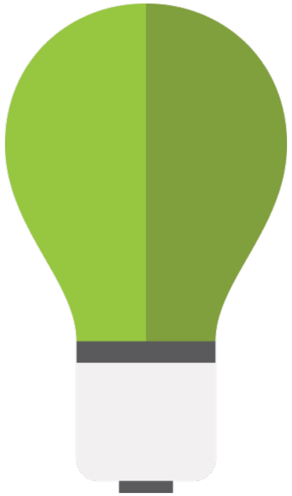
Parallel Data Processing Pipelines Using Java 8 Streams

José Paumard

@JosePaumard | blog.paumard.org

Agenda



Parallel streams

Stateful vs. stateless operations

Parallel reductions

Parallel Streams

Fast computations!

Going Parallel

- To allow for faster computation
- To leverage the multicore

- Multithread ≠ parallel
- Multithread: one process = one thread, so many processes at the same time
  - Problems: race condition, thread synchronization, variable visibility
- Parallel: one process = many thread, to go faster
  - Problems: algorithm, data ditribution among the CPU cores

What Are the Available Tools?

- In Java 6 and before: none!
  - Everything has to be handled « by hand »
- In Java 7
  - The fork / join framework
  - A 3rd party API: parallel arrays (Java 6 compatible, with its own embedded fork / join)
- In Java 8: parallel streams
  - Much easier and safer to use

Parallel Streams

- Two patterns:

```
// create the stream by calling parallelStream()
List<Person> people = ... ;
people.parallelStream()
      .filter(person -> person.getAge() > 20)
      .forEach(System.out::println);
```

Parallel Streams

- Two patterns:

```
// call parallel on an existing stream
List<Person> people = ... ;
people.stream().parallel()
       .filter(person -> person.getAge() > 20)
       .forEach(System.out::println);
```

- The order in which the people will be printed out is not guaranteed

Parallel Streams

- Two patterns:

```
// call parallel on an existing stream
List<Person> people = ... ;
people.stream().parallel()
      .filter(person -> person.getAge() > 20)
      .sorted()
      .forEach(System.out::println);
```

- The order in which the people will be printed out is not guaranteed
- To guarantee the order of the elements, we must use `sorted()`

Stateful vs. Stateless Operations

Caveats when going parallel

Caveats with Parallel Streams

- Parallel streams are built on top of the fork / join pattern
- Some things are to be avoided when computing things with the fork / join

- Synchronization and visibility issues!
- Stateful streams will not be computed efficiently in parallel

Stateful vs. Stateless Operations

- Example of a stateless operation

```
// call parallel on an existing stream
List<Person> people = ... ;
people.stream().parallel()
      .filter(person -> person.getAge() > 20)
      .sorted()
      .forEach(System.out::println);
```

- No outside information is needed to compute this boolean

Stateful vs. Stateless Operations

- Example of a stateful operation

```
// call parallel on an existing stream
List<Person> people = ... ;
people.stream().parallel()
      .skip(2)
      .limit(5)
      .forEach(System.out::println);
```

- We need a counter to remove the first 2 and keep the next 5 people
- This counter has to be visible among the threads → `AtomicLong`

Stateful vs. Stateless Operations

- How can we tell a stateful operation from a stateless one?
  - It is written in the Javadoc
  - With a little habit, it is easy to tell

- A stateful operation should not be used in parallel, it will kill performances!

Example 1: Performance

- Let us see this 1st code

```
List<Long> list = new ArrayList<>(10_000_100);
for (int i = 0 ; i < 10_000_000 ; i++) {
   list.add(ThreadLocalRandom.current().nextLong());
}
```

- We just generate 10M random longs in a loop, and store them in a list

Example 1: Performance

- Let us see this 2nd code

```
Stream<Long> stream =
    Stream.generate(() -> ThreadLocalRandom.current().nextLong());
List<Long> list =
    stream.limit(10_000_000).collect(Collectors.toList());
```

- The same as the previous one, with a stream
- We will be able to call `parallel()` on that stream
- There is a stateful operation there!

