Parallel functional programming in Java 8

A tutorial for Java programmers (By: David Ådvall & Erik Pihl)

Does parallelism in Java give you headache-inducing thoughts about things like threads, deadlocks and race conditions? These days, it doesn't have to be that way. Since the release of Java 8, functions are first class citizens. These make it rather easy to write parallel code operating on arrays.

Functions and immutable data

The idea of functions in functional programming is the same as you remember from math class. A function is *pure*; it always gives the *same output for a specific input*, with *no side effects*. This means that the only thing a function can do is to give a return value. It can never change the state of anything else.

It also means that data needs to be *immutable*, i.e. once a piece of data has been initialized, it can never be modified. By having immutable data we also remove the need of locks and thus making parallel computations faster.

```
final int i1 = 42; // i1 is immutable, it can never be assigned a new value
int i2 = 66; // i2 is clearly not immutable, you can later say "i2 = 7;"
```

A really neat thing with functional programming is the concept of *higher order functions*. Wikipedia tells you that a higher order function *is a function that does at least one of the following:*

- *takes one or more functions as arguments*
- *returns a function as its result*

This is essential to parallel functional programming, since this gives us a simple way to express, for example, that a function should be applied to all elements of a data structure, without any notion of sequence. Later in this tutorial, we will give examples of higher order functions and how to use them in Java 8.

Streams

Streams are a way of expressing sequences of elements on which you can apply aggregate operations. There are many types of things you can stream in Java but in this tutorial we will use arrays. We will now show you some of the aggregate operations available and explain how they work.

The *map()* function

map() is a common higher order function. It takes a function as an argument and applies it to all elements of a stream. It will return an array of the same size as the input array. In this example we have used an anonymous function that increments a value.

```
printArray(a); // Prints the array a, for example 0 1 2 3 4
Integer[] b = Arrays.stream(a).map(i -> i + 1).toArray(Integer[]::new);
printArray(b); // Prints the incremented array, for example 1 2 3 4 5
```

The *filter()* function

filter() is a function that takes a boolean function and returns all the elements which returns true when applied to the boolean function.

```
// assume a is an Integer array
printArray(a); // Prints: 0 1 2 3 4
Integer[] c = Arrays.stream(a).filter(i -> i % 2 == 0).toArray(Integer[]::new);
printArray(c); // Prints: 0 2 4
```

It won't surprise you that we can combine filter() with another function, for example map(), in the following way. You can create as long chains of functions as you need.

```
Integer[] d = Arrays.stream(a)
        .filter(i -> i % 2 == 0)
        .map(i -> i * 2)
        .toArray(Integer[]::new);
printArray(d); // Prints: 0 4 8
```

The *reduce()* function

Not all higher order functions return arrays. reduce() for example returns a single value. What it does is it "folds" in a binary operator (the function that you pass to it as an argument) between all the elements of the array. It also has to be passed the operator's identity element as an argument.
For example,
[1,2,3,4].reduce(0,⊕) produces a result equivalent to (0⊕1⊕2⊕3⊕4).
If ⊕ is the function *add*, the result would be equivalent to *add*(0, *add*(1, *add*(2, *add*(3, 4))))

```
Integer s = Arrays.stream(a).reduce(0, (m, n) -> m + n)
System.out.println(s); // Prints: 10
```

Do that in parallel!

So far all the examples have been fully sequential code. To make them run in parallel we only need to add *parallel()* after *stream()*. It will take care of everything for us, creating threads, balancing load between cores and merging the results.

For example,

```
Integer[] b = Arrays.stream(a).parallel().map(i -> i + 1).toArray(Integer[]::new);
```

Is guaranteed to return the same value (given the same input) as

```
Integer[] b = Arrays.stream(a).map(i -> i + 1).toArray(Integer[]::new);
```

with the only difference that the former is run in parallel thanks to the `.parallel()` there in the middle.

Let's try a little more intense problem and see if that parallelism does any good.

```
Double[] randoms = new Double[10000000];
for (int i = 0; i < randoms.length; i++) {
            randoms[i] = Math.random();
}

Arrays.stream(randoms).map(i -> i*Math.sin(i)/i).toArray(Double[]::new);
Arrays.stream(randoms).parallel().map(i -> i*Math.sin(i)/i).toArray(Double[]::new);
```

Running this example on a MacBook with 2 cores and 4 threads we get the following execution times:

- Sequential run: 4402 ms
- Parallel run: 2741 ms

Which is a speedup of about a factor 1.6. The reason we don't get a factor 2 (or even more) here is that parallelism doesn't only give. It also comes with some overhead costs, which makes every parallel implementation a matter of cost-benefit balancing.

Try running the example yourself, and compute what kind of speedup you get! If you are running on a chip with more cores than used here, you should be able to see bigger speedups.

You might think you can do better performance for this specific case with some old school Java parallel magic. You might be right. This way of programming is not guaranteed to be optimal in a performance perspective (although it will be groundbreakingly fast in some important cases) but, from a maintenance and development time point of view, you can see that it is very good.

This tutorial ends here, but there is still much for you to learn. We have not show you all of what you can do with streams. There are more stream operations that you can use to match your needs. Some examples are sorted(), count(), max(), min().