Exercice C.1 Complétion de l'exécution d'une fractale Il s'agit d'utiliser un réservoir de threads afin de développer une version multithread du programme `Mandelbrot.java` de l'archive `TP_A.zip`, de manière un peu plus professionnelle, *c'est-à-dire sans reprendre le code que vous avez développé lors du TP A.*

Question 1. Ajoutez au programme `Mandelbrot.java`, fourni dans l'archive `TP_A.zip` et décrit sur la figure 3, une classe **`TraceLigne`** qui implémente l'interface **`Runnable`** et dont la méthode **`run()`** consiste à colorier les pixels d'*une seule ligne* de l'image. Le numéro de la ligne à tracer sera déterminé par un attribut entier, fixé à l'aide d'un constructeur de la classe **`TraceLigne`**.
Vous pourrez ici opter pour une classe interne statique placée dans la classe **`Mandelbrot`** elle-même.

Question 2. Testez cette nouvelle classe en modifiant le **`main()`** afin qu'il se limite à lancer un seul thread, chargé de dessiner la ligne du milieu (la ligne numéro 250) ; le **`main()`** devra aussi lancer la méthode **`show()`** afin de visualiser le résultat de la tâche soumise à ce thread.

Question 3. En reprenant l'un des modèles de programme vus en cours et disponibles dans l'archive `CM_3.zip`, créez ensuite un réservoir de threads comportant 4 threads et affectez lui le calcul des 500 lignes de l'image, vues chacune comme une instance de la classe **`TraceLigne`**.
Vous pourrez explorer la documentation de l'interface **`ExecutorService`** afin trouver un moyen de soumettre un **`Runnable`**, ou bien modifier **`TraceLigne`** afin qu'elle implémente un **`Callable`**.

Question 4. Testez le programme obtenu en prenant soin d'attendre la fin de l'exécution des 500 tâches avant l'affichage de l'image par la méthode **`show()`**.

```java
public class PremiersNombresPremiers {
  static final ArrayList<Long> premiersNombresPremiers = new ArrayList<Long>() ;
  static final long borne = 10_000_000 ;
  public static void main(String[] args) {
    final long début = System.nanoTime() ;
    for (long i = 1; i <= borne; i++) {
      if (BigInteger.valueOf(i).isProbablePrime(50)) premiersNombresPremiers.add(i) ;
    }
    final long fin = System.nanoTime() ;
    final long durée = (fin - début) / 1_000_000 ;
    System.out.print("Nombre de nombres premiers inférieurs à " + borne + " : ") ;
    System.out.println(premiersNombresPremiers.size()) ;
    System.out.format("Durée du calcul: %.3f s.%n", (double) durée/1000) ;
  }
}
```

FIGURE 15 – Programme Java calculant les premiers nombres premiers

Exercice C.2 Liste des premiers nombres premiers Le petit programme Java de la figure 15 est disponible dans l'archive `TP_C.zip` sur AMETICE. Il construit la liste des nombres premiers inférieurs à une borne, puis affiche le nombre d'éléments de la liste obtenue et la durée du calcul. Dans cet exercice, vous devez écrire un programme multithread qui produit également la liste des premiers nombres premiers, mais qui s'appuie sur un réservoir de quatre threads (autrement dit : un threadpool) afin de réduire le temps de calcul.

La liste produite par votre programme n'a pas besoin d'être identique à celle du code fourni, dans le sens où il n'est pas nécessaire qu'elle soit triée par ordre croissant. Cependant, vous pourrez, si vous le souhaitez, ajouter à votre code un appel à la méthode **`sort()`** en fin de calcul afin d'obtenir au final la liste croissante des premiers nombres premiers.

Avant toute chose, vous fixerez une valeur de la constante **`borne`** afin que le temps de calcul t_0 de ce programme soit proche de 10 secondes sur la machine avec laquelle vous travaillez. Le gain de performance obtenu est défini par la formule suivante : gain $= t_0/t_1$ où t_1 désigne le temps de calcul de votre programme multithread. Idéalement, ce gain devrait être voisin de 4, si votre machine dispose d'au moins 4 cœurs.