Example 1: Performance

- Let us see this 3rd code

```
Stream<Long> stream =
   ThreadLocalRandom.current().longs(10_000_000).mapToObj(Long::new);
List<Long> list = stream.collect(Collectors.toList());
```

- The same as the first one, with a stream
- We will be able to call `parallel()` on that stream
- And again, there is a stateful operation there!

Example 1: Performance

- Let us see the performances

	Not parallel	Parallel
Code 1 (for)	270 ms	
Code 2 (limit)	310 ms	
Code 3 (longs)	250 ms	

Example 1: Performance

- Let us see the performances

	Not parallel	Parallel
Code 1 (for)	270 ms	
Code 2 (limit)	310 ms	500 ms
Code 3 (longs)	250 ms	320 ms

- Performances are worse!
- And it consumes all the cores instead of one!

Example 2: a Sneaky Stateful Operation

- Stateful? Not stateful?

```
// stateful?
List<Person> people = Arrays.asList(p1, p2, p3);
people.stream().parallel()
      .filter(person -> person.getAge() > 20)
      .forEach(System.out::println);
```

- Stateful!

Example 2: a Sneaky Stateful Operation

- Stateful? Not stateful?

```
// stateful?
List<Person> people = Arrays.asList(p1, p2, p3);
people.stream().parallel() // this stream is ordered!
       .filter(person -> person.getAge() > 20)
       .forEach(System.out::println);
```

- Stateful!
- Because the stream on `ArrayList` is ordered!

Example 2: a Sneaky Stateful Operation

- Stateful? Not stateful?

```
// stateful?
List<Person> people = Arrays.asList(p1, p2, p3);
people.stream().parallel()
      .unordered() // set the ORDERED bit to 0
      .filter(person -> person.getAge() > 20)
      .forEach(System.out::println);
```

- Stateful!
- Calling unordered() will relax the constraint

Parallel Reductions

Use collectors instead of `reduce()`

Parallel Reduce Reduction

- Do not use this code in parallel!

```
List<Person> people = ...;

List<Integer> ages =
people.stream().parallel()
       .reduce(
          new ArrayList<Integer>(),
          (list, p) -> { list.add(p.getAge()) ; return list ; }
          (list1, list2) -> return list1 ;
 );
```

- Why?

Parallel Reduce Reduction

- Do not use this code in parallel!

```
List<Person> people = ...;

List<Integer> ages =
people.stream().~~parallel()~~
      .reduce(
         new ArrayList<Integer>(),
         (list, p) -> { list.add(p.getAge()) ; return list ; }
         (list1, list2) -> return list1 ;
 );
```

- Because `ArrayList` is not concurrent aware, and race conditions will occur

Parallel Reduce Reduction

- The right pattern is this one

```
List<Person> people = ...;

List<Integer> ages =
people.stream().parallel()
      .collect(Collectors.toList());
```

- `Collectors.toList()` will handle parallelism and thread-safety for us

Tuning Parallelism

- By default, the Fork / Join takes all the available CPUs
- It uses a pool of threads: the Common Fork / Join pool
- We can control this pool:

```
System.setProperty(
   "java.util.concurrent.ForkJoinPool.common.parallelism", 2) ;
```

Tuning Parallelism

- And we can also launch our computations in our own pool:

```
List<Person> persons = ... ;

ForkJoinPool fjp = new ForkJoinPool(2);
fjp.submit(
    () ->                                   //
    persons.stream().parallel()             // this is an implementation
           .mapToInt(p -> p.getAge())       // of Callable<Integer>
           .filter(age -> age > 20)         //
           .average()                       //
).get(); // from Future
```

Live Coding

Parallel streams in action

Distribution of the computation in the CommonForkJoinPool

Live Coding Summary

- We saw how parallelism is implemented in the stream API
- We saw the right pattern to collect elements in a list, in a thread safe way
- And we saw which pattern NOT to use!

Summary

- How parallelism can speed up computations
- But also how it can kill performances! (stateful vs stateless operations)
- How to configure our applications to control parallelism
- Hints at patterns to conduct parallel reductions