How Java Parallel Streams Work "Under the Hood"

Douglas C. Schmidt

<u>d.schmidt@vanderbilt.edu</u>

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software Integrated Systems

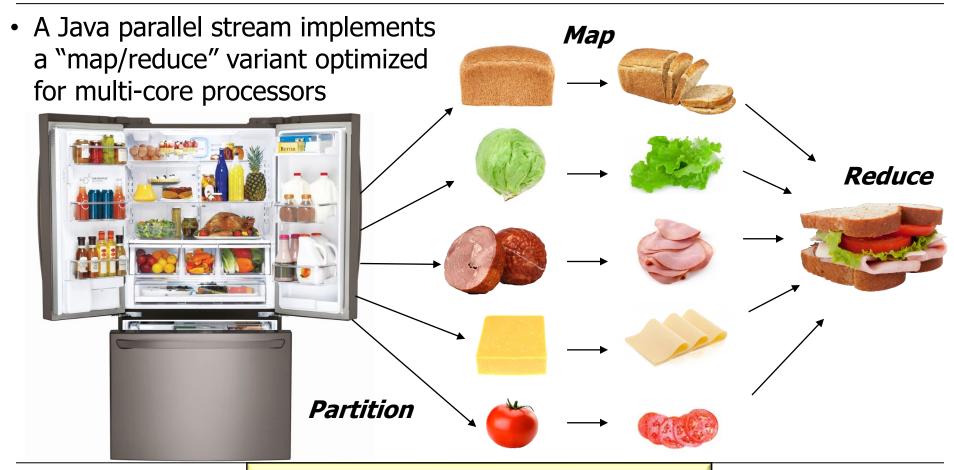
Vanderbilt University Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

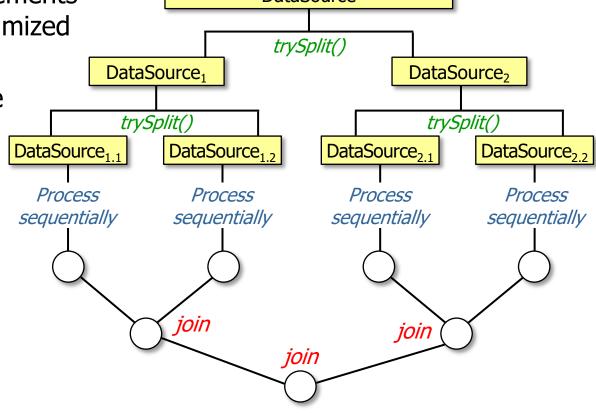
 Know how aggregate operations & functional programming features are applied seamlessly in parallel streams **DataSource** Learn how parallel stream phases trySplit() work "under the hood" DataSource₁ DataSource₂ trySplit() trySplit() DataSource_{1 1} DataSource_{1,2} DataSource_{2 1} DataSource_{2,2} **Process Process Process Process** sequentially sequentially sequentