Université Paris Cité

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

A finir l'exercice 1 de TP10 si ce n'est pas déjà fait.

1 Problèmes récursifs avec `ForkJoin`

Exercice 1 : Tri fusion

La méthode `fusion()` fait une fusion de deux listes triés :

```java
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public final class Fusion {
    private Fusion(){}
    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();
        //System.out.println( "Fusion dans thread "
        //     + Thread.currentThread().getName() + "  Listes de taille "
        //+ l1.size() + " et " + l2.size());
        while (it1.hasNext() && it2.hasNext()) {
            E e1 = it1.next();
            E e2 = it2.next();
            if (e1.compareTo(e2) <= 0) {
                l3.add(e1);
                it2.previous();
            } else {
                l3.add(e2);
                it1.previous();
            }
        }
        if( it1.hasNext() )
            l3.addAll( l1.subList(it1.nextIndex(),  l1.size()) );
        if( it2.hasNext() )
            l3.addAll( l2.subList(it2.nextIndex(),  l2.size()) );

        return l3;
    }
}
```

Elle est utilisée dans l'implémentation de tri fusion :

```java
import java.util.*;

public class TriFusion {

    private TriFusion() {     }

    public static <E extends Comparable<? super E>> List<E> triMonoThread(List<E> l) {
        if (l.size() <= 1) return l;

        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.fusion(l1, l2);
    }
}
```

Et on ajoute un petit programme de test :

```java
  public class Main {
    public static void main(String[] args) {

        List<Integer> p = Arrays.asList(234, 2134, 1, 122, 122, 2, 99, 12, 81);
```

```
        System.out.println("monothread");
        System.out.println(TriFusion.triMonoThread(p));
      }
  }
```

Question 1. Ajoutez dans la classe `TriFusion` la méthode

```
public static <E extends Comparable<? super E>> List<E> triForkJoin(List<E> l)
```

qui implémente une version concurrent de tri fusion en utilisant `ForkJoinTask`. Ajoutez un test dans `main()`.

Ajoutez l'affichage du nom de thread dans les fonctions `fusion()` et `triForkJoin()` et observer quel thread exécute chaque appel fonction.

Question 2. Quelle est l'efficacité de `triForkJoin` ?
Générez une très longue liste d'entiers aléatoires (essayez d'abord avec 10 millions d'éléments, si le temps de tri est trop long essayez avec les listes plus courtes).
La liste doit être générée sans boucle dans votre code.
Indication. `new Random(System.currentTimeMillis()).ints()` donne un stream infini d'entiers, limitez la longueur et transformer en liste.
Comparer le temps d'exécution de trois implémentations de tri :

- `TriFusion.triMonoThread()`
- `TriFusion.triForkJoin()`
- la fonction de tri fournie par java : `Collections.sort()`. (Cette fonction modifie la liste passée en paramètre. Pour garder la liste initiale passez dans `Collection.sort()` une copie de la liste initiale.)

sur la même liste générée dans la première partie de la question.
Indication. On peut assumer que le temps d'exécution c'est la différence de valeurs retournées par `System.nanoTime()` juste après et juste avant l'exécution de code.
Remarque. D'après la doc `Collections.sort()` est une version optimisée de tri fusion.

Question 3. L'utilisation de `ForkJoin` réaliste évite, si possible, de lancer les threads sur des petits sous-problèmes.
Dans la méthode `triForkJoin()` cela se traduit par la modification suivante : quand la liste devient plus petite qu'une valeur `threshold` au lieu de lancer les appels récursifs on appliquera `Collections.sort()` et on retournera le résultat.
Pour gérer la valeur de `threshold` vous ajouterez dans la classe `TriFusion` le code suivant :

```
private static long threshold = 1;
  public static void setThreshold(long threshold) {
      if(threshold <= 0) throw new IllegalArgumentException("threshold negative");
      TriFusion.threshold = threshold;
  }
  public long getThreshold(){
      return threshold;
  }

```

et vous modifierez `triForkJoin()` pour prendre en compte la valeur de `threshold`.
Comparer le temps d'exécution de :

- `TriFusion.triMonoThread()`
- `TriFusion.triForkJoin()` avec `threshold` par exemple `10_000`
- `Collections.sort()`

sur une liste de plusieurs millions d'entiers.

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 la liste obtenue en ajoutant le résultat de l'un de ces appels à la liste retournée par l'autre appel (chaque facteur premier p apparaît k fois sur la liste si le nombre est divisible par p^k mais pas par p^{k+1}).

