Java 8 Parallel Streams

Ethan Williams

February 20, 2018

Contents

0 Prerequisites

0.1 Technical

This document was written for Java developers who have an interest in using concurrency in streams, and as such assumes knowledge of serial streams. 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 Language

Some of the language used in this paper may be ambiguous/confusing, the author's intended meanings for these terms are below:

- A stream instantiated with `.stream()` only is referred to as a serial stream
- A stream instantiated with `.parallel().stream()` or `.parallelStream()` is referred to as a parallel 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 the case of streams this is all handled by the JVM, in order to more efficiently utilize modern multiprocessor design.

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.

2.1 Basics

A Spliterator is an object which is represents the elements in a given range of the backing collection. At the beginning of a parallel stream there is one Spliterator that represents the entire collection. Calling the method `trySplit()` will return a new Spliterator that represents the 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.

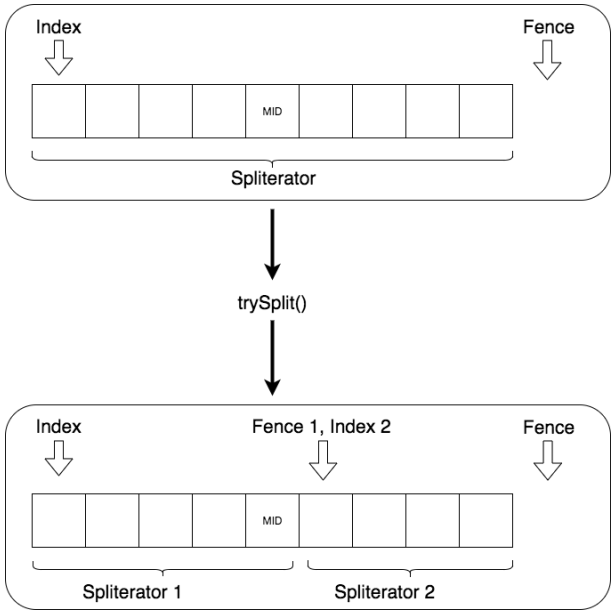

Figure 1: A Spliterator before and after splitting

2.2 Implementation

Spliterator implementations differ based on the collection so this section focuses on a simplified (easier to read) implementation of the Spliterator for Java's ArrayList.

Spliterator has 3 instance fields: a backing collection, an index, and a fence. The backing collection for every Spliterator the entire collection it is based on. 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 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()

```
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()

Spliterator's primary functionality is encapsulated within the `trySplit()` method in Figure 4. This 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. This is the code behind the split shown previously 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()

3 Collectors

4 Practical Considerations when Using Parallel Streams