Java 8 Parallel Streams

Ethan Williams

February 28, 2018

Contents

0 Prerequisites

0.1 Technical

This document was written for Java developers who have an interest in using concurrency in streams, and assumes knowledge of serial streams and lambda expressions. Developers in other languages with similar mechanisms such as C# with `Linq` may also find the topics useful with the understanding that syntax, implementation, and functionality will differ.

Additionally, functional knowledge of `java.util.concurrent` and the `Consumer` interface will help in gaining a more practical knowledge but is not required.

0.2 Vocabulary Clarification

Some of the vocabulary in the paper may be unfamiliar to those with a knowledge of streams and are defined/clarified below:

- A stream is composed of 3 parts:
  - the `Collection` the stream is built on is the *source*
  - *intermediate operations* such as `map()` transforms but doesn't serialize the stream (see below for serialization definition)
  - *terminal operations* like `toArray()` serialize the stream
- A stream instantiated with the `stream()` method only is referred to as a *serial stream*
- A stream instantiated with the `parallel().stream()` or `parallelStream()` method is referred to as a *parallel stream*
- *Serializing data* or *serializing a stream* is the final step in a stream when references to the source are ditched and the new data structure is instantiated
- *Stateful expressions* are lambda expressions passed as arguments to an intermediate operation which depends on the ordering of the input elements

1 Introduction

Parallel streams were introduced into Java 8 alongside serial streams so that developers could utilize concurrency in order to more efficiently utilize modern multiprocessor design [2]. Making a serial stream into a parallel stream is as easy as calling `parallel()` after `stream()`. In order for the `parallel()` method to be applicable on the stream, the source `Collection` must have an implementation of a `Spliterator` [2]. A `Spliterator` object breaks up the original stream into parts which are each handled by a new thread from the JVM's common pool, illustrated in Figure 1. The substreams are then assembled and computed based on the bahavior of the terminal operation.

