TP de Compléments en Programmation Orientée Objet n° 11-12 : ForkJoin, CompletableFuture

Exercice 1 : Tri fusion

Le code suivant présente une implémentation du tri fusion en Java :

```java
import java.util.*;

public class TriFusion {

   protected static <E extends Comparable<? super E>> List<E> fusion(List<E> l1, List<E> l2) {
      // ArrayList plutôt que LinkedList (pour get en temps constant)
      List<E> l3 = new ArrayList<>(l1.size() + l2.size());
      ListIterator<E> it1 = l1.listIterator(), it2 = l2.listIterator();
      while (it1.hasNext() || it2.hasNext())
         if (it1.hasNext()
            && (!it2.hasNext()
               || l1.get(it1.nextIndex()).compareTo(l2.get(it2.nextIndex())) < 0))
            l3.add(it1.next());
         else l3.add(it2.next());
      return l3;
   }

   public static <E extends Comparable<? super E>> List<E> triMonoThread(List<E> l) {
      if (l.size() <= 1) return l;
      else {
         int pivot = Math.floorDiv(l.size(), 2);
         List<E> l1 = triMonoThread(l.subList(0, pivot));
         List<E> l2 = triMonoThread(l.subList(pivot, l.size()));
         return fusion(l1, l2);
      }
   }

   public static void main(String[] args) {
      // doit afficher : [1, 2, 12, 81, 99, 122, 122, 234, 2134]
      System.out.println(triMonoThread(Arrays.asList(234,2134,1,122,122,2,99,12, 81)));
   }
}
```

1. Programmez une version concurrente de ce code à l'aide de `ForkJoinPool`.
2. Même consigne avec `CompletableFuture`.

   Pour vous aider voici comment on traduit en `CompletableFuture` le programme du cours pour calculer la suite de Fibonacci avec `ForJoinTask` :

```java
import java.util.concurrent.CompletableFuture;
import static java.util.concurrent.CompletableFuture.*;

public class Fibo {
   public static CompletableFuture<Integer> calculFibo(int n) {
      if (n <= 1) return completedFuture(1);
      else return completedFuture(n - 1).thenComposeAsync(
      /* on "enrobe" les pseudo-appels récursifs à calculFibo2 dans une lambda passée à
      * thenComposeAsync afin d'éviter une exécution récursive dans le thread courant */
                  nn -> calculFibo(nn).thenCombine(calculFibo(nn - 1), (x, y) -> x + y)
               );
   }

   public static void main(String[] args) {
      System.out.println(calculFibo(30).join());
   }

}
```

L'idée c'est que la méthode "récursive" retourne un `CompletableFuture` et compose sa recette à partir de sous-tâches, tout en :

— évitant l'attente du résultat d'une sous-tâche (`get` ou `join`) au sein d'une tâche (c'est ce qu'on faisait avec `ForkJoinTask`, mais l'idée de `CompletableFuture` c'est justement de décomposer en tâches élémentaires autonomes).

— se méfiant des appels récursifs directs qui n'auraient pas été confiés à des tâches asynchrones (*i.e.* appel "enrobé" dans une lambda-expression passée en paramètre d'une méthode `xxxAsync` fournie par `CompletableFuture`). Le risque en faisant cela est d'exécuter tout l'algorithme récursif dans le *thread* courant.
Si vous faites quand-même un appel récursif direct, il faut bien s'assurer qu'ensuite, dans la méthode, l'essentiel du calcul est bien confié à une tâche asynchrone.

Exercice 2 : Factorisation d'entiers

But/prétexte de l'exercice : écrire une méthode qui factorise les nombres entiers en facteurs premiers en suivant l'algorithme récursif suivant :

factorize(n) :

— entrée : n, le nombre à factoriser

— on calcule : $m = \lfloor \sqrt{n} \rfloor$ (arrondi vers l'entier en dessous de la racine carrée de n)

— on part de m et on décroit jusqu'à trouver d le premier diviseur de n inférieur ou égal à m.

— on appelle factorize(d) et factorize(n/d)

— on retourne l'union des 2 listes obtenues ci-dessus.

1. Implémentez cet algorithme à l'aide d'une méthode faisant des appels récursifs sur le même *thread* (faites comme si vous n'aviez jamais entendu parler de *threads*). N'utilisez pas des `int` mais des `long`.
2. Réécrivez cette méthode pour que les tâches soient des `ForkJoinTask` qu'on envoie sur un `ForkJoinPool` de taille fixée. Pour cela, implémenter une classe `Factorisation` qui `extends` la classe `RecursiveTask<>`.

   Testez sur des entiers pour lesquels vous savez qu'il y a beaucoup de facteurs. Testez par exemple sur l'entier `17308840695300001` (notez le `l` : on travaille sur des `long`).
3. Réécrivez cette méthode en utilisant `CompletableFuture` à la place de `ForkJoinTask`.