurs au pivot soient à sa droite.

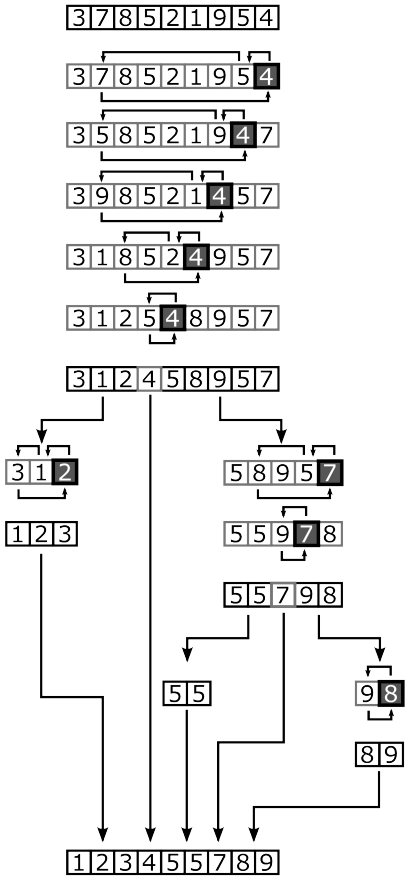
Cette opération s'appelle *le partitionnement*. Dès lors, il ne reste qu'à trier les deux sous-tableaux de chaque côté du pivot. *Ce sont là deux tâches indépendantes qui peuvent être traitées en parallèle* (cf. figure ci-contre).

Pour chacun des deux sous-tableaux résiduels, on choisit un nouveau pivot et on répète l'opération de partitionnement, récursivement, jusqu'à ce que chaque élément soit correctement placé : c'est le rôle dévolu à la méthode **`trierRapidement()`**.

En pratique, pour partitionner un sous-tableau :

— on place le pivot choisi à la fin du sous-tableau, en l'échangeant avec le dernier élément du sous-tableau ;
— on place tous les éléments inférieurs au pivot en début du sous-tableau, en conservant l'indice de celui le moins à gauche ;
— puis on place le pivot à la fin des éléments déplacés.

Ces opérations sont assurées par la méthode **`partitionner()`**. Le choix du pivot étant arbitraire, il peut par exemple correspondre systématiquement au dernier élément du tableau. Le fonctionnement global de cet algorithme est illustré sur la figure ci-contre.



L'exercice consiste à concevoir et à implémenter une version parallèle de cet algorithme en s'appuyant sur 4 threads rassemblés dans un **`executor`**, puis à mesurer le gain en terme de temps de calcul. Pour celà, le programme devra mesurer le temps de calcul de la version séquentielle donnée puis celui de la version parallèle obtenue, *sur un même tableau*, avant d'afficher le rapport entre ces deux mesures. Celles-ci seront effectuées avec une valeur de **`taille`** qui nécessite, en moyenne, un temps de calcul séquentiel de l'ordre de 10 s.

En guise de test, le programme devra également vérifier que les deux tableaux obtenus par le code séquentiel et le code parallèle sont bien identiques. Afin de maîtriser le nombre de tâches confiées aux threads, après chaque partionnement, les deux sous-tableaux seront traités en parallèle uniquement si leur taille est supérieure à 1000 et supérieure à un centième de la taille du tableau global. Dans le cas contraire, l'algorithme séquentiel sera employé.

2. Hoare, C. A. R. « Quicksort : Algorithm 64 » Comm. ACM 4 (7), 321-322, 1961

```
import java.util.Random ;

public class TriRapide {
  static int taille = 1_000_000 ;                    // Longueur du tableau à trier
  static int [] tableau = new int[taille] ;          // Le tableau d'entiers à trier
  static int borne = 10 * taille ;                   // Valeur maximale dans le tableau

  private static void echangerElements(int[] t, int m, int n) {
    int temp = t[m] ;
    t[m] = t[n] ;
    t[n] = temp ;
  }

  private static int partitionner(int[] t, int début, int fin) {
    int v = t[fin] ;                                 // Choix (arbitraire) du pivot : t[fin]
    int place = début ;                              // Place du pivot, à droite des éléments déplacés
    for (int i = début ; i<fin ; i++) {              // Parcours du *reste* du tableau
      if (t[i] < v) {                                // Cette valeur t[i] doit être à droite du pivot
        echangerElements(t, i, place) ;              // On le place à sa place
        place++ ;                                    // On met à jour la place du pivot
      }
    }
    echangerElements(t, place, fin) ;                // Placement définitif du pivot
    return place ;
  }

  private static void trierRapidement(int[] t, int début, int fin) {
    if(début < fin) {                                // S'il y a un seul élément, il n'y a rien à faire!
      int p = partitionner(t, début, fin) ;
      trierRapidement(t, début, p-1) ;
      trierRapidement(t, p+1, fin) ;
    }
  }

  private static void afficher(int[] t, int début, int fin) {
    for (int i = début ; i <= début+3 ; i++) {
      System.out.print("␣" + t[i]) ;
    }
    System.out.print("...") ;
    for (int i = fin-3 ; i <= fin ; i++) {
      System.out.print("␣" + t[i]) ;
    }
    System.out.print("\n") ;
  }

  public static void main(String[] args) {
    Random alea = new Random(System.currentTimeMillis()) ;
    for(int i=0 ; i<taille ; i++) {                          // Remplissage aléatoire du tableau
      tableau[i] = alea.nextInt(2*borne) - borne ;
    }
    System.out.print("Tableau␣initial␣:␣") ;
    afficher(tableau, 0, taille -1) ;                        // Affiche le tableau à trier

    long démarrage = System.nanoTime() ;
    trierRapidement(tableau, 0, taille-1) ;                  // Tri du tableau
    long terminaison = System.nanoTime() ;
    long durée = (terminaison - démarrage) / 1_000_000 ;

    System.out.print("Tableau␣trié␣:␣") ;
    afficher(tableau, 0, taille -1) ;                        // Affiche le tableau obtenu
    System.out.println("obtenu␣en␣" + durée + "␣millisecondes.") ;
  }
}
```

FIGURE 17 – Codage en Java de l'algorithme du tri rapide