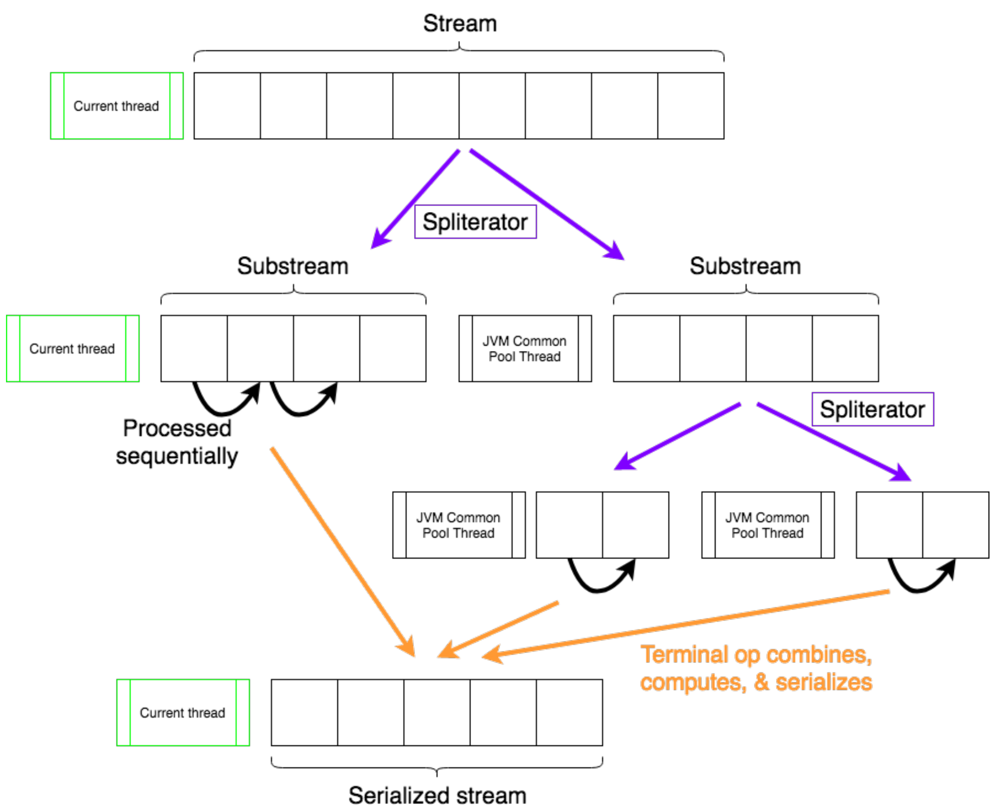

Figure 1: Overview of a parallel stream
Source: Ethan Williams

There are several terminal operations for a parallel stream, but the `collect()` method is the most reliable form of reduction. The method uses a `Collector` to reduce the stream, an object that defines how input elements should be added to a given data structure. This document will focus on the `Collector`'s functionality and implementation, finishing with an explanation of special use cases for parallel streams.

Despite parallel streams being introduced into Java to simplify concurrency, developers can easily corrupt data and cause system bugs. All possible bugs in streams derive from developers not following standard concurrency practices such as using long-running or blocking operations in streams. Additionally, considerations have to be taken with streams specifically to avoid interference with the source and using stateful expressions.

2 Spliterator

The `Spliterator` is the backbone of parallel streams, allowing the program to split a collection apart (illustrated in Figure 2) and iterate through it. If a class extending a `Collection` does not have a `spliterator()` method returning a `Spliterator` object, then Java is not able to process the collection with a parallel stream at all. Figure 2 shows how an instance may behave when it splits itself. It is worth noting that the splitting doesn't actually break up the collection. `Spliterator`s share the collection and simply keep track of what element it is currently iterating on (index) and one more than the last element it is allowed to execute with (fence).

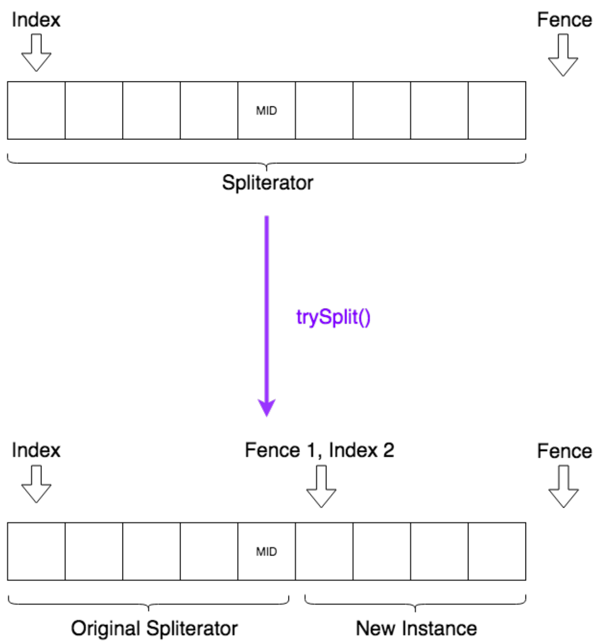

Figure 2: A `Spliterator` Before and after splitting
Source: Ethan Williams

2.1 Implementation

A `Spliterator` must be able to traverse and split the portion of the stream it represents. `tryEachRemaining()` and `tryNext()` are the two methods which dictate how traversal is handled for the collection. `tryEachRemaining()` in Figure 3 takes a Consumer object which is the operation to be executed on each element of the collection. The example simply iterates through each element and uses it as a parameter to the `accept()` method of the `Consumer` object. `tryNext()` in Figure 4 is similar although the operation is only attempted on element at the current cursor position. If that cursor position is past the fence of the Spliterator, then the method returns false, otherwise it returns true.

```java
public void forEachRemaining(Consumer<? super E>
   action) {
        for (int i = index; i < fence; ++i)
            action.accept((E) list.elementData[i]);
}
```

