# 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

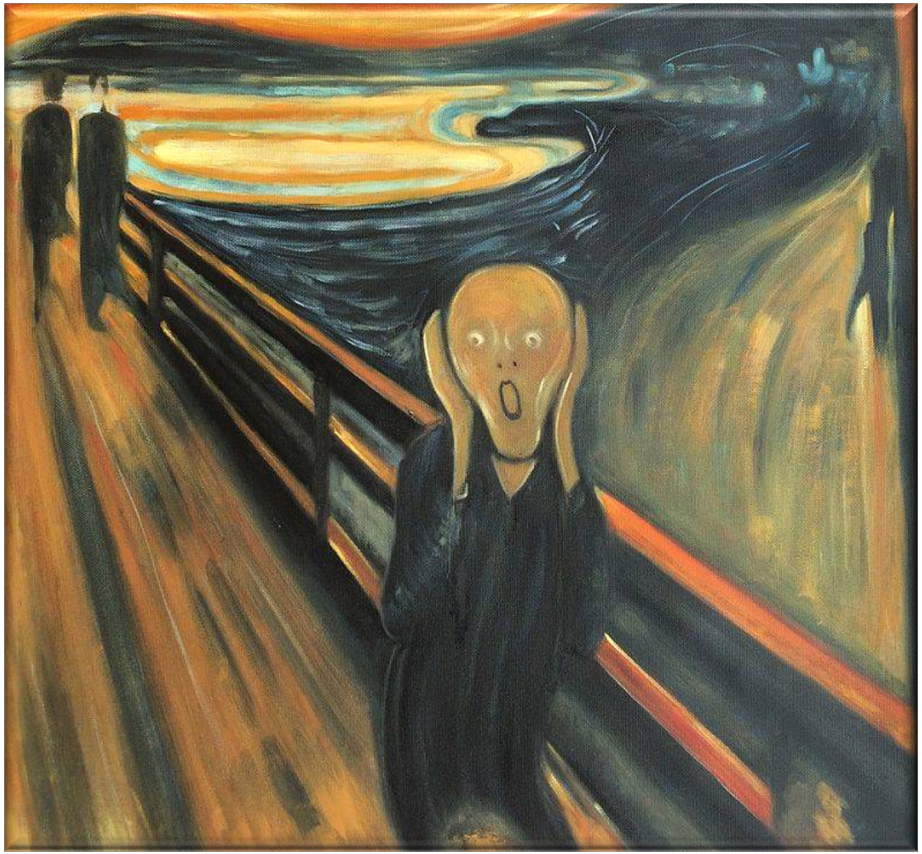
# Learning Objectives in this Part of the Lesson

- Learn when to use parallel streams
  - & when *not* to use parallel streams
    - e.g., the source is expensive to split or splits unevenly, startup costs of parallelism are too high, combining partial results is costly, as well as when there aren't many cores

# When Not to Use Java Parallel Streams

# When Not to Use Java Parallel Streams

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

See developer.ibm.com/articles/j-java-streams-5-brian-goetz

# 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> arrayWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS,
     sWHITESPACE_AND_PUNCTUATION");

List<CharSequence> listWords =
  new LinkedList<>(arrayWords);

arrayWords.parallelStream()
                  ...;

listWords.parallelStream()
                  ...;
```

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

*Make an ArrayList that contains all words in the works of Shakespeare*

```
List<CharSequence> arrayWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS,
     sWHITESPACE_AND_PUNCTUATION");

List<CharSequence> listWords =
  new LinkedList<>(arrayWords);

arrayWords.parallelStream()
                  ...;

listWords.parallelStream()
                  ...;
```

# 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> arrayWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS,
     sWHITESPACE_AND_PUNCTUATION");

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

arrayWords.parallelStream()
                  ...;

listWords.parallelStream()
                  ...;
```

*Make a LinkedList that contains all words in the works of Shakespeare*

LinkedList doesn't split evenly or efficiently compared with ArrayList

# 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

```
Starting spliterator tests for 100000
words....printing results
   599 msecs: ArrayList parallel
   701 msecs: LinkedList parallel

Starting spliterator tests for 883311
words....printing results
  5718 msecs: ArrayList parallel
 31226 msecs: LinkedList parallel
```

```
List<CharSequence> arrayWords =
  TestDataFactory.getInput
    (sSHAKESPEARE_WORKS,
     sWHITESPACE_AND_PUNCTUATION");

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

arrayWords.parallelStream()
                  ...;

listWords.parallelStream()
                  ...;
```

