Java Parallel Streams Internals: Combining Results (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

- Understand parallel stream internals, e.g.
  - Know what can change & what can’t
  - Partition a data source into “chunks”
  - Process chunks in parallel via the common fork-join pool
  - Configure the Java parallel stream common fork-join pool
- Perform a reduction to combine partial results into a single result
  - Be aware of common traps & pitfalls with parallel streams
Differences for collect() & reduce() in a Parallel Stream
Differences for collect() & reduce() in a Parallel Stream

• It’s important to understand the semantic differences between collect() & reduce()
Differences for collect() & reduce() in a Parallel Stream

- It’s important to understand the semantic differences between collect() & reduce(), e.g.
- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce3 (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream.reduce(new StringBuilder(),
        StringBuilder::append,
        StringBuilder::append)
        .toString();
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
Differences for collect() & reduce() in a Parallel Stream

- It’s important to understand the semantic differences between collect() & reduce(), e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce3
      (boolean parallel) {
      ...
        Stream<String> wordStream =
            allWords.stream();

        if (parallel)
            wordStream.parallel();

        String words = wordStream
            .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append)
            .toString();
```

**Convert a list of words into a stream of words**

Naturally, this call doesn’t really do any work since streams are “lazy”
It's important to understand the semantic differences between `collect()` & `reduce()`, e.g.

- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions.

```java
void buggyStreamReduce3 (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream .reduce(new StringBuilder(),
                                       StringBuilder::append,
                                       StringBuilder::append) .toString();
}
```

*A stream can be dynamically switched to "parallel" mode!*

See docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#parallel
It’s important to understand the semantic differences between collect() & reduce(), e.g.

- Always test with a parallel stream to detect mistakes with mutable vs. immutable reductions

```java
void buggyStreamReduce3
    (boolean parallel) {
        ...
    Stream<String> wordStream =
        allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream
        .sequential()
        .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append)
        .toString();
```

The “last” call to .parallel() or .sequential() in a stream “wins”

See mail.openjdk.java.net/pipermail/lambda-libs-spec-experts/2013-March/001504.html
Differences for collect() & reduce() in a Parallel Stream

- It’s important to understand the semantic differences between collect() & reduce(), e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce3(
    boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel) wordStream.parallel();
    String words = wordStream
        .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append)
        .toString();
}
```

This code works when parallel is false since the StringBuilder is only called in a single thread

See [docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)
Differences for collect() & reduce() in a Parallel Stream

- It’s important to understand the semantic differences between `collect()` & `reduce()`, e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce3
    (boolean parallel) {
    ...
    Stream<String> wordStream =
        allWords.stream();

    if (parallel)
        wordStream.parallel();

    String words = wordStream
        .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append)
        .toString();
```

*This code fails when parallel is true since reduce() expects to do an “immutable” reduction*
It’s important to understand the semantic differences between `collect()` & `reduce()`, e.g.

- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```java
void buggyStreamReduce3
    (boolean parallel) {
    ...
    Stream<String> wordStream =
        allWords.stream();

    if (parallel)
        wordStream.parallel();

    String words = wordStream
        .reduce(new StringBuilder(),
            StringBuilder::append, StringBuilder::append)
        .toString();
```

There’s a race condition here since `StringBuilder` is not thread-safe..

See [www.baeldung.com/java-string-builder-string-buffer](http://www.baeldung.com/java-string-builder-string-buffer)
It’s important to understand the semantic differences between `collect()` & `reduce()`, e.g.

- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- One solution use `reduce()` with string concatenation

```java
void streamReduceConcat(boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream
        .reduce(new String(),
        (x, y) -> x + y);
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
Differences for `collect()` & `reduce()` in a Parallel Stream

- It’s important to understand the semantic differences between `collect()` & `reduce()`, e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- One solution use `reduce()` with string concatenation

```java
void streamReduceConcat (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream
        .reduce(new String(),
            (x, y) -> x + y);
}
```

This simple fix is inefficient due to string concatenation overhead

See javarevisited.blogspot.com/2015/01/3-examples-to-concatenate-string-in-java.html
Differences for collect() & reduce() in a Parallel Stream

- It’s important to understand the semantic differences between collect() & reduce(), e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- One solution use reduce() with string concatenation
- Another solution uses collect() with the joining collector

```java
void streamCollectJoining (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream .collect(joining());
}
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex17](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex17)
It’s important to understand the semantic differences between `collect()` & `reduce()`, e.g.

- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  - One solution uses `reduce()` with string concatenation
  - Another solution uses `collect()` with the joining collector

```java
void streamCollectJoining (boolean parallel) {
    ...
    Stream<String> wordStream = allWords.stream();
    if (parallel)
        wordStream.parallel();
    String words = wordStream
        .collect(joining());
}
```

*This is a much better solution!!*
Differences for collect() & reduce() in a Parallel Stream

- Also beware of issues related to associativity & identity with reduce()

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
                (x, y) -> x - y);
}

void testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                 // Could use (x, y) -> x + y
                 Math::addExact);
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
Differences for collect() & reduce() in a Parallel Stream

- Also beware of issues related to associativity & identity with reduce()

```java
testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
            (x, y) -> x - y);
}
```

```java
testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
            // Could use (x, y) -> x + y
            Math::addExact);
}
```

This code fails for a parallel stream since subtraction is not associative

See developer.ibm.com/articles/j-java-streams-2-brian-goetz
Differences for collect() & reduce() in a Parallel Stream

- Also beware of issues related to associativity & identity with reduce()

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
                (x, y) -> x - y);
}

void testSum(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                // Could use (x, y) -> x + y
                Math::addExact);
}
```

This code fails if identity is not 0L

The “identity” of an OP is defined as “identity OP value == value” (& inverse)
• Also beware of issues related to associativity & identity with reduce()

```java
void testDifferenceReduce(...) {
    long difference = LongStream
        .rangeClosed(1, 100)
        .parallel()
        .reduce(0L,
                (x, y) -> x - y);
}
```

```java
void testProd(long identity, ...) {
    long sum = LongStream
        .rangeClosed(1, 100)
        .reduce(identity,
                 (x, y) -> x * y);
}
```

This code fails if identity is not 1L
Differences for `collect()` & `reduce()` in a Parallel Stream

- More good discussions about `reduce()` vs. `collect()` appear online

See [www.youtube.com/watch?v=oWIWEKNM5Aw](http://www.youtube.com/watch?v=oWIWEKNM5Aw)
End of Java Parallel Streams Internals: Combining Results (Part 2)