Figure 3: Implementation of `forEachRemaining()`
Source: Java ArrayList, modified by Ethan Williams

```java
public boolean tryAdvance(Consumer<? super E>
   action) {
     if(i == fence) return false;
     index++;
     action.accept((E) list.elementData[i]);
     return true;
}
```

Figure 4: Implementation of `tryAdvance()`
Source: Java ArrayList, modified by Ethan Williams

A `Spliterator`'s primary functionality is encapsulated within the `trySplit()` method in Figure 5. This method is called when the JVM wants to break the source collection in order to start processing the stream on another thread and if implemented incorrectly can be a subtle but important error in an application []. The example implementation simply finds the midpoint and either returns a new `Spliterator` from the cursor to the midpoint and the current instance of `Spliterator` now covers mid to the fence. The example `trySplit()` method is the code behind the split behavior illustrated in Figure 2.

```java
public Spliterator<E> trySplit() {
    int mid = (index + fence) >>> 1;
    return (index >= mid) ? null : new
       Spliterator<E>(list, index, index = mid);
}
```

Figure 5: Implementation of `trySplit()`
Source: Java ArrayList, modified by Ethan Williams

3 Collector

A `Collector` object defines a mutable reduction operation for a group of input elements. In other words, it provides information on how to instantiate a data structure and perform several operations on it. Its functionality is encompassed in 4 methods (illustrated in Figure 6):

- `supplier()` provides information on how to construct a new instance of the desired data structure
- `accumulator()` details how to add any given element to the data structure
- `combine()` directs how to assemble multiple instances of the data structure
- `finisher()` simply serializes the stream, completing the reduction

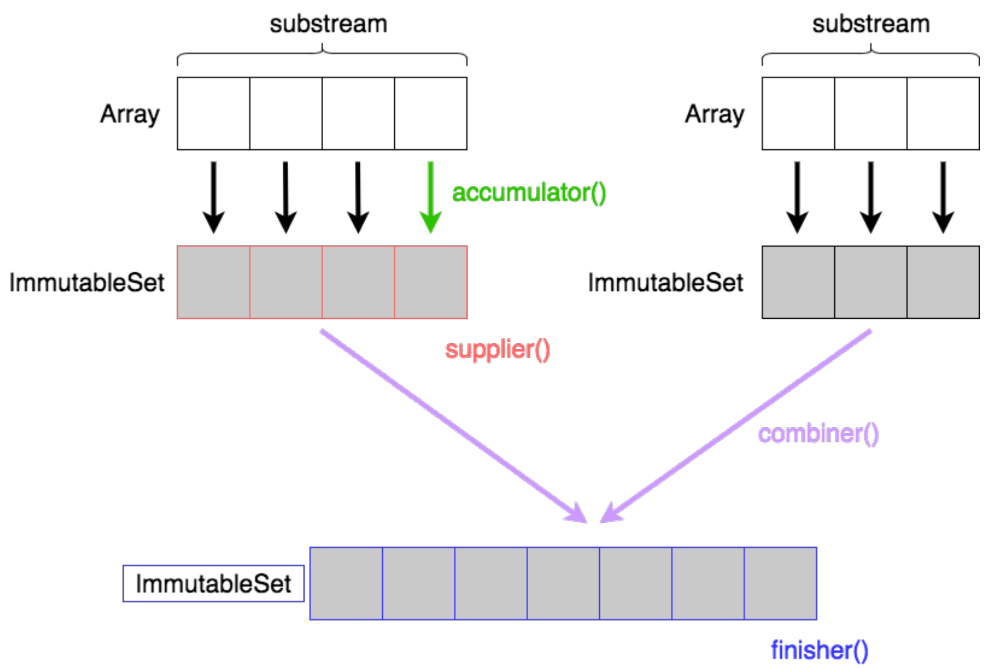

Figure 6: How a `Collector` is used
Source: Ethan Williams

`Collector` objects are used during a reduction operation to join substreams together in the form of the data structure defined by the instance. Java has a static class called `Collectors` which provide basic reduction instances via method calls. For example `Collectors.toMap()` returns a `Collector` instance which reduces the stream to a `Map`.