La classe suivante implémente l'algorithme et fournit une fonction auxiliaire qui fait le tri de la liste obtenue.

```java
public class Factorisation {
    private Factorisation(){}
    public static List<Long> factoriseList(long number){
        var divisor = (long)Math.sqrt(number);
        while(number % divisor!=0){
            divisor--;
        }
        if(divisor==1){
            var selfSingleton = new ArrayList<Long>();
            selfSingleton.add(number);
            return selfSingleton;
        }
        else
        {
            var factors = factoriseList(divisor);
            factors.addAll(factoriseList(number/divisor));
            return factors;
        }
    }
    public static List<Long> factoriseWithMultiplicity(long number){
        List<Long> l = factoriseList(number);
        Collections.sort(l);
        return l;
      }
}
```

Question 1. Testez la méthode en factorisant un nombre, par exemple 1730884069530000L. Calculez le produit de tous les facteurs (sans boucle, en utilisant `stream`) et comparer avec le nombre de départ.

Question 2 Réécrivez l'algorithme pour que les tâches soient des `ForkJoinTask` qu'on envoie sur un `ForkJoinPool` de taille fixée. Pour cela, implémenter une classe `FactorisationFJ` qui **`extends`** la classe `RecursiveTask<>`.
Testez sur des entiers pour lesquels vous savez qu'il y a beaucoup de facteurs. Testez par exemple sur le même entier que dans la question précédente.

2 Calculs concurrents avec `CompletableFuture`

Exercice 3 :

On considère le programme suivant :

```java
public class Main {

    public static void main(String[] args) {

        var f = new ArrayList<CompletableFuture<Integer>>(4);
                f.add(CompletableFuture.completedFuture(12));
                f.add(CompletableFuture.completedFuture(2));
                f.add(CompletableFuture.completedFuture(5));
                f.add(CompletableFuture.completedFuture(8));

        CompletableFuture<Integer> x = f.get(0).thenCombine(f.get(1),
                (a, b) ->  a + b)
                .thenCombine(f.get(2), (a, b) ->  a * b );

        CompletableFuture<Integer> y = f.get(0).thenCombineAsync(f.get(1),
                (a, b) ->  a + b)
                .thenCombineAsync(f.get(2), (a, b) ->  a * b );

        x.join(); y.join();
    }
}
```

Question 1 Regarder le doc java pour les méthodes `thenCombine()` et `thenCombineAsync()` de la classe `CompletableFuture`.
Nous voulons observer quel thread exécute les `BiFunction`s implémentées par les lambda expressions `(a,b) -> a*b` et `(a,b) -> a+b`.

1. Ecrire la méthode `private static printThread(int i)` qui affiche l'entier `i` et le nom du thread courant.
2. Pour voir quel thread exécute les lambdas modifier le code de `main` de telle sorte que chaque l'expression lambda commence par exécuter `printThread(i)` avec les valeurs `i` différentes dans chaque lambda avant de retourner la valeur de lambda.

Exécutez et observez le threads affichés.

Question 2. Enlevez la ligne `x.join(); y.join();` de votre code.
Construire un nouveau `CompletableFuture` qui utilise les `CompletableFuture`s `x, y, f.get(3)` pour fabriquer intuitivement l'expression $(x + y) * f.get(3)$. Pour faire les compositions utilisez `thenCombine()`. Bien sûr vous devez utiliser `printThread()` dans les `BiFunction`s.
Afficher le résultat final du nouveau `CompletableFuture`.
Exécutez plusieurs fois le programme pour voir si le thread `main` est utilisé toujours dans les mêmes `Bifunction`s.

Conseil : n'hésitez pas à vous référer à la documentation de `CompletableFuture` [1].

Exercice 4 :

Le but est de calculer la somme $f(1) + \cdots + f(5)$ pour une certaines fonctions f dont on sait qu'elle est thread-safe et on sait que le temps d'exécution est considérable.
Nous voulons utiliser l'API de `CompletableFuture`.
A la place de la fonction f réelle vous allez utiliser une fonction factice :

1. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html

```java
public static int f( int i ){
    try{
        Thread.sleep(1000);
    }catch (InterruptedException e){}
    finally{    printThread(i); return i * i ; }
}

```

où `printThread()` est la fonction qui affiche le thread courant et que vous avez déjà implémenté dans l'exercice 3.

Question 1. Dans un premier temps vous devez écrire une « solution » basée sur les méthodes de l'exercice précédent (`completedFuture`, `thenCombine`, `thenCombineAsync`). Comme dans l'exercice précédent `printThread()` doit être aussi incorporé dans les lambda expressions.
Le programme doit afficher les résultat de calcul de $f(1) + \cdots + f(5)$ et aussi le temps de l'exécution (il suffit utiliser `System.currentTimeMillis()` pas la peine de mesurer en nano-secondes). Exécutez le programme.

1. Dans quels threads sont effectués les appels à la fonction f ? Observer aussi le temps d'exécution global.
2. Est-ce que cette « solution » utilise effectivement les threads pour paralléliser le problème ?

Question 2. La première tentative de solution a donné un programme avec les appels à f exécutés dans le même thread `main`. Modifiez cette solution pour que les appels à f soient exécutés dans d'autres threads. Observer les temps global d'exécution.
Suggestion : Regarder la méthode `supplyAsync()` de `CompletableFuture`.