Java 8 Parallel Streams

Ethan Williams

February 25, 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 instantiated with the `stream()` method only is referred to as a *serial stream*
- A stream instantiated with the `parallel().stream()` or `parallelStream()` methods is referred to as a *parallel stream*
- *Serializing data* refers to destroying the stream and bringing all collection items left in the stream into memory
- A stream is composed of 3 parts: a source which is the `Collection` it is operating on, intermediate operations such as `map()` which don't serialize data, and terminal operations like `toArray()` which serialize the stream

1 Introduction

Parallel streams were introduced into Java 8 alongside serial streams so that developers could utilize concurrency to make stream processing faster. Concurrency allows the program to utilize threads in order to more efficiently utilize modern multiprocessor design. In the case of parallel streams, each thread handles a different part of the stream and assemble all the parts together when the stream is serialized [6]

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 `Collection` must have an implementation of a `Spliterator` [2], an object which can effectively break up a collection to be processed separately.

Several operations are more essential to parallel streams, as is the case with `Collectors`. Calling the `collect()` method on the stream which will reduce and serialize it based on the behavior of the `Collector` object passed as an argument. With parallel streams, `Collector`s are the recommended strategy for reduction because other mechanisms that can be used in serial streams will result in inconsistent results when run in parallel. The reason for this is discussed in Section 4.3.

Despite parallel streams being introduced into Java to simplify concurrency, developers can easily corrupt data and cause system bugs. Although there are a plethora of bugs that can arise from improper use of Java concurrency practices, there are several issues specific to the parallel stream implementation which are covered in this paper. This includes submitting long-running or blocking operations to a stream, interference with the source of the stream, and using stateful expressions.

2 Spliterator

The `Spliterator` is the backbone of parallel streams, allowing the program to split a collection apart 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.

`Spliterator` *implementations differ based on the type of source collection, so this section focuses on a modified implementation for Java's* `ArrayList`.

2.1 Functionality

A `Spliterator` is an object which is represents the elements in a given range of the backing collection and represents the entire source collection at the beginning of a parallel stream. Calling the method `trySplit()`

will return a new `Spliterator` that represents some subset of the original `Spliterator`, while the original represents the rest of the collection. In the code example, the new `Spliterator` represents first half of untraversed elements while modifying the instance to represent the second half of untraversed elements. Figure 1 illustrates a call to `trySplit()` on a Spliterator representing the entire collection.

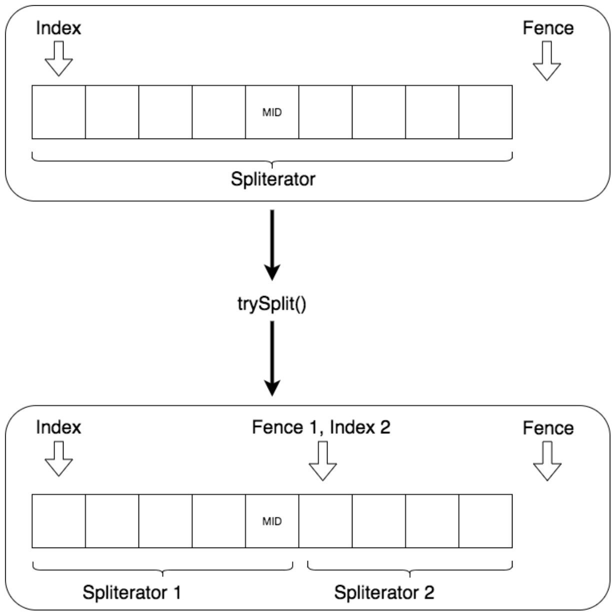

Figure 1: A Spliterator Before and After Splitting
Source: Ethan Williams

2.2 Implementation

In the example implementation of `Spliterator`, there are 3 instance fields: a source collection, an index, and a fence. The backing collection for each instance is the entire collection it is based on which ensures every instance has a consistent copy of the source. The index is the current cursor position of the `Spliterator`, which is analogous to the cursor position in an Iterator object. The fence is the index of the last element the Spliterator is responsible for plus one. The primary methods used are `tryEachRemaining()`, `tryNext()`, & `trySplit()`.