3.1 Implementation

A `Collector` object has four methods which comprise the majority of its functionality: `supplier()`, `accumulator()`, `combiner()`, and `finisher()` [1]. The examples below are a `Collector` which will yield an `ImmutableSet` when used.

The `supplier()` method returns a mechanism to build an instance of a mutable data structure that will hold the elements of the stream called an *accumulator* [6], in our code example this is a builder for `ImmutableSet`.

```java
public Supplier<ImmutableSet.Builder<T>> supplier()
   {
    return ImmutableSet::builder;
}
```

Figure 7: Implementation of `supplier()`
Source: [6]

The `accumulator()` method takes an accumulator and an element as parameters and will return a `Consumer` object which details how the element should be added to the accumulator. In the example a `BiConsumer` object will add the element to the given `ImmutableSet`.

```java
public BiConsumer<ImmutableSet.Builder<T>, T>
   accumulator() {
    return (builder, t) -> builder.add(t);
}
```

Figure 8: Implementation of `accumulator()`
Source: [6]

The `combiner()` method details the logic on how two accumulators should be joined together and is used to assemble all instances of the new data structure that were derived from substreams. In the code example the `combiner()`'s behavior is that when two `ImmutableSet`s are combined, one is simply appended to the other.

```java
public BinaryOperator<ImmutableSet.Builder<T>>
   combiner() {
    return (left, right) -> {
        left.addAll(right.build());
        return left;
    };
}
```

Figure 9: Implementation of `combiner()`
Source: [6]

Finally, the `finisher()` method serializes the accumulator that is the result of combining all the substreams, completing the reduction. In the code example, this is as easy as returning the `build()` method which serializes the `ImmutableSet`, completing the conversion of the source.

```
public Function<ImmutableSet.Builder<T>,
   ImmutableSet<T>> finisher() {
     return ImmutableSet.Builder::build;
}
```

Figure 10: Implementation of `finisher()`
Source: [6]

3.2 Special Uses for Parallelism

Although the `collect()` method can be used on both serial and parallel streams, Java also includes special `Collector` instances for better parallel performance. For example, Figure 11 has the same collect operation on a parallel stream, one reduces with the `groupingBy()` method and the other reduces with the `groupingByConcurrent()` method. Even though both are parallel, the second example runs significantly faster because the reduction can be parallelized by using a `ConcurrentMap`. The first example processes each element in parallel, but the reduction is to a non-thread-safe data structure, limiting the potential benefits of parallelism.

```java
// Example 1
ConcurrentHashMap<Department, List<Employee>> byDept
    = employees.stream()
               .parallel()
               .collect(Collectors
                  .groupingBy(Employee::getDepartment)
                );

// Example 2
Map<Department, List<Employee>> byDept
    = employees.stream()
               .parallel()
               .collect(Collectors
                  .groupingByConcurrent(Employee::getDepartment)
                );
```

Figure 11: Reduction of employees into map by a `Collector`
Source: Ethan Williams

4 Practical Considerations when Using Parallel Streams

Parallel streams were introduced to make implementing parallelism in a Java application easier, but with this ease comes common pitfalls that arise from the abstraction. For example, a common mistake is using long-running or blocking operations in a stream which is a bad concurrency practice to begin with. Additionally, many developers aren't familiar with stream-specific concurrency practices which can lead to bugs. These are known as interference and use of stateful expressions and are extremely difficult to debug.

4.1 Long-Running/Blocking Operations

Using long-running or blocking operations in a stream will degrade performance drastically, a result of how streams implement threading. The JVM begins by processing on the calling thread and as more subtasks are broken off, the JVM gets threads from `ForkJoinPool.common()`, which is a thread pool used in the background of the whole application [4].

