Overview of Java 8 Parallel Streams

(Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Know how aggregate operations & functional programming features are applied in the parallel streams framework

• Be aware of how a parallel stream works “under the hood”

Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied in the parallel streams framework
- Be aware of how a parallel stream works “under the hood”
- Recognize now to avoid concurrency hazards in parallel streams
Overview of How a Parallel Stream Works
A Java 8 parallel stream implements a “map/reduce” variant optimized for multi-core processors.

See [en.wikipedia.org/wiki/MapReduce](http://en.wikipedia.org/wiki/MapReduce)
A Java 8 parallel stream implements a “map/reduce” variant optimized for multi-core processors.

It’s actually a three phase “split-apply-combine” data processing strategy.

See [www.jstatsoft.org/article/view/v040i01](http://www.jstatsoft.org/article/view/v040i01)
The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”

- Splitterators are defined to partition collections in Java 8

```java
public interface Splitter<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Splitter<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/Splitter.html](docs.oracle.com/javase/8/docs/api/java/util/Splitter.html)
The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”
   - Spliterators are defined to partition collections in Java 8
   - You can also define custom spliterators

See [github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamSpliterator](https://github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamSpliterator)
The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”
   - Spliterators are defined to partition collections in Java 8
   - You can also define custom spliterators
   - Parallel streams perform better on data sources that can be split efficiently & evenly

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent "chunks"

2. **Apply** – Process chunks independently in one (common) thread pool

Splitting & applying run simultaneously (after certain limit met), not sequentially.
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”

2. **Apply** – Process chunks independently in one (common) thread pool
   - Programmers have some control over how many threads are in the pool
The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into independent “chunks”

2. **Apply** – Process chunks independently in one (common) thread pool

3. **Combine** – Join partial results into a single result
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into independent “chunks”

  2. **Apply** – Process chunks independently in one (common) thread pool

  3. **Combine** – Join partial results into a single result

    - Performed by terminal operations like collect() & reduce()

See [www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples](http://www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples)
Avoiding Concurrency Hazards in Java 8 Parallel Streams
Avoiding Concurrency Hazards in Java 8 Parallel Streams

- The Java 8 parallel streams framework assumes behaviors don’t incur race conditions

Race conditions arise when an app depends on the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition#Software
Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid behaviors with side-effects

See docs.oracle.com/javase/tutorial/collectionsSTREAMS/parallelism.html#side_effects
Parallel streams should therefore avoid behaviors with side-effects, e.g.

- **Stateful lambda expressions**
- Where results depend on shared mutable state

```java
class BuggyFactorial {
    static class Total {
        long mTotal = 1;
        void multiply(long n) {
            mTotal *= n;
        }
    }

    static long factorial(long n) {
        Total t = new Total();
        LongStream .rangeClosed(1, n) .parallel() .forEach(t::multiply);

        return t.mTotal;
    }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness](http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness)
Parallel streams should therefore avoid behaviors with side-effects, e.g.

- **Stateful lambda expressions**
  - Where results depend on shared mutable state
  - i.e., state that may change in parallel execution of a pipeline

```java
class BuggyFactorial {
    static class Total {
        long mTotal = 1;
        void multiply(long n) {
            mTotal *= n;
        }
    }
    static long factorial(long n) {
        Total t = new Total();
        LongStream.rangeClosed(1, n).parallel().forEach(t::multiply);
        return t.mTotal;
    }
}
```
Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid behaviors with side-effects, e.g.
  - **Stateful lambda expressions**
    - Where results depend on shared mutable state
    - i.e., state that may change in parallel execution of a pipeline

  Race conditions can arise due to the unsynchronized access to `mTotal` field

```java
class BuggyFactorial {
    static class Total {
        long mTotal = 1;
        void multiply(long n) {
            mTotal *= n;
        }
    }

    static long factorial(long n) {
        Total t = new Total();
        LongStream
            .rangeClosed(1, n)
            .parallel()
            .forEach(t::multiply);
        return t.mTotal;
    }
}
```

See [github.com/douglas craigschmidt/LiveLessons/tree/master/Java8/ex16](https://github.com/douglas craigschmidt/LiveLessons/tree/master/Java8/ex16)
Parallel streams should therefore avoid behaviors with side-effects, e.g.
- **Stateful lambda expressions**
- **Interference w/the data source**
  - Occurs when source of stream is modified within the pipeline

```java
List<Integer> list = IntStream
    .range(0, 10)
    .boxed()
    .collect(toList());

list
    .parallelStream()
    .peek(list::remove)
    .forEach(System.out::println);
```

See [docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference)
Parallel streams should therefore avoid behaviors with side-effects, e.g.

- *Stateful lambda expressions*
- *Interference w/the data source*
  - Occurs when source of stream is modified within the pipeline

```java
List<Integer> list = IntStream.range(0, 10)
    .boxed()
    .collect(toList());

list
    .parallelStream()
    .peek(list::remove)
    .forEach(System.out::println);
```

*Aggregate operations enable parallelism with non-thread-safe collections provided the collection is not modified while it's being operated on.*

Java 8 lambda expressions & method references containing no shared state are useful for parallel streams since they needn’t be explicitly synchronized.

```java
return new SearchResults
    (Thread.currentThread().getId(),
    currentCycle(), phrase, title,
    StreamSupport
        .stream(new PhraseMatchSpliterator
            (input, phrase),
            parallel)
        .collect(toList()));
```

```java
return mList.size() == 0;
```

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html
End of Overview of Java 8 Parallel Streams (Part 2)