Collections Concurrentes

Lab 4 : **Fork/Join**

Fork/Join pool et Décomposition Récursive d'un problème en vue de sa parallélisation.

SORAN Altan - ESIPE INFO 2020

https://github.com/asoran/Structures-et-Algo-Concurentes

Intro : Parallélisme

Que ce soit en Java ou dans n'importe quel autre langage, il se peut que l'on doit traiter de la grosse donnée ou que l'on veut résoudre un "gros problème".

Dans la vraie vie, si on a beaucoup de chose a faire, on voudrait bien diviser la tâche en plusieurs sous-tâches et les distribuer sur plusieurs personnes.

On peut faire presque pareille avec du code ! On va utiliser les plusieurs coeurs de l'ordinateur pour diviser la tâche et la traiter de façon ***parallèle*** !

On va voir dans les prochaines slides plusieurs façons de faire cela en Java.

Parallel Streams

En java on peut représenter un gros ensemble de donnée à traiter par un Stream.

Avec la méthode `Stream::parallel` on peut rendre ce stream parallel, c'est à dire les prochaine opérations vont essayées de s'exécuter en tirant partie des plusieurs coeurs de la machine.

Une des opérations que l'on veut souvent faire c'est l'opération de réduction `Stream::reduce` qui prend en paramètre un `BinaryOperator<T>` !

La fonction donnée en paramètre doit savoir prendre 2 éléments du stream et de retourner un élément du même type.

Fork/Join

Depuis la version 7 de Java, l'API propose une mécanique de fork/join qui se repose sur le principe de "diviser pour mieux régner" (pour avoir un peu plus de contrôle).

Avant de paralléliser un problème il faut qu'on le décompose de façon récursive :

```
resoudre(probleme):
    si le problème est assez "petit":
        (Résoudre le problème de façon séquentiel)()
        Note: ça ne sert a rien de diviser le problème si la division du problème
        prend plus de temps que sa résolution par calcul séquentiel ;)
    sinon:
        Diviser le problème en 2 parties (partie1, partie2)
        fork resoudre(partie1)
        resoudre(partie2)
        join partie1
        retourner les résultats combinés
```

ForkJoinPool & RecursiveTask

Pour utiliser cette mécanique de *fork/join*, on utilise une ForkJoinPool, qui est juste un ExecutorService, qui joue un pool de thread ayant des tâches sachant se subdiviser (des RecursiveTask).

On peut se demander “pourquoi ne pas utiliser un ThreadPoolExecutor classique ?”: Dans une executor classique, on peut créer un deadlock si on fait des appels bloquants (et arrêter toutes les threads du pool) entre la soumission d'une nouvelle tâche et toutes les threads en attente que la tâche que l'on doit soumettre est fini

La ForkJoinPool met en place des sécurités avec les méthodes `fork()` et `join()`. On enlève la tâche qui appel le `join()` et on la remet lorsqu'elle a finit son calcul, comme cela pas de deadlock.

RecursiveTask

Pour créer un tache récursive on doit créer une class qui implémente `RecursiveTask<T>` et coder la méthode `<T> compute()` .

C'est dans cette méthode compute que l'on va utiliser les `fork()`/`join()`.
Exemple avec une RecursiveTask qui calcule fibonacci:

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }
    @Override protected Integer compute() {
        if (n <= 1) { return n; }
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }}
```

ForkJoinPool & RecursiveTask: Recap

Pour paralléliser notre tâche il nous faut 2 choses:

- Une ForkJoinPool
- Une classe qui peut résoudre notre tâche et la diviser si besoin

Voici un exemple de code (avec la classe Fibonacci de la slide d'avant), on peut obtenir le ForkJoinPool par défaut avec `ForkJoinPool::commonPool`.

```
public static int fibonaciForkJoin(int n) {
    var pool = ForkJoinPool.commonPool();
    return pool.invoke(new Fibonacci(n));
}
```