Evaluate the Limitations of Java Parallel Streams

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

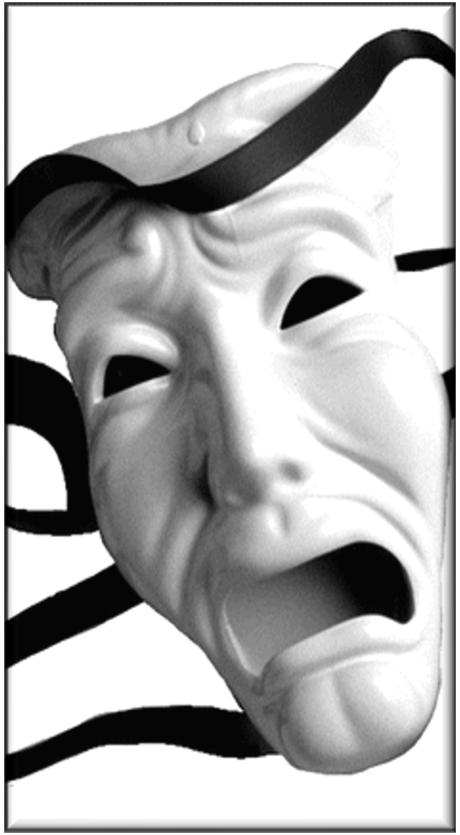
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Evaluate the benefits of Java parallel streams
- Evaluate the limitations of Java parallel streams



Limitations of Java Parallel Streams

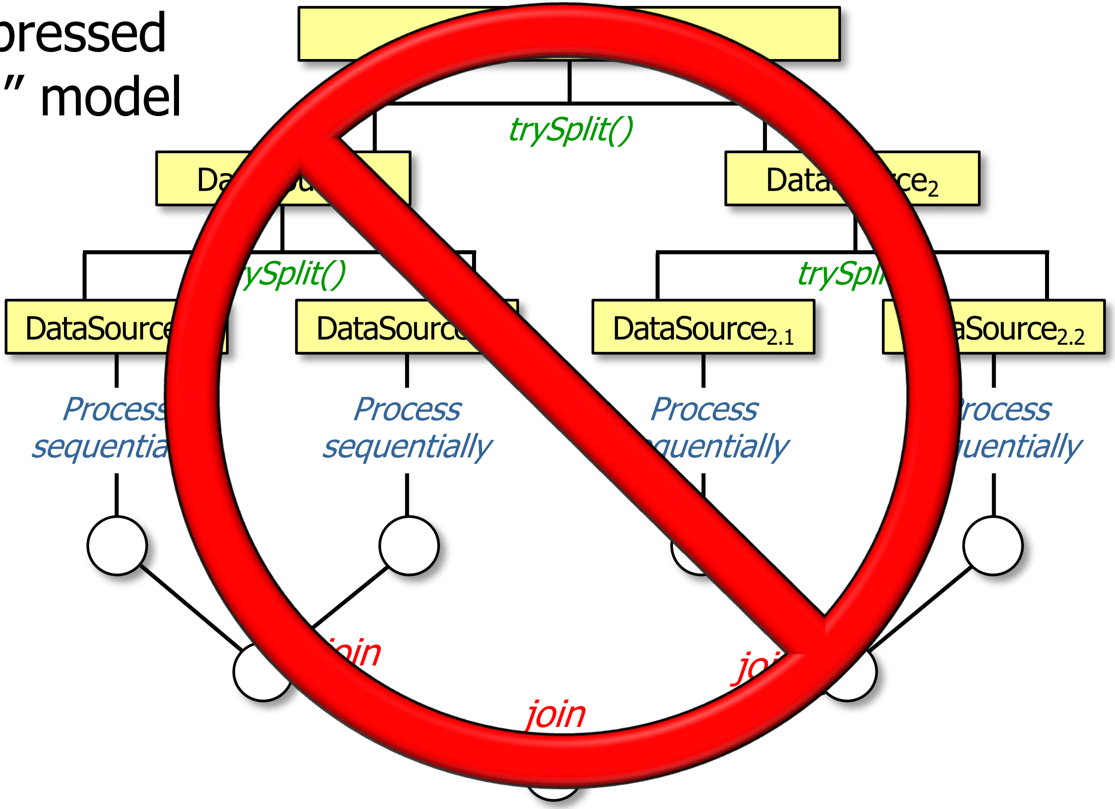
Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams

The Java parallel streams framework is not all unicorns & rainbows!!

