When to Not to Use Java Parallel Streams

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 Lesson

- Know when to use parallel streams
- & when *not* to use parallel streams
When Not to Use Java Parallel Streams
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs

When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14)
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

List<CharSequence> arrayAllWords =
    TestDataFactory.getInput
    (sSHAKESPEARE_WORKS, "\s+"));

List<CharSequence> listAllWords =
    new LinkedList<>(arrayAllWords);

arrayAllWords.parallelStream()
    .count();

listAllWords.parallelStream()
    .count();
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
List<CharSequence> arrayAllWords =
    TestDataFactory.getInput(sSHAKESPEARE_WORKS, "\s+");

List<CharSequence> listAllWords =
    new LinkedList<>(arrayAllWords);

arrayAllWords.parallelStream()
    .count();

listAllWords.parallelStream()
    .count();
```

Make a LinkedList that contains all words in the works of Shakespeare
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

List<CharSequence> arrayAllWords = TestDataFactory.getInput(sSHAKESPEARE_WORKS, "\\s+");

List<CharSequence> listAllWords = new LinkedList<>(arrayAllWords);

arrayAllWords.parallelStream()
    .count();

listAllWords.parallelStream()
    .count();

The ArrayList parallel stream is much faster than the LinkedList parallel stream

LinkedList performs poorly since it doesn’t try to split evenly/efficiently
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
class ArrayListSpliterator {
    ...
    ArrayListSpliterator<E> trySplit() {
        int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
        return lo >= mid
            ? null
            : new ArrayListSpliterator<E>(list, lo, index = mid,
                                          expectedModCount);
    }
    ...
}
```

The ArrayList spliterator runs in $O(1)$ constant time

See openjdk/8u40-b25/java/util/ArrayList.java
When Not to Use Java Parallel Streams

• Parallel streams aren't suitable for certain types of programs, e.g.
• The source is expensive to split or splits unevenly

```java
class ArrayListSpliterator {
    //...
    ArrayListSpliterator<E> trySplit() {
        int hi = getFence(), lo = index,
            mid = (lo + hi) >>> 1;
        return lo >= mid
            ? null
            : new ArrayListSpliterator<E>(list, lo, index = mid,
                                         expectedModCount);
    }
    //...
}
```

*Compute the mid-point efficiently*
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
class ArrayListSpliterator {
    ...
    ArrayListSpliterator<E> trySplit() {
        int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
        return lo >= mid ? null :
            new ArrayListSpliterator<E>
            (list, lo, index = mid, expectedModCount);
    }
    ...
}
```

Split the array list evenly without copying the data
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
class LLSpliterator {
    ...
    public Spliterator<E> trySplit() {
        ...
        int n = batch + BATCH_UNIT;
        ...
        Object[] a = new Object[n];
        int j = 0;
        do { a[j++] = p.item; } 
        while ((p = p.next) != null 
             && j < n);
        ...
        return Spliterators.spliterator(a, 0, j, 
                          Spliterator.ORDERED);
    }
}
```

The LinkedList spliterator runs in $O(n)$ linear time

See openjdk/8-b132/java/util/LinkedList.java
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
class LLSpliterator {
    ...
    public Spliterator<E> trySplit()
    {
        ...
        int n = batch + BATCH_UNIT;
        ...
        Object[] a = new Object[n];
        int j = 0;
        do { a[j++] = p.item; }
        while ((p = p.next) != null && j < n);
        ...
        return Spliterators.spliterator(a, 0, j,
                                           Spliterator.ORDERED);
    }
}
```

Create a fixed-size chunk
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
- The source is expensive to split or splits unevenly

```java
class LLSpliterator {
    
    public Spliterator<E> trySplit()
    {
        
        int n = batch + BATCH_UNIT;
        
        Object[] a = new Object[n];
        int j = 0;
        do {
            a[j++] = p.item;
        } while ((p = p.next) != null && j < n);
        
        return Spliterators.spliterator(a, 0, j,
                                           Spliterator.ORDERED);
    }

    
}
```

Copy data into the chunk
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly

```java
class LLSpliterator {
  ...
  public Spliterator<E> trySplit(){
    ...
    int n = batch + BATCH_UNIT;
    ...
    Object[] a = new Object[n];
    int j = 0;
    do { a[j++] = p.item; } while ((p = p.next) != null && j < n);
    ...
    return Spliterators.spliterator(a, 0, j,
                                      Spliterator.ORDERED);
```

Create a new spliterator that covers the chunk
When Not to Use Java Parallel Streams

• Parallel streams aren’t suitable for certain types of programs, e.g.
  • The source is expensive to split or splits unevenly
  • The startup costs of parallelism overwhelm the amount of data

---

TO-DO LIST:

1. EVERYTHING

---

```java
class ParallelStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream
            .rangeClosed(1, n)
            .parallel() ...
            .reduce(BigInteger.ONE,
                     BigInteger::multiply);
    }
}
```

```java
class SequentialStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream
            .rangeClosed(1, n) ...
            .reduce(BigInteger.ONE,
                     BigInteger::multiply);
    }
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data

```java
class ParallelStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream.rangeClosed(1, n)
            .parallel() ...
            .reduce(BigInteger.ONE, BigInteger::multiply);
    }
}
```

```java
class SequentialStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream.rangeClosed(1, n) ...
            .reduce(BigInteger.ONE, BigInteger::multiply);
    }
}
```

When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data

If n is small then this parallel solution will be inefficient
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data