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

See earlier lesson on "*Java Parallel Stream Internals: Demo'ing Spliterator Performance*"

# 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 ArrayList spliterator runs in O(1) constant time*

```
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);
  }
  ...
```

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

```
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

```
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

```
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

*Create a fixed-size chunk*

```
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);
```

# 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

*Copy data into the chunk*

```
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);
```

# When Not to Use Java Parallel Streams

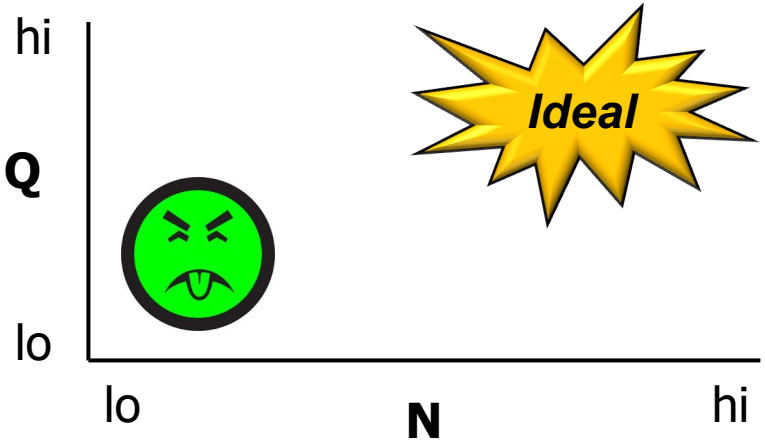
- Parallel streams aren't suitable for certain types of programs, e.g.
  - The source is expensive to split or splits unevenly

```
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




```
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);
  ...
```

See previous lesson on "*When to Use Parallel Streams*"

# 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

*The overhead of creating a parallel stream is > than the benefits of parallelism for small values of 'n'*

```
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);
  ...
```

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

*If n is small then this parallel solution will be inefficient*

```
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

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

```
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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
         + testType,
         true,
         arrayWords,
         toCollection
           (setSupplier));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex36

# 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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
         + testType,
         true,
         arrayWords,
         toCollection
           (setSupplier));
```

*An array 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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
           + testType,
         true,
         arrayWords,
         toCollection
           (setSupplier));
```

*Performance may be poor due to the overhead of combining partial results for a set in a parallel stream*

In this case setSupplier is TreeSet::new

# 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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
         + testType,
         true,
         arrayWords,
         toCollection
           (setSupplier));
```

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

See developer.ibm.com/articles/j-java-streams-5-brian-goetz

# 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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
         + testType,
         true,
         arrayWords,
         toCollection
           (setSupplier));
```

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#newKeySet

# 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

```
List<CharSequence> arrayWords =
  new ArrayList<>
    (TestDataFactory.getInput
       (sSHAKESPEARE_DATA_FILE,
        sSPLIT_WORDS));
...

collect
  .apply("non-concurrent "
         + testType,
         true,
         arrayWords,
         toCollection
```

```
Starting collector tests for 100000 words..printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words..printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

*Concurrent collector may scale much better than non-concurrent collector*

See previous earlier lesson on "*Java Parallel Stream Internals: Demo'ing Collector Performance*"

# 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());
```

See 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

*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

```
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

# 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

*LongStream.range() splits nicely & thus runs efficiently in parallel*

```
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
  - There aren't many cores

*Older computing devices just have a single core, which limits available parallelism*

# When Not to Use Java Parallel Streams

- Also be aware that there is no built-in means to shutdown processing of a parallel stream

See video.disney.com/watch/sorcerer-s-apprentice-fantasia-4ea9ebc01a74ea59a5867853

# When Not to Use Java Parallel Streams

- Also be aware that there is no built-in means to shutdown processing of a parallel stream

```
private static volatile
  boolean mCancelled;
```

*Define a static volatile flag*

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

# When Not to Use Java Parallel Streams

- Also be aware that there is no built-in means to shutdown processing of a parallel stream

```
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