Parallel Stream

This lesson introduces parallel streams, which are used to process the stream elements in parallel.

We'll cover the following

Until now, we have only been looking at serial Stream. However, Java 8 introduced the concepts of the parallel stream and parallel processing. As we have a greater number of CPU cores nowadays, due to cheap hardware costs, parallel processing can be used to perform operation faster.

Creating a parallel stream

There are two ways in which we can create a parallel Stream:

- Using the `parallelStream()` method.
- Or, if we already have a stream, we can use the `parallel()` method to convert it into a parallel stream.

Let us look at an example of a parallel stream.

```
import java.util.stream.Stream;

public class ParallelStreamDemo {

    public static void main(String args[]) {
        System.out.println("------------------------------ Serial Stream  ----------------------
        Stream.of(1, 2, 3, 4, 5, 6, 7)
                .forEach(num -> System.out.println(num + " " + Thread.currentThread().getName()));


        System.out.println("------------------------------ Parallel Stream ---------------------
        Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
                .parallel()
                .forEach(num -> System.out.println(num + " " + Thread.currentThread().getName()));
    }
}
```

When I run this program in my local environment, I get the below output.

In the case of a serial stream, the main thread is doing all the work. Meanwhile, in the case of a parallel stream, two threads are spawned simultaneously, and the stream internally uses `ForkJoinPool` to create and manage threads. Parallel streams create a `ForkJoinPool` instance via the static `ForkJoinPool.commonPool()` method.

The parallel stream makes use of all available CPU cores and processes the tasks in parallel. If the number of tasks exceeds the number of cores, then the remaining tasks wait for the currently running tasks to complete.

```
"C:\Program Files\Java\jdk1.8.0_171\bin\java.exe" ...
------------------------------- Serial Stream  --------------------------------
1 main
2 main
3 main
4 main
5 main
6 main
7 main
------------------------------- Parallel Stream ----------------------------
9 main
5 ForkJoinPool.commonPool-worker-1
13 ForkJoinPool.commonPool-worker-2
10 main
14 ForkJoinPool.commonPool-worker-2
4 ForkJoinPool.commonPool-worker-1
8 main
7 ForkJoinPool.commonPool-worker-1
12 ForkJoinPool.commonPool-worker-2
6 ForkJoinPool.commonPool-worker-1
11 main
1 ForkJoinPool.commonPool-worker-1
2 ForkJoinPool.commonPool-worker-2
3 main

Process finished with exit code 0
```

The parallel stream is fast but does this mean we should always create a parallel stream? Are there any situations where we should not use a parallel stream?

The answer is **yes**.

A parallel stream has a lot of overhead compared to a sequential one. Coordinating the threads takes a significant amount of time. We should always use serial stream and consider using a parallel Stream in the following cases:

1. We have a large amount of data to process.
2. We already have a performance problem in the first place.
3. All the shared resources between threads need to be synchronized properly otherwise it might produce unexpected results.

According to Brian Goetz (Java Language Architect & specification lead for Lambda Expressions), the following things should be considered before going for parallelization:

1. Splitting is not more expensive than doing the work.
2. Task dispatch or management costs between the threads is not too high.
3. The result combination cost must not be too high.
4. Use the NQ formula to decide if you want to use parallelism.

```
NQ Model:

N x Q > 10000

where,
N = number of data items
Q = amount of work per item
```

In the next lesson, you will learn about the lazy evaluation in Stream.