If $n$ is small then this sequential solution will be more efficient

```java
class ParallelStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream
            .rangeClosed(1, n)
            .parallel()
            .reduce(BigInteger.ONE, 
            BigInteger::multiply);
    }
}

class SequentialStreamFactorial {
    BigInteger factorial(long n) {
        return LongStream
            .rangeClosed(1, n) ...
            .reduce(BigInteger.ONE, 
            BigInteger::multiply);
    }
}
```
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly

```java
List<CharSequence> allWords =
    new LinkedList<>
    (TestDataFactory.getInput
    (sSHAKESPEARE_DATA_FILE,
        "\s+"));
...
Set<CharSequence> uniqueWords =
    allWords
    .parallelStream()
    ...
    .collect(toCollection
        (TreeSet::new));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly

```java
List<CharSequence> allWords = new LinkedList<>((TestDataFactory.getInput(sSHAKESPEARE_DATA_FILE, "\\s+"));

Set<CharSequence> uniqueWords = allWords
    .parallelStream()
    .collect(toCollection(TreeSet::new));
```

A linked list of all words in the complete works of Shakespeare
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly

```java
List<CharSequence> allWords = new LinkedList<>((TestDataFactory.getInput(sSHAKESPEARE_DATA_FILE, "\s+"));

Set<CharSequence> uniqueWords = allWords
  .parallelStream()
  .collect(toCollection(TreeSet::new));
```

Performance will be poor due to the overhead of combining partial results for a set in a parallel stream
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly

```java
List<CharSequence> allWords = new LinkedList<>((TestDataReaderFactory.getInput(sSHAKESPEARE_DATA_FILE, "\s+"));

Set<CharSequence> uniqueWords = allWords.parallelStream()
  .collect(toCollection(TreeSet::new));
```

Combining costs can be alleviated if the amount of work performed per element is large (i.e., the “NQ model”)

When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly

```java
List<CharSequence> allWords = new LinkedList<>((TestDataFactory.getInput(sSHAKESPEARE_DATA_FILE, "\s+"));
...
Set<CharSequence> uniqueWords = allWords.parallelStream()
  .collect(toSet());
```

A concurrent collector can also be used to optimize the reduction phase

See Java8/ex14/src/main/java/utils/ConcurrentHashSetCollector.java
When Not to Use Java Parallel Streams

- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don't sufficiently exploit parallelism

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex15](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex15)
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism

List<Double> result = Stream
  .iterate(2, i -> i + 1)
  .parallel()
  .filter(this::isEven)
  .limit(n)
  .map(this::findSQRT)
  .collect(toList());

List<Double> result = LongStream
  .range(2, (n * 2) + 1)
  .parallel()
  .filter(this::isEven)
  .mapToObj(this::findSQRT)
  .collect(toList());

Create a list containing sqrt of the first ‘n’ even numbers
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism

```
List<Double> result = Stream
    .iterate(2, i -> i + 1)
    .parallel()
    .filter(this::isEven)
    .limit(n)
    .map(this::findSQRT)
    .collect(toList());

List<Double> result = LongStream
    .range(2, (n * 2) + 1)
    .parallel()
    .filter(this::isEven)
    .mapToObj(this::findSQRT)
    .collect(toList());
```

Stream.iterate() & limit() split & parallelize poorly since iterate creates an ordered stream...

See [www.java2novice.com/java-8/streams/limit-method-example](http://www.java2novice.com/java-8/streams/limit-method-example)
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism

Create a list containing sqrt of the first ‘n’ even numbers

```
List<Double> result = Stream
    .iterate(2, i -> i + 1)
    .parallel()
    .filter(this::isEven)
    .limit(n)
    .map(this::findSQRT)
    .collect(toList());
```

```
List<Double> result = LongStream
    .range(2, (n * 2) + 1)
    .parallel()
    .filter(this::isEven)
    .mapToObj(this::findSQRT)
    .collect(toList());
```
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism

```java
List<Double> result = Stream
    .iterate(2, i -> i + 1)
    .parallel()
    .filter(this::isEven)
    .limit(n)
    .map(this::findSQRT)
    .collect(toList());
```

```java
List<Double> result = LongStream
    .range(2, (n * 2) + 1)
    .parallel()
    .filter(this::isEven)
    .mapToObj(this::findSQRT)
    .collect(toList());
```

`LongStream.range()` splits nicely & thus runs efficiently in parallel
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism
  - There aren’t many/any cores

*Older computing devices just have a single core, which limits available parallelism*
When Not to Use Java Parallel Streams

• Parallel streams aren’t suitable for certain types of programs, e.g.
  • The source is expensive to split or splits unevenly
  • The startup costs of parallelism overwhelm the amount of data
  • Combining partial results is costly
  • Some streams operations don’t sufficiently exploit parallelism
  • There aren’t many/any cores
  • No built-in means to shutdown processing of a parallel stream
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism
  - There aren’t many/any cores
  - No built-in means to shutdown processing of a parallel stream

```java
private static volatile boolean mCancelled;

Image downloadImage(Cache.Item item) {
    if (mCancelled)
        throw new CancellationException(
            "Canceling crawl.");
    ...
```
When Not to Use Java Parallel Streams

- Parallel streams aren’t suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly
  - The startup costs of parallelism overwhelm the amount of data
  - Combining partial results is costly
  - Some streams operations don’t sufficiently exploit parallelism
  - There aren’t many/any cores
  - No built-in means to shutdown processing of a parallel stream

```java
private static volatile boolean mCancelled;

Image downloadImage(Cache.Item item) {
    if (mCancelled) {
        throw new CancellationException("Canceling crawl.");
    }
    ...
}
```

Before downloading the next image, check for cancellation & throw an exception if cancelled
End of When Not to Use Java Parallel Streams