TRAINA Tanguy
INFO2 – Logiciel

Structures et Algorithmes concurrents

Lab 4 : Fork/Join

Github : https://github.com/furtiveJack/Collections_Concurrente

Optimiser l'opération *reduce*

-> **Problème** : on veut factoriser et optimiser ces deux méthodes

```java
public static int sum(int[] array) {
    var sum = 0;
    for (var value : array) {
        sum += value;
    }
    return sum;
}
```

```java
public static int max(int[] array) {
    var max = Integer.MIN_VALUE;
    for (var value : array) {
        max = Math.max(max, value);
    }
    return max;
}
```

Optimiser l'opération *reduce*

- **Solution 1** : Ecrire une méthode *reduce* prenant un *IntBinaryOperator* pour connaitre l'opération à faire.

```java
public static int sum(int[] array) {
    return reduce(array,  initial: 0, Integer::sum);
}
```

```java
public static int max(int[] array) {
    return reduce(array, Integer.MIN_VALUE, Math::max);
}
```

- Méthode *reduce* "naïve":

```java
public static int reduce(int[] array, int initial, IntBinaryOperator op) {
    var acc = initial;
    for( var value : array) {
        acc = op.applyAsInt(acc, value);
    }
    return acc;
}
```

- Méthode *reduce* en utilisant l'API des Streams:

```java
public static int reduceWithStream(int[] array, int initial, IntBinaryOperator op) {
    return Arrays.stream(array).reduce(initial, op);
}
```

Optimiser l'opération *reduce*

- **Solution 2** : Si on veut accélérer le calcul, on peut utiliser des Streams parallèles plutôt que séquentiels.

```java
public static int parallelReduceWithStream(int[] array, int initial, IntBinaryOperator op) {
    return Arrays.stream(array).parallel().reduce(initial, op);
}
```

- Si on veut accélérer le calcul ET avoir plus de contrôle, on peut utiliser la méthode *divide and conquer* avec la classe *ForkJoin*. Le principe est le suivant :

```
solve(problem):
 if problem is small enough:
    solve problem directly (sequential algorithm)
 else:
    divide the problem in two parts (part1, part2)
    fork solve(part1)
    solve(part2)
    join part1
    return combined results
```

ForkJoinPool

- Pourquoi choisir un *ForkJoinPool* plutôt qu'un *ThreadPoolExecutor* ?
- -> Avec un *ThreadPoolExecutor*, si on fait des appels bloquants, on risque d'arrêter tous les *threads* du *pool*. Un *deadlock* apparait donc entre la soumission d'une nouvelle tâche (attendant qu'un *thread* soit disponible) et les autres *threads* en attente que la tâche que l'on veut soumettre soit finie.

- -> Avec un *ForkJoinPool*, on a le même comportement final qu'avec un *ThreadPoolExecutor*, à la différence qu'il est capable de s'arrêter et attendre la fin de l'exécution d'une tâche grâce à la méthode *join*, ce qui empêche l'apparition de *deadlocks*.

- -> **Solution 3** : Implémenter l'opération *reduce* à l'aide d'un ForkJoinPool

ForkJoinPool

Pour implémenter un ForkJoinPool, il faut commencer par créer une classe héritant de la classe RecursiveTask:

```
private static class ReduceTask extends RecursiveTask<Integer> {
ReduceTask(int[] array, int initial, IntBinaryOperator op, int start, int end) {
    Objects.requireNonNull(array);
    Objects.requireNonNull(op);
    this.array = array;
    this.op = op;
    this.initial = initial;
    this.start = start;
    this.end = end;
}
```

```
@Override
protected Integer compute() {
    if (end - start < 1024) {           // small enough
        return Arrays.stream(array, start, end).reduce(initial, op);
    }
    var middle = (start + end) / 2;
    ReduceTask t1 = new ReduceTask(array, initial, op, start, middle);
    t1.fork();
    ReduceTask t2 = new ReduceTask(array, initial, op, middle, end);
    var result2 = t2.compute();
    var result1 = t1.join();
    return op.applyAsInt(result1, result2);
}
```

Note : la méthode *compute* implémente l'algorithme *divide and conquer*

ForkJoinPool

- - Méthodes sur le ForkJoinPool:
  - - ***ForkJoinPool.commonPool()*** : renvoie le *ForkJoinPool* par défaut de la classe
  - - ***ForkJoinPool.invoke()*** : renvoie la valeur de retour de la *RecursiveTask* associée au *ForkJoinPool*

```
public static int parallelReduceWithForkJoin(int[] array, int initial, IntBinaryOperator op) {
    var task = new ReduceTask(array, initial, op, start: 0, array.length);
    var pool = ForkJoinPool.commonPool();
    return pool.invoke(task);
}
```

On peut généraliser cette idée pour implémenter *reduce* sur n'importe quelle collection, à l'aide d'un *spliterator*.

Spliterator

Méthodes à utiliser :

- ***Spliterator.estimateSize()*** : essaye d'estimer la taille du *spliterator* (renvoie Long.MAX_VALUE si échec)
- ***Spliterator.tryAdvance()*** : parcourt les éléments et applique une action à chacun d'entre eux
- ***Spliterator.trySplit()*** : essaie de découper en deux le *spliterator* (renvoie *null* si échec)

Exemple d'implémentation du divide and conquer dans la méthode compute:

```
@Override
protected V compute() {
    var size = spliterator.estimateSize();
    if (size == Long.MAX_VALUE) {
        return null;
    }
    if (size < threshold) {
        var box = new Object() {
            private V acc = initialValue;
        };
        while (spliterator.tryAdvance(e -> box.acc = acc.apply(e, box.acc)));
        return box.acc;
    }
    var t1 = new ReduceTask<>(spliterator.trySplit(), initialValue, acc, combiner, threshold);
    t1.fork();
    var t2 = new ReduceTask<>(spliterator, initialValue, acc, combiner, threshold);
    var result2 = t2.compute();
    var result1 = t1.join();
    return combiner.apply(result1, result2);
}
```