Limitations of Java Parallel Streams

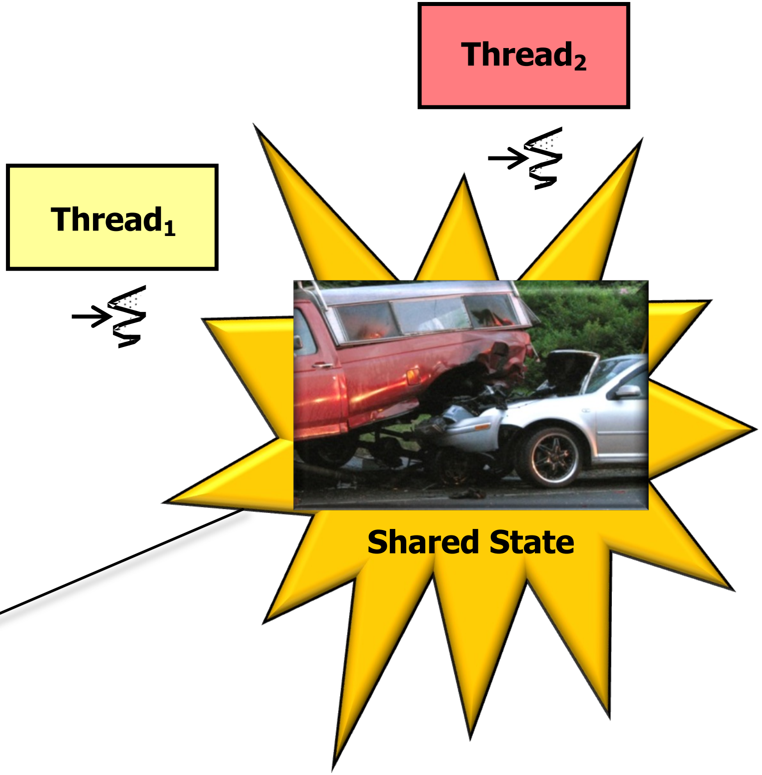
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model



See dzone.com/articles/whats-wrong-java-8-part-iii

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe



Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition#Software

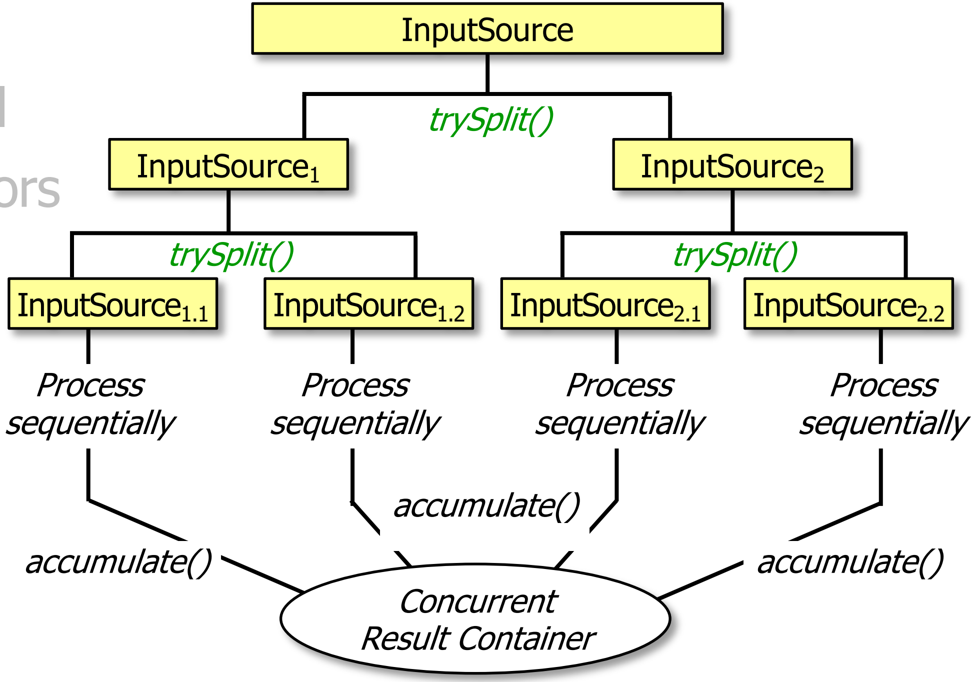
Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…

