Understand Java Parallel Streams Internals: Order of Results for Operations

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, e.g.
  • Know what can change & what can’t
    • Splitting, combining, & pooling mechanisms
  • Order of processing
  • Order of results
    • Overview
    • Collections that affect results order
    • Operations that affect results order
Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
- Know what can change & what can’t
  - Splitting, combining, & pooling mechanisms
  - Order of processing
  - Order of results
  - Overview
  - Collections that affect results order
  - Operations that affect results order

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

Multiple examples are analyzed in detail

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex21
Intermediate Operations that Affect Results Order
Certain intermediate operations affect ordering behavior.
Intermediate Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

See developer.ibm.com/languages/java/articles/j-java-streams-3-brian-goetz
Again, recall that “ordered” isn’t the same as “sorted”!

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

*The encounter order is [2, 3, 1, 4, 2] since list is ordered & non-unique*
Intermediate Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

- Remove duplicate elements from the stream (a stateful intermediate operation)

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct)
Intermediate Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
- e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#limit
Intermediate Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

```
The result must be [4, 8], but the code is slow due to limit() & distinct() "stateful” semantics in parallel streams
```
Certain intermediate operations affect ordering behavior

- e.g., sorted(), unordered(), skip(), & limit()

List<Integer> list = Arrays.asList(2, 3, 1, 4, 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, so therefore limit() & distinct() incur less overhead

See docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered
Intermediate Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

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

Since encounter order needn’t be maintained the results could either be [8, 4] or [4, 8]

See developer.ibm.com/languages/java/articles/j-java-streams-3-brian-goetz
Terminal Operations that Affect Results Order
Terminal Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior

```
Stream factory operation ()

Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (reducer)
```
Terminal Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - `forEachOrdered()`

The encounter order is [2, 3, 1, 4, 2] since list is ordered & non-unique.

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

List<Integer> results =
    new ArrayList<>();

list.
    parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .forEachOrdered(results::add);
```
Terminal Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - `forEachOrdered()`

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);
List<Integer> results = new ArrayList<>();
list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEachOrdered(results::add);
```

*This list supports unsynchronized insertions & removals of elements*
Terminal Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - forEachOrdered()

```java
List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);
List<Integer> results = new ArrayList<>();
list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEachOrdered(results::add);
```

Results must appear in encounter order, but may be slow due to implicit synchronization in forEachOrdered()

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEachOrdered
Certain terminal operations also affect ordering behavior, e.g.
- `forEachOrdered()`
- `forEach()`

List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

ConcurrentLinkedQueue<Integer> results = new ConcurrentLinkedQueue<>();

list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEach(results::add);

Results need not appear in the encounter order, but may be faster since `forEach()` isn’t synchronized.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach
Certain terminal operations also affect
ordering behavior, e.g.

- forEachOrdered()
- forEach()

Terminal Operations that Affect Results Order

List<Integer> list = Arrays.asList(2, 3, 1, 4, 2);

ConcurrentLinkedQueue<Integer> results = new ConcurrentLinkedQueue<>();

list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEach
  (results::add);

However, this collection must support
thread-safe insertions & removals!!

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html
End of Understand Java Parallel Streams Internals: Order of Results for Operations