With the JVM's use of a common thread pool, a long-running operation in a parallel stream as shown in the first example of Figure 12 will degrade performance drastically. Performance will be such that the stream will benefit very little from parallelization and may even be less performant than a serial stream. Each thread in the pool will be consumed executing that operation and subsequently all other JVM tasks using the common thread pool have to wait.

In a situation where several streams are all attempting to process in parallel, each stream's performance will suffer even if it should have no problem. Java does allow a custom `ThreadPool` like in example 2 of Figure 12 which can help by using threads created outside the common pool [5]. Unfortunately, since the issue described above results from a bad concurrency practice, more threads won't provide any significant performance enhancement.

```
// Example 1
Optional<String> result =
   collection.stream().parallel().map((base) ->
   longOperation(argument)).findAny();

// Example 2
ForkJoinPool customPool = new ForkJoinPool(4);
Optional<String> result = customPool.submit(() ->
   collection.stream().parallel().map((arg) ->
   longOperation(arg)).findAny()).get();
```

Figure 12: Reduction of employees into map by a `Collector`
Source: Ethan Williams

4.2 Interference

Since streams don't contain any of the elements of the collection, instead storing references, if the source is modified then the reference is invalidated [2]. Editing the source of the stream in an intermediate operation is called *interference* and will throw a `ConcurrentModificationException` [7]. The stream in figure 13 attempts to add each element to the collection again using the `map()` method. Since it attempts to modify the source before the stream has serialized, this operation will throw an exception.

```
collection.stream().parallel().map((x) ->
   collection.add(x)).toArray();
```

Figure 13: A stream which causes interference and will throw an error
Source: Ethan Williams

4.3 Stateful Expressions

The third practice which will cause errors in a stream is using stateful expressions, operations that depend on the ordering of the elements [3]. The code in Figure 14 shows an example of a stateful operation while attempting to add elements to `parallelStorage` and print them. Although the `forEachOrdered()` method is just fine and will print in the expected order, `parallelStorage` will have a different ordering every time the stream is executed. The addition is stateful, meaning it depends on ordering and in parallel streams ordering can't be guaranteed in intermediate operations [7].

```
List<String> parallelStorage =
   Collections.synchronizedList(new ArrayList<>());
collection.stream().parallel().map(x ->
   parallelSotrage.add(x)).forEachOrdered(x ->
   System.out.println(x));
```

Figure 14: A stream which causes interference and will throw an error
Source: Ethan Williams

References

[1] *Collector. Java SE 8*. Oracle Help Center. Version 8. Oracle. 2017. URL: `https://docs.oracle.com/javase/8/docs/api/java/util/Collector.html`.

[2] Brian Goetz. *Java Streams*. Version 8. IBM. Feb. 2016. URL: `https://www.ibm.com/developerworks/library/j-java-streams-1-brian-goetz/index.html`.

[3] *Java 8 Streams. Stateful vs Stateless behavioral parameters*. LogicBig. Feb. 2017. URL: http://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stateful-vs-stateless/.

[4] Lucas Krecan. *Think Twice Before Using Java 8 Parallel Streams*. DZone. Feb. 2014. URL: https://dzone.com/articles/think-twice-using-java-8.

[5] Dan Newton. *Common Fork Join Pool and Streams*. DZone. Feb. 2017. URL: https://dzone.com/articles/common-fork-join-pool-and-streams.

[6] Tomasz Nurkiewicz. *Introduction to Writing Custom Collectors in Java 8*. Tomasz Nurkiewicz around Java and Concurrency. July 2014. URL: http://www.nurkiewicz.com/2014/07/introduction-to-writing-custom.html.

[7] *Parallelism. The Java^{TM} Tutorials*. Version 8. Oracle. 2017. URL: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.