See lesson on "*Java SearchWithParallelSpliterator Example: trySplit()*"

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
    - Concurrent collectors are easier



See lesson on "*Java Parallel Stream Internals: Non-Concurrent & Concurrent Collectors*"

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool

See dzone.com/articles/think-twice-using-java-8

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
    - Java completable futures don't have this limitation

See dzone.com/articles/think-twice-using-java-8

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
    - Java completable futures don't have this limitation
    - It's important to know how to apply ManagedBlockers

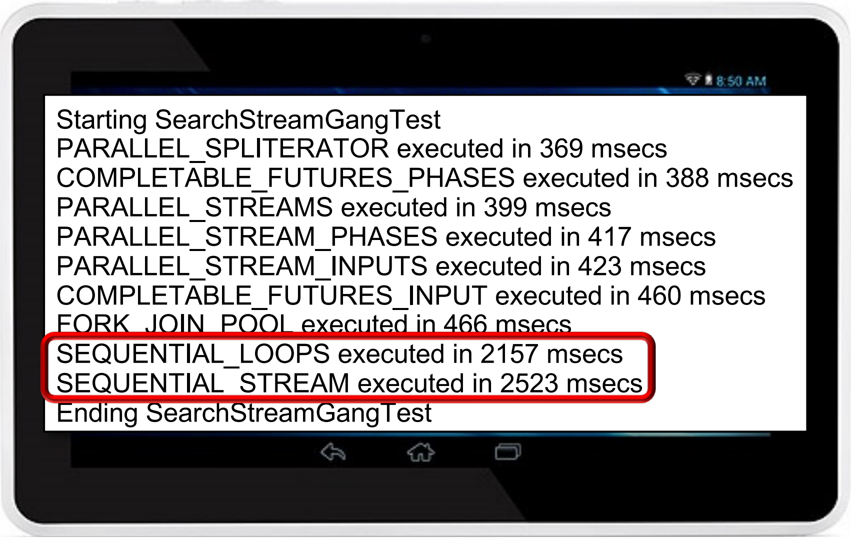
See "*The Java Fork-Join Pool: Applying the ManagedBlocker Interface*"

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead

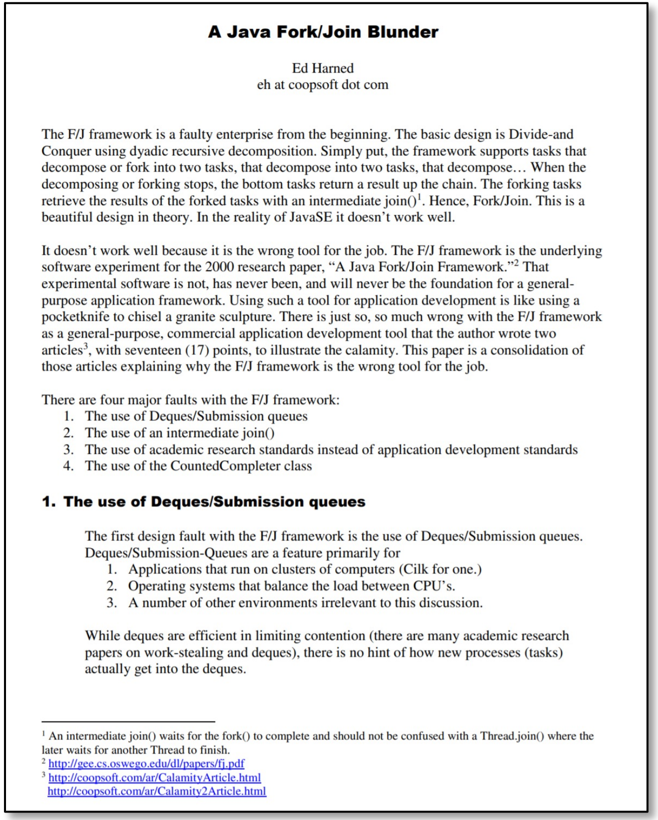


```
Starting SearchStreamGangTest
PARALLEL_SPLITERATOR executed in 369 msecs
COMPLETABLE_FUTURES_PHASES executed in 388 msecs
PARALLEL_STREAMS executed in 399 msecs
PARALLEL_STREAM_PHASES executed in 417 msecs
PARALLEL_STREAM_INPUTS executed in 423 msecs
COMPLETABLE_FUTURES_INPUT executed in 460 msecs
FORK_JOIN_POOL executed in 466 msecs
SEQUENTIAL_LOOPS executed in 2157 msecs
SEQUENTIAL_STREAM executed in 2523 msecs
Ending SearchStreamGangTest
```

