Collections Concurrentes
TP 04 - Fork/Join pool et Décomposition Récursive d'un problème en vue de sa parallélisation

https://github.com/MelissaDaCosta/structures_concurrentes

Mélissa DA COSTA

Contexte

Pour paralléliser un problème, il est plus simple de le décomposer en sous-problèmes plus facile à résoudre.

C’est la technique de **Fork/Join**.

Fork pour la partie : **diviser** le problème
Join pour la partie : **combiner** les résultats des sous-problèmes

Fonctions d'ordre supérieur : reduce

Une fonction d'**ordre supérieur** est une fonction qui :

- prend une ou plusieurs fonctions en paramètres
  ou
- renvoie une fonction

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

C'est le cas de la fonction **reduce**. Elle prend un *IntBinaryOperator* qui permet de lui donner une méthode à appliquer (ex : Math::sum)

Parallélisation

La parallélisation consiste à décomposer un problème en **sous-problèmes** pour résoudre ces problèmes simultanément dans différents **threads**. Puis, assembler les résultats pour résoudre le problème initial.

La parallélisation ne veut pas dire plus rapide : si le temps passer à diviser le calcul et à assembler les résultats est plus long que de résoudre le calcul, alors la parallélisation n’est pas utile.

La parallélisation est efficace pour des opérations **lourdes**.

Fork/Join

La parallélisation est utilisé dans la technique Fork/Join qui consiste à utiliser l'algorithme 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

Un **ForkJoinPool** est une piscine de tâches sachant se diviser.

La seule différence entre un **ForkJoinPool** et un **ThreadPool** est que le **ForkJoinPool** est capable de faire des join, de s'arrêter.

On ne peut pas utiliser un **ThreadPoolExecutor** car il n’est pas possible de faire un **join**. **Join** est bloquant, il attend que le thread soit terminé. Si plusieurs appels bloquant sont faits, toutes les **threads** de la piscine sont arrêtés et il peut y avoir un **deadlock**.

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

RecursiveTask

Une **RecursiveTask** est une tâche qui peut se diviser en plus petite tâche.
Lorsque l'on fait un **join** dans une **RecursiveTask**, on enlève la tâche qui appel le **join** du **ForkJoinPool** et on la remet lorsque la tâche qui fait le calcul sur lequel on attend a fini son calcul.
Cela empêche les **deadlock**.

```
  protected Integer compute() {
    if (end - start < SIZE_LIMIT) { // assez petit pour faire le calcul direct
      return Arrays.stream(array, start, end).reduce(initial, op);
    } else {
      var middle = (start + end) / 2;
      ReduceTask r1 = new ReduceTask(start, middle, array, initial, op);
      ReduceTask r2 = new ReduceTask(middle, end, array, initial, op);
      r1.fork(); // déléguer cette partie a une autre thread
      var res2 = r2.compute(); // continue de faire le calcul dans cette thread
      var res1 = r1.join(); // attend tant que r1 n'a pas fini son calcul
      return op.applyAsInt(res1, res2);
    }
  }
}
```