Although the cursor is similar to Iterator's, traversal through a Spliterator using `tryEachRemaining()` and `tryNext()` is different. `tryEachRemaining()` in Figure 2 takes a Consumer object which is the operation to be executed on each element of the collection. A typical implementation will simply iterate through all elements and call the `accept()` method on the consumer with the element as its only parameter. `tryNext()` in Figure 3 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.

```
public void forEachRemaining(Consumer<? super E>
   action) {
    int i;
    if ((i = index) >= 0 && (index = fence) <=
       a.length) {
        for (; i < hi; ++i)  action.accept((E)
           list.elementData[i]);
    }
    throw new ConcurrentModificationException();
}
```

Figure 2: Implementation of forEachRemaining()
Source: Java ArrayList, modified by Ethan Williams

```
public boolean tryAdvance(Consumer<? super E>
   action) {
    int hi = getFence(), i = index;
    if (i < hi) {
        index = i + 1;
        action.accept((E) list.elementData[i]);
        return true;
    }
    return false;
}
```

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

`Spliterator`'s primary functionality is encapsulated within the `trySplit()` method in Figure 4. 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 1.

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

Figure 4: 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 and is used in a stream with the `collect()` method. In other words, instances reduce a stream into a data structure which may be different than the source. To utilize `Collectors` in streams, developers can either use one of the many methods of the `Collectors` class such as `averagingInt()`, `groupingByConcurrent()`, and many more. Alternatively, a developer may choose to write their own `Collector` to reduce the stream differently.

3.1 Functionality

The example shown in Figure 5 groups employees by department into a `Map` object with the department as the key and a list of employees in that department as a value. `Collectors` can be used on both serial and parallel streams, though using the general reduction function `reduce()` is harder to use correctly with parallel streams since ordering is not guaranteed.

```
Map<Department, List<Employee>> byDept
    = employees.stream()
      .collect(Collectors.groupingBy(Employee::getDepartment));
```

Figure 5: Reduction of employees into map by a Collector
Source: Ethan Williams

3.2 Implementation

A `Collector` object has four methods which comprise the majority of its functionality: `supplier()`, `accumulator()`, `combiner()`, & `finisher()` [1]. The code example in Figure 11 illustrates a basic implementation which reduces a stream into an `ImmutableSet` using these 4 methods.

The `supplier()` method returns an instance of a mutable data structure that will hold the elements of the stream which is known as the *accumulator* [5], which in the code example is simply an `ImmutableSet` builder. Behind the scenes, the JVM will call this method several times as it assembles the resultant pieces of the full data structure from each of the substreams.

The `accumulator()` method returns a `BiConsumer` object which embodies an accumulator and an element which are passed as arguments to the method. When the instance is accepted then the operation will be invoked on the accumulator (this is often a simple add operation to the data structure) with the element. The JVM will utilize this to add new elements from substreams to an `ImmutableSet`.

The `combiner()` method details the logic on how two accumulators should be joined together, which is simply appending right to left in the code example. Funcionally, this is joining together two different `ImmutableSets`.

Finally, the `finisher()` method converts an accumulator into the resultant data type and returns it, finishing the reduction. In the code example, this is as easy as returning the built `ImmutableSet`.