See blog.jooq.org/2015/12/08/3-reasons-why-you-shouldnt-replace-your-for-loops-by-stream-foreach

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join "blunder"



A Java Fork/Join Blunder

Ed Harned
eh at coopsoft dot com

The F/J framework is a faulty enterprise from the beginning. The basic design is Divide-and Conquer using dyadic recursive decomposition. Simply put, the framework supports tasks that decompose or fork into two tasks, that decompose into two tasks, that decompose… When the decomposing or forking stops, the bottom tasks return a result up the chain. The forking tasks retrieve the results of the forked tasks with an intermediate join()[1]. Hence, Fork/Join. This is a beautiful design in theory. In the reality of JavaSE it doesn't work well.

It doesn't work well because it is the wrong tool for the job. The F/J framework is the underlying software experiment for the 2000 research paper, "A Java Fork/Join Framework."[2] That experimental software is not, has never been, and will never be the foundation for a general-purpose application framework. Using such a tool for application development is like using a pocketknife to chisel a granite sculpture. There is just so, so much wrong with the F/J framework as a general-purpose, commercial application development tool that the author wrote two articles[3], with seventeen (17) points, to illustrate the calamity. This paper is a consolidation of those articles explaining why the F/J framework is the wrong tool for the job.

There are four major faults with the F/J framework:

1. The use of Deques/Submission queues
2. The use of an intermediate join()
3. The use of academic research standards instead of application development standards
4. The use of the CountedCompleter class

1. The use of Deques/Submission queues

The first design fault with the F/J framework is the use of Deques/Submission queues. Deques/Submission-Queues are a feature primarily for

1. Applications that run on clusters of computers (Cilk for one.)
2. Operating systems that balance the load between CPU's.
3. A number of other environments irrelevant to this discussion.

While deques are efficient in limiting contention (there are many academic research papers on work-stealing and deques), there is no hint of how new processes (tasks) actually get into the deques.

[1] An intermediate join() waits for the fork() to complete and should not be confused with a Thread.join() where the later waits for another Thread to finish.

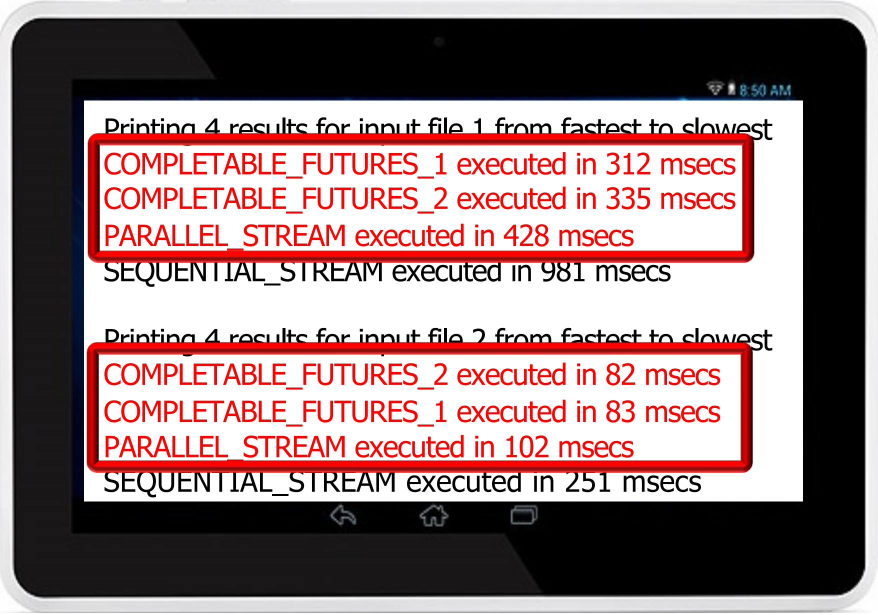
[2] http://gee.cs.oswego.edu/dl/papers/fj.pdf

