Java 8 Parallel Streams Internals

(Part 1)

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

- Understand parallel stream internals

See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz
Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
- Know what can change & what can’t

See en.wikipedia.org/wiki/Serenity_Prayer
Why Knowledge of Parallel Streams Matters
Why Knowledge of Parallel Streams Matters

- Converting a Java 8 sequential stream to a parallel stream is usually quite straightforward

```java
List<List<SearchResults>>
processStream() {
    return getInput().stream()
        .map(this::processInput)
        .collect(toList());
}
```

vs

```java
List<List<SearchResults>>
processStream() {
    return getInput().parallelStream()
        .map(this::processInput)
        .collect(toList());
}
```

See "Java 8 SearchWithParallelStreams Example"
Why Knowledge of Parallel Streams Matters

- However, knowledge of parallel streams internals will make you a better Java 8 streams programmer!

When performance is critical, it's important to understand how streams work internally

See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz
Why Knowledge of Parallel Streams Matters

• Recall the 3 phases of a Java 8 parallel stream

  - Stream factory operation ()
  - Intermediate operation (behavior f)
  - Intermediate operation (behavior g)
  - Terminal operation (reducer)

See docs.oracle.com/javase/tutorial/collections_streams/parallelism.html
Why Knowledge of Parallel Streams Matters

• Recall the 3 phases of a Java 8 parallel stream
  • *Split* – Uses a spliterator to partition stream elements into multiple chunks
Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of a Java 8 parallel stream:
  - **Split** – Uses a splitter to partition stream elements into multiple chunks.
  - **Apply** – Independently processes these chunks in the common fork-join pool.
Why Knowledge of Parallel Streams Matters

• Recall the 3 phases of a Java 8 parallel stream
  • *Split* – Uses a spliterator to partition stream elements into multiple chunks
  • *Apply* – Independently processes these chunks in the common fork-join pool
  • *Combine* – Joins partial sub-results into a single result
Why Knowledge of Parallel Streams Matters

- Recall the 3 phases of a Java 8 parallel stream
  - **Split** – Uses a spliterator to partition stream elements into multiple chunks
  - **Apply** – Independently processes these chunks in the common fork-join pool
  - **Combine** – Joins partial sub-results into a single result

It’s important to which of these phases you can control & which you can’t!
Parallel Stream Splitting & Thread Pool Mechanisms
A parallel stream’s splitting & thread pool mechanisms are often invisible.

Stream factory operation ()

Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (behavior h)
Parallel Stream Splitting & Thread Pool Mechanisms

• A parallel stream’s splitting & thread pool mechanisms are often invisible, e.g.
• Java collections have predefined spliterators

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

See blog.logentries.com/2015/10/java-8-introduction-to-parallelism-and-spliterator
A parallel stream’s splitting & thread pool mechanisms are often invisible, e.g.

- Java collections have predefined spliterators
- The common fork-join pool is used by default

See [www.baeldung.com/java-fork-join](http://www.baeldung.com/java-fork-join)
Parallel Stream Splitting & Thread Pool Mechanisms

• However, programmers can customize the behavior of splitting & thread pools

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

```java
public interface ManagedBlocker {
    boolean block() throws InterruptedException;
    boolean isReleasable();
}
```

See Parts 2 & 4 of this lesson on "Java 8 Parallel Stream Internals"
Parallel Stream Ordering
The order in which chunks are processed is non-deterministic.

The ordering can exhibit different behaviors on different runs even for the same input.

See [en.wikipedia.org/wiki/Nondeterministic_algorithm](en.wikipedia.org/wiki/Nondeterministic_algorithm)
Parallel Stream Ordering

- The order in which chunks are processed is non-deterministic
- Programmers have little/no control over how chunks are processed
Parallel Stream Ordering

- The *order* in which chunks are processed is non-deterministic
- Programmers have little/no control over how chunks are processed
- Non-determinism is useful since it enables optimizations at multiple layers!

Additional Frameworks & Languages

Java Execution Environment (e.g., JVM)

System Libraries

Operating System Kernel

Operating System Kernel

Applications

Threading & Synchronization Packages

e.g., scheduling & execution of tasks via fork-join pool, JVM, hardware cores, etc.
The results of the processing are more deterministic.

Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)

EO is order in which the stream source makes its elements available

Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order

> It doesn’t matter whether the stream is parallel or sequential

Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”

```java
List<Integer> list = Arrays.asList(1, 2, ...);
Integer[] doubledList = list.parallelStream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .toArray(Integer[]::new);
```

The encounter order is [1, 2, 3, 4, …] since list is ordered
Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in "encounter order" (EO)
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”

```java
List<Integer> list = Arrays.asList(1, 2, ...);

Integer[] doubledList = list
    .parallelStream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .toArray(Integer[]::new);
```

The result *must* be [2, 4, ...]

See [github.com/douglas craig schmidt/LiveLessons/tree/master/Java8/ex21]
Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
    - Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”
  - Unordered collections don’t need to respect “encounter order”

```java
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, ...));

Integer[] doubledSet = set.parallelStream()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .toArray(Integer[]::new);
```

*A HashSet is unordered*

Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Ordered spliterators, ordered collections, & static stream factory methods respect “encounter order”
- Unordered collections don’t need to respect “encounter order”

```java
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, ...));
Integer[] doubledSet = set.parallelStream() .filter(x -> x % 2 == 0) .map(x -> x * 2) .toArray(Integer[]::new);
```

This code runs faster since encounter order need not be maintained

• The *results* of the processing are more deterministic

• Programmers can control if results are presented in “encounter order” (EO)
  • Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order

• Certain intermediate operations affect ordering behavior
Parallel Stream Ordering

- The *results* of the processing are more deterministic
  - Programmers can control if results are presented in “encounter order” (EO)
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```java
List<Integer> list = Arrays.asList(1, 2, ...);
Integer[] doubledList = list
    .parallelStream()
    .distinct()  // The result must be [2, 4, ...], but the code is slow due to limit() & distinct() “stateful” semantics in parallel streams
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[] ::new);
```
Parallel Stream Ordering

- The **results** of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

List<Integer> list = Arrays.asList(1, 2, ...);

Integer[] doubledList = list
    .parallelStream()
    .unordered()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);

*This code runs faster since stream is unordered & thus limit() & distinct() incur less overhead*

The results of the processing are more deterministic.

- Programmers can control if results are presented in “encounter order” (EO).
  - Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order.
  - Certain intermediate operations affect ordering behavior.
  - Certain terminal operations also affect ordering behavior.

Parallel Stream Ordering

Stream factory operation ()

Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (reducer)
The results of the processing are more deterministic.

Programmers can control if results are presented in “encounter order” (EO).

Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order.

Certain intermediate operations affect ordering behavior.

Certain terminal operations also affect ordering behavior.

e.g., forEachOrdered() & forEach()

```java
List<Integer> list = Arrays.asList(1, 2, ...);
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEachOrdered(queue::add);
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex21](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex21)
Parallel Stream Ordering

- The *results* of the processing are more deterministic
- Programmers can control if results are presented in “encounter order” (EO)
- Order is maintained if the source is ordered & the aggregate operations used are obliged to maintain order
- Certain intermediate operations affect ordering behavior
- Certain terminal operations also affect ordering behavior
- e.g., forEachOrdered() & forEach()

```java
List<Integer> list = Arrays.asList(1, 2, ...);
ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEach(queue::add);  // Unordered
```

End of Java 8 Parallel Stream Internals (Part 1)