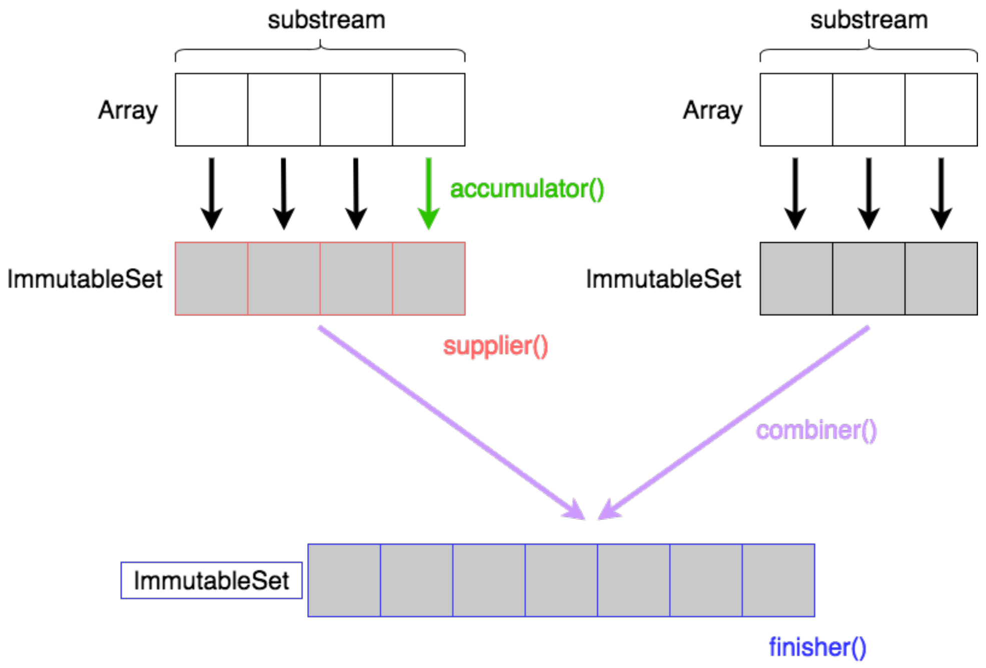

Figure 6: How a Collector is Used
Source: Ethan Williams

4 Practical Considerations when Using Parallel Streams

Using parallel streams may seem easy but there are several common pitfalls for developers. Many of these errors arise from programmers not considering normal facets of concurrent programming such as submitting blocking or long-running tasks during execution. Additionally, some of the implementation details of parallel streams prohibit interference with the stream source and stateful expressions inside intermediate operations.

4.1 Long-Running/Blocking Operations

Mistakes that can arise from not knowing Java's implementation can have arguably more severe consequences on application performance. One example relates to how the JVM breaks a parallel stream into threads. When a parallel stream is processes, the JVM begins by processing on the calling thread. As more subtasks are broken off, the JVM gets threads from `ForkJoinPool.common()`, which is a common thread pool used in the background of the whole application [3].

With the JVM's use of a common thread pool, one example of an error that could arise is if there is a long-running operation in a parallel stream as shown in Figure 7. 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. This can lead to severe degradation in performance. Luckily, there is a way to fix this (or at least make it better).

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

Figure 7: Reduction of employees into map by a Collector
Source: Ethan Williams

Although parallel streams can run into issues because it uses the common thread pool for the whole application, Java now allows a parallel stream to run with its own `Thread Pool`. An example of how to implement this on the previous code snippet is shown in Figure 8. This alleviates the issue because if one parallel stream is undermining performance, other parallel streams can still use threads even though the common pool doesn't have any more threads [4].

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

Figure 8: Reduction of employees into map by a Collector with custom Thread Pool
Source: Ethan Williams

4.2 Interference

Editing the source of the stream in an intermediate operation will throw a `ConcurrentModificationException` [6]. `Streams` do not contain any of the elements of the collection, instead they store references to the location of each element currently in the stream [2]. The stream in figure 9 attempts to add each element to the collection again. Since it attempts to modify the source before the stream has serialized, this operation will throw a `ConcurrentModificationException`.

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

Figure 9: 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 and should be watched carefully is using stateful expressions in an intermediate operation. The code in Figure 10 shows an example of a stateful operation while attempting to add elements to `parallelStorage` and print them. The issue is that although the `forEachOrdered` method is just fine and will print in the expected order. `parallelStorage`, however, 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 [6].

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

Figure 10: 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. *Streams under the hood. Java Streams, Part 3.* Understand java.util.stream internals. Version 8. IBM. Feb. 2016. URL: `https://www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz/index.html`.

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

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

[5] 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`.

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

A Code Example from Section 3.2

```java
import com.google.common.collect.ImmutableSet;

public class ImmutableSetCollector<T>
        implements Collector<T,
           ImmutableSet.Builder<T>,
           ImmutableSet<T>> {
    @Override
    public Supplier<ImmutableSet.Builder<T>>
       supplier() {
        return ImmutableSet::builder;
    }

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

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

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

    @Override
    public Set<Characteristics> characteristics() {
        return
           EnumSet.of(Characteristics.UNORDERED);
    }
}
```

Figure 11: Custom Collector
Source: [5]