[3] http://coopsoft.com/ar/CalamityArticle.html
http://coopsoft.com/ar/Calamity2Article.html

See coopsoft.com/dl/Blunder.pdf

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join "blunder"
    - Java completable futures may be more efficient & scalable



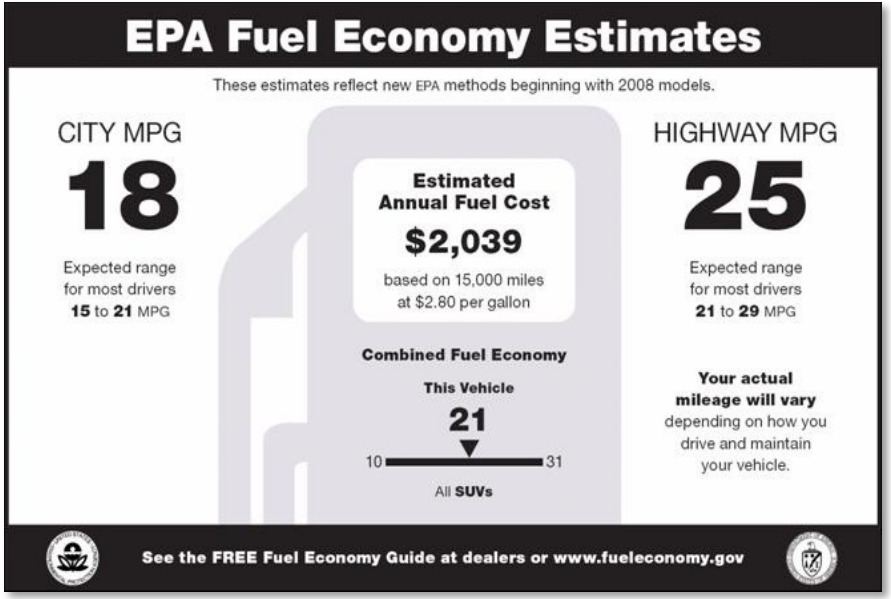
```
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Limitations of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join "blunder"
    - Java completable futures may be more efficient & scalable



EPA Fuel Economy Estimates

These estimates reflect new EPA methods beginning with 2008 models.

CITY MPG
18
Expected range for most drivers
15 to **21** MPG

Estimated Annual Fuel Cost
$2,039
based on 15,000 miles at $2.80 per gallon

Combined Fuel Economy
This Vehicle
21
10 ▼ 31
All **SUVs**

HIGHWAY MPG
25
Expected range for most drivers
21 to **29** MPG

Your actual mileage will vary depending on how you drive and maintain your vehicle.

See the FREE Fuel Economy Guide at dealers or www.fueleconomy.gov

Naturally, your mileage may vary..

Limitations of Java Parallel Streams