Votre programme devra exécuter le code séquentiel puis le code parallèle afin de calculer et d'afficher le gain obtenu. Vous indiquerez en commentaire au début du code à quoi correspondent les tâches soumises au threadpool.

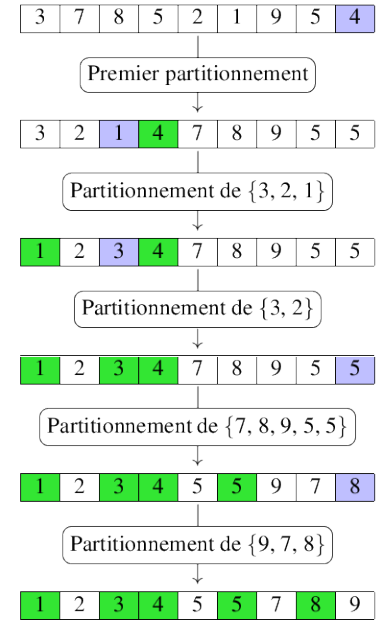

Illustration du tri par des partitionnements

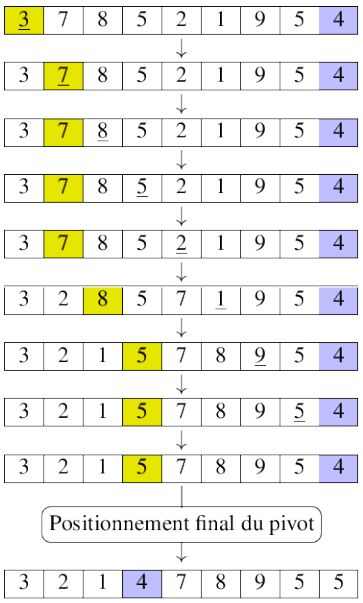

Illustration d'un partitionnement

FIGURE 16 – Fonctionnement de l'algorithme de tri rapide

Exercice C.3 Accélération du tri rapide Le programme `TriRapide.java` de la figure 17 est disponible dans l'archive `TP_C.zip` sur AMETICE. Il implémente le tri rapide [4] 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 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*. Il ne reste alors qu'à trier les deux sous-tableaux de chaque côté du pivot. *Ce sont là deux tâches indépendantes qui peuvent être traitées en parallèle* sur le même tableau.

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 récursive **`trierRapidement()`**. Une illustration globale du tri sur un tableau à 9 éléments est fournie sur la gauche de la figure 16, les pivots successifs étant placés à leur position définitive dans les cases vertes.

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 parcourt l'ensemble du tableau, de la gauche vers la droite, et l'on déplace tous les éléments inférieurs à la valeur du pivot en début du sous-tableau, en conservant l'indice de celui le moins à gauche ;

— puis on place le pivot à droite 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 à trier. Une illustration de l'opération de partition sur un tableau à 9 éléments est fournie sur la droite de la figure 16, la position présentie comme étant la future position du pivot étant indiquée en jaune.

Il s'agit ici de concevoir et d'implémenter une version multithread de ce programme qui s'appuie sur un threadpool formé de 4 threads, puis de mesurer le *gain* en terme de temps de calcul. Pour celà, le programme devra mesurer le temps de calcul du code fourni puis celui du programme obtenu, *sur un même tableau*. Ces deux mesures seront effectuées avec une **`taille`** de tableau qui nécessite approximativement un temps de calcul séquentiel de l'ordre de 10 s.

En guise de test, votre programme devra aussi 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 au réservoir de threads, après chaque partitionnement d'un tableau, les deux sous-tableaux seront traités en parallèle uniquement si la taille du tableau est supérieure à un centième de la taille du tableau global. Dans le cas contraire, l'approche séquentielle sera conservée.

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

```java
import java.util.Random ;

public class TriRapide {
  static final int taille = 1_000_000 ;                    // Longueur du tableau à trier
  static final int [] tableau = new int[taille] ;          // Le tableau d'entiers à trier
  static final 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.println() ;
  }

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

    long débutDuTri = System.nanoTime() ;
    trierRapidement(tableau, 0, taille-1) ; // Tri du tableau
    long finDuTri = System.nanoTime() ;
    long duréeDuTri = (finDuTri - débutDuTri) / 1_000_000 ;

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

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