How Java Parallel Streams Work “Under the Hood”

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

• Learn how parallel stream phases work “under the hood”

See developer.ibm.com/articles/j-java-streams-3-brian-goetz
Overview of How a Parallel Stream Works
A Java parallel stream implements a “map/reduce” variant optimized for multi-core processors.

See en.wikipedia.org/wiki/MapReduce
A Java 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
The split-apply-combine phases are:

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

Overview of How a Parallel Stream Works

• The split-apply-combine phases are:

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

   - **trySplit()**
   - Each chunk is an independent & “atomic” subset of the data source

See upcoming lesson on “Java Parallel Stream Internals: Partitioning”
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into “chunks”
   - Spliterators partition collections in Java

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

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

- Spliterators partition collections in Java

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

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance)
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

- Spliterators partition collections in Java

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

  Used only for parallel streams

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit)
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into “chunks”
     - Spliterators partition collections in Java
     - Each Java collection has a spliterator

```java
interface Collection<E> {
  ...  
  default Spliterator<E> spliterator() {
    return Spliterators.spliterator(this, 0);
  }

  default Stream<E> parallelStream() {
    return StreamSupport.stream(spliterator(), true);
  }
  ...  
}
```

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

1. **Split** – Recursively partition a data source into “chunks”
   - Spliterators partition collections in Java
   - Each Java collection has a spliterator
   - Programmers can define custom spliterators

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

1. **Split** – Recursively partition a data source into “chunks”
   - Spliterators partition collections in Java
   - Each Java collection has a spliterator
   - Programmers can 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 “chunks”

  2. **Apply** – Process chunks in a common thread pool

See lesson on “Java Parallel Stream Internals: Parallel Processing via the Common ForkJoinPool”
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool

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

- The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool
   - Utilization’s maximized via “work-stealing”

See lesson on “Java Parallel Stream Internals: Mapping onto the Common ForkJoinPool”
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool
   - Utilization’s maximized via “work-stealing”
   - Programmers can control # of threads in the pool

See lesson on “Java Parallel Stream Internals: Configuring the Common Fork-Join Pool”
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool

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

See upcoming lessons on “Java Parallel Stream Internals: Combining Results”
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool

3. **Combine** – Join partial results to a single result
   - Performed by terminal operations
   - e.g., 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)
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:
  
  1. **Split** – Recursively partition a data source into “chunks”
  
  2. **Apply** – Process chunks in a common thread pool
  
  3. **Combine** – Join partial results to a single result
     
     - Performed by terminal operations
     
     - Collectors can either be
       
       - Concurrent – synchronized
       
       - Non-concurrent – non-synchronized

See lessons on “Java Parallel Stream Internals: Non-.Concurrent & Concurrent Collectors”
Overview of How a Parallel Stream Works

• The split-apply-combine phases are:

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

2. **Apply** – Process chunks in a common thread pool

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

• Performed by terminal operations

• Collectors can either be
  • Concurrent – synchronized
  • Non-concurrent – non-synchronized

Programmers can define custom collectors
End of How Java Parallel Streams Work “Under the Hood”