Java 8 Parallel Stream Internals

(Part 5)

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
  - Configure the Java 8 parallel stream common fork-join pool
  - Avoid pool starvation & improve performance w/ManagedBlocker
- Perform a reduction that combines partial results into a single result

Combining Results in a Parallel Stream
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result.

This discussion assumes a non-concurrent collector (more discussions follow).
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result.
- `join()` occurs in a single thread at each level.
  - i.e., the “parent”
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result
- `join()` occurs in a single thread at each level
  - i.e., the “parent”

As a result, there’s typically no need for synchronizers during the joining
Different terminal operations combine partial results in different ways.

Understanding these differences is particularly important for parallel streams.
Combining Results in a Parallel Stream

• Different terminal operations combine partial results in different ways, e.g.
  • reduce() creates a new immutable value

See docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value

```java
long factorial(long n) {
  return LongStream
      .rangeClosed(1, n)
      .parallel()
      .reduce(1, (a, b) -> a * b,
             (a, b) -> a * b);
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16
long factorial(long n) {
    return LongStream
        .rangeClosed(1, n)
        .parallel()
        .reduce(1, (a, b) -> a * b,
               (a, b) -> a * b);
}

• Different terminal operations combine partial results in different ways, e.g.
• reduce() creates a new immutable value

reduce() combines two immutable values (e.g., long or Long) & produces a new one
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

See greenteapress.com/thinkapjava/html/thinkjava011.html
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value
  - collect() mutates an existing value

Set<CharSequence>
  uniqueWords =
  getInput(sSHAKESPEARE, "\s+")
  .parallelStream()
  ...
  .collect(toCollection(TreeSet::new));

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - `reduce()` creates a new immutable value
  - `collect()` mutates an existing value

```
Set<CharSequence>
uniqueWords =
getInput(sSHAKESPEARE),
"\\s+")
.parallelStream()
...
.collect(toCollection(TreeSet::new));
```

- `collect()` mutates a container to accumulate the result it’s producing
Set<CharSequence> uniqueWords = getInput(sSHAKESPEARE, "\s+")
    .parallelStream()
    ...
    .collect(ConcurrentHashSetCollector.toSet());

Concurrent collectors (covered later) are different than non-concurrent collectors.
Combining Results in a Parallel Stream

- More discussion about `reduce()` vs. `collect()` appears online

See [www.youtube.com/watch?v=oWIWEKNM5Aw](http://www.youtube.com/watch?v=oWIWEKNM5Aw)
More discussion about `reduce()` vs. `collect()` appears online, 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](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17)
More discussion about `reduce()` vs. `collect()` appears online, 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
        .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
        .toString();
```

Naturally, this call doesn’t really do any work since streams are “lazy”
Combining Results in a Parallel Stream

- More discussion about `reduce()` vs. `collect()` appears online, 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!
```
Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, 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.sequential()
        .reduce(new StringBuilder(),
                StringBuilder::append,
                StringBuilder::append).
        toString();

    See mail.openjdk.java.net/pipermail/lambda-libs-spec-experts/2013-March/001504.html
```

The “last” call to .parallel() or .sequential() in a stream “wins”
More discussion about `reduce()` vs. `collect()` appears online, 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();
}
```

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
Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, 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
        .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
        .toString();
```

*This code fails when parallel is true since reduce() expects to do an “immutable” reduction*
More discussion about reduce() vs. collect() appears online, 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 are race conditions here since there's just one shared StringBuilder, which is not properly thread-safe..

See [www.baeldung.com/java-string-builder-string-buffer](http://www.baeldung.com/java-string-builder-string-buffer)
More discussion about reduce() vs. collect() appears online, 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 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
More discussion about reduce() vs. collect() appears online, e.g. Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```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!!

See [www.mkyong.com/java8/java-8-stringjoiner-example](http://www.mkyong.com/java8/java-8-stringjoiner-example)
Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
- Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- Beware of issues related to associativity & identity

```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)
Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  - Beware of issues related to associativity & identity

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

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

Combining Results in a Parallel Stream

- More discussion about `reduce()` vs. `collect()` appears online, e.g.
- Always test with a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- Beware of issues related to associativity & identity

```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)
End of Java 8 Parallel Stream Internals (Part 5)