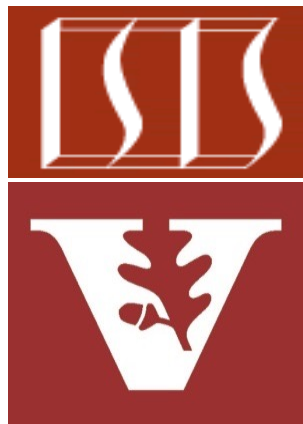


# Java Parallel Streams Internals: Order of Results for Aggregate Operations

**Douglas C. Schmidt**

**[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)**

**[www.dre.vanderbilt.edu/~schmidt](http://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
    - How collections affect results order
    - How aggregate operations affect results order



# Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g. 

```
List<Integer> list = Arrays.asList(1, 2, ...);
```
  - Know what can change & what can't
    - Splitting, combining, & pooling mechanisms
    - Order of processing
    - Order of results
      - Overview
      - How collections affect results order
      - How aggregate operations affect results order
- ```
Integer[] doubledArray = list  
.parallelStream()  
.distinct()  
.filter(x -> x % 2 == 0)  
.map(x -> x * 2)  
.limit(sOutputLimit)  
.toArray(Integer[]::new);
```

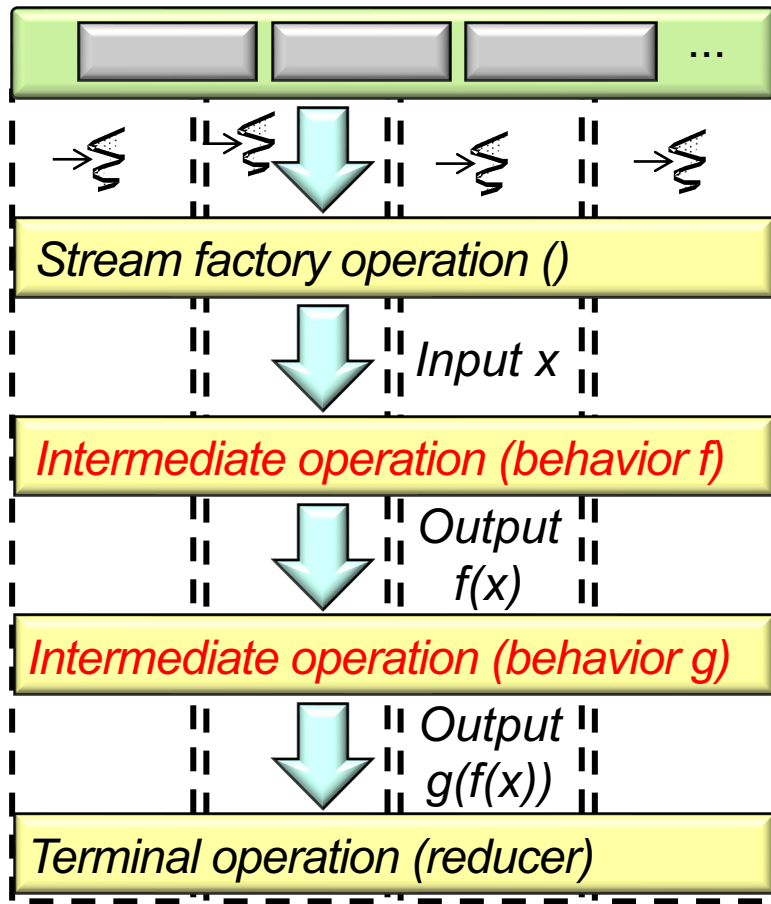
*Multiple examples are analyzed in detail*

---

# Intermediate Operations that Affect Results Order

# 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()`

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

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

# 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);
```

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

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

Again, recall that "ordered" isn't the same as "sorted"!

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

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

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);
```

*Only process `sOutputLimit` elements in the stream (a stateful intermediate operation)*

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

# 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[] doubledArray = 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*

# 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[] doubledArray = 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*

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

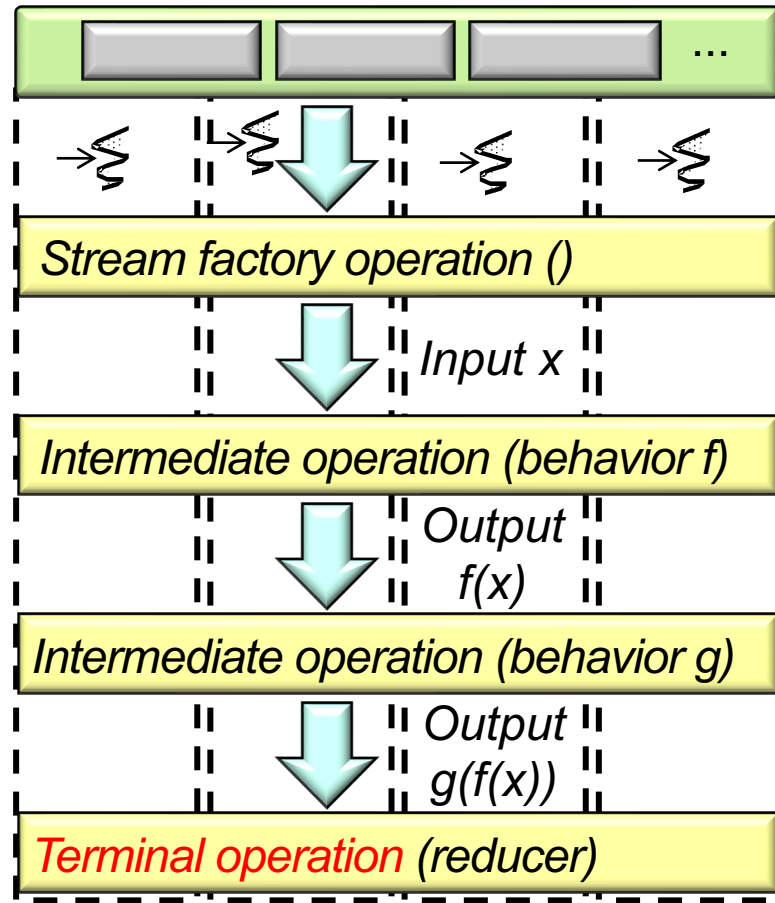
*Since encounter order needn't be maintained the results could either be [8, 4] or [4, 8]*

---

# Terminal Operations that Affect Results Order

# Terminal Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior



# 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.*

```
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()`

*This list supports unsynchronized insertions & removals of elements*

```
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()`

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

```
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()`
  - `forEach()`

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

```
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);
```

# Terminal Operations that Affect Results Order

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

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

```
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);
```

# Terminal Operations that Affect Results Order

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

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

```
List<Integer> results = list  
    .parallelStream()  
    .distinct()  
    .filter(x -> x % 2 == 0)  
    .map(x -> x * 2)  
    .limit(sOutputLimit)  
    .collect(toList());
```

*It's almost always better to use a terminal operation that produces a result, such as `collect()` or `reduce()`*

---

# End of Java Parallel Streams Internals: Order of Results for Aggregate Operations