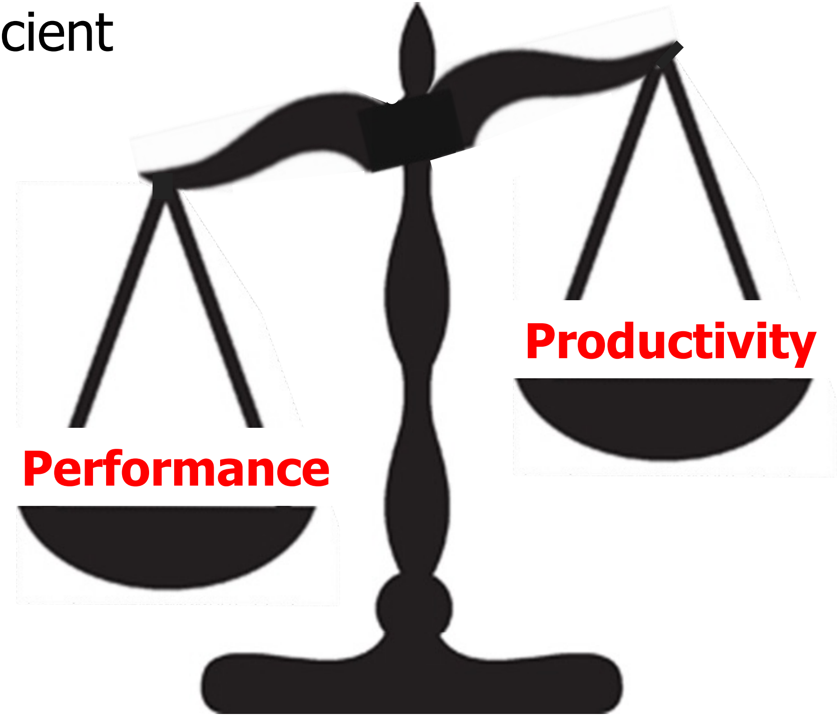
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - Race conditions may occur if behaviors aren't stateless & thread-safe
  - Parallel spliterators may be tricky…
  - All parallel streams can only share the common fork-join pool
  - Streams incur some overhead
  - There's no substitute for benchmarking!

See java-performance.info/jmh

Wrapping Up Java Parallel Streams

Wrapping Up Java Parallel Streams

- In general, there's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks
  - i.e., completable futures are more efficient & scalable, but are harder to program



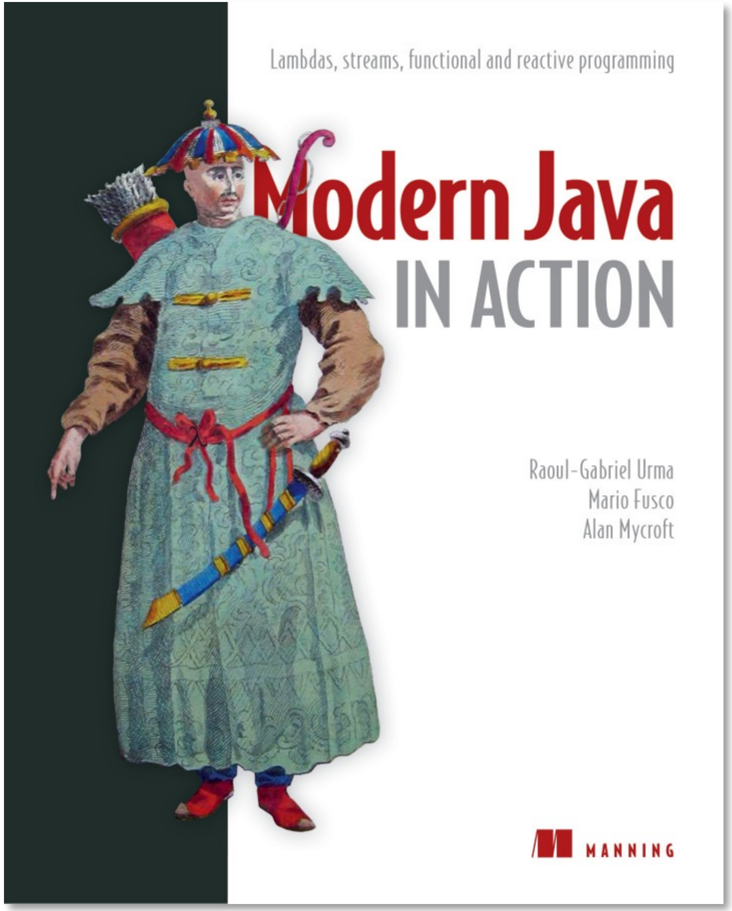
Wrapping Up Java Parallel Streams

- In general, however, the pros of Java parallel streams far outweigh the cons for many use cases!!

See www.ibm.com/developerworks/library/j-jvmc2

Wrapping Up Java Parallel Streams

- Good coverage of parallel streams appears in the book "Modern Java in Action"



See www.manning.com/books/modern-java-in-action

End of Evaluate the Limitations of Java Parallel Streams