Java 8 Parallel Streams

Ethan Williams

February 26, 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 in order to more efficiently utilize modern multiprocessor design [2]. 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 [7]

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 `Collector`s. Calling the `collect()` method on the stream will reduce and serialize it based on the behavior of the `Collector` object passed as an argument [1]. 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. 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

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

The `Spliterator` is the backbone of parallel streams, allowing the program to split a collection apart (illustrated in Figure 1) 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. In many implementations including the one used in this paper, each instance has 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.

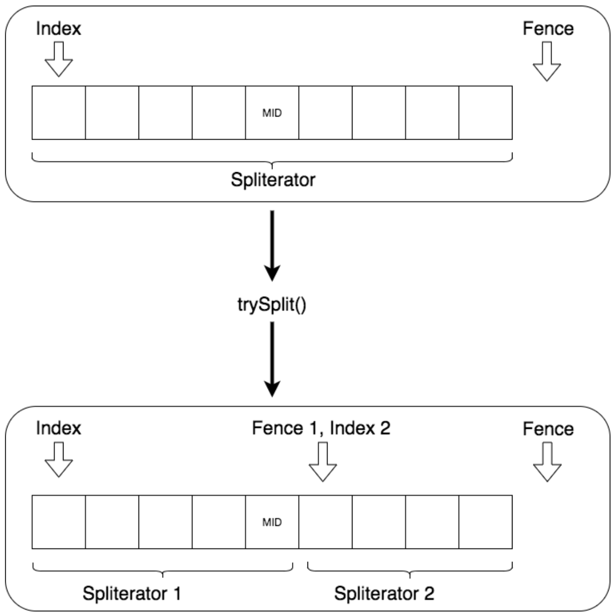

Figure 1: A Spliterator Before and After Splitting
Source: Ethan Williams

2.1 Implementation

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.

```
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. In the code example 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. `Collector`s can be used on both serial and parallel streams, though using the general reduction method `reduce()` is harder to use correctly with parallel streams since ordering is not guaranteed. To utilize `Collector`s in streams, developers can either use one of the many methods of the `Collectors` class such as `groupingByConcurrent()` or write their own `Collector` for custom behavior.

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

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

3.1 Implementation

A `Collector` object has four methods which comprise the majority of its functionality: `supplier()`, `accumulator()`, `combiner()`, & `finisher()` [1]. The functionality of each method is shown in Figure **??** and a code example in Figure 11, Appendix A shows a basic implementation which reduces a stream into an `ImmutableSet`.

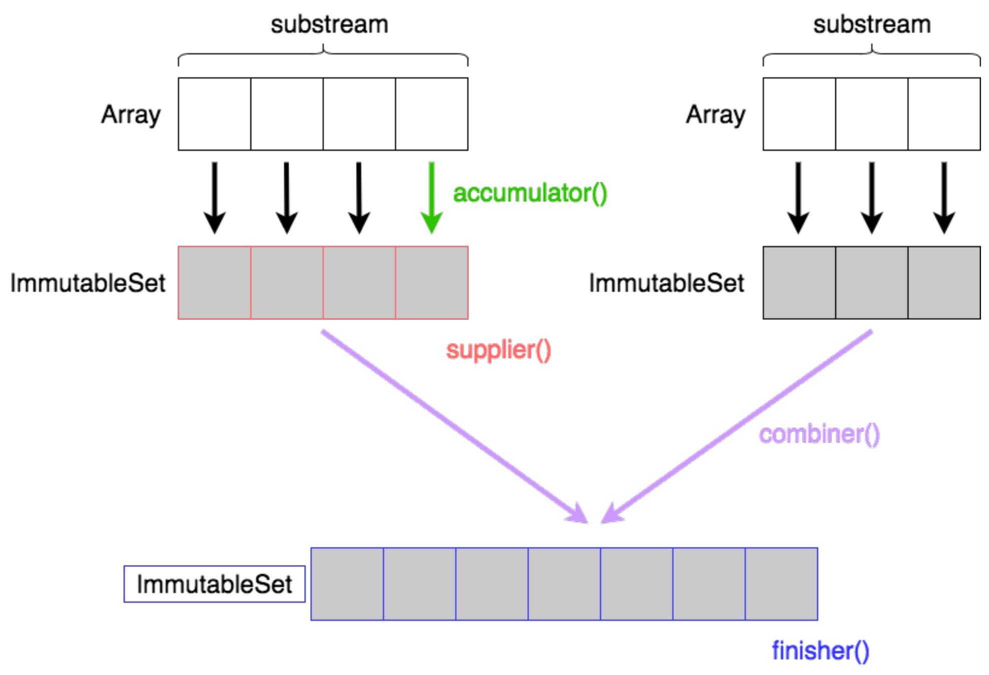

Figure 6: How a Collector is Used
Source: Ethan Williams

The `supplier()` method returns a mechanism to build an instance of a mutable data structure that will hold the elements of the stream which is known as the *accumulator* [6]. Behind the scenes, the JVM will call this method several times to create `ImmutableSets` for all of the substreams []. In the code example our `supplier()` is simply an `ImmutableSet` builder so the JVM can generate new instances at will.

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. The JVM uses this when of any element in the substream is done with processing to determine how or if it's added to the final data structure. In the code example, the `accumulator` simply appends the element to the end of the `ImmutableSet`.

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.

Finally, the `finisher()` method serializes the accumulator that is the result of combining all the substreams, finishing 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.

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 interference and stateful expressions which can lead to severe errors in the application.

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 7 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 7 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.

```java
// 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 7: Reduction of employees into map by a Collector
Source: Ethan Williams

Figure 8: Reduction of employees into map by a Collector with custom Thread Pool
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 9 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 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, which are operations that depends on the ordering of the elements [3]. The code in Figure 10 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 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. *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`.

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;
    }
}
```

Figure 11: Custom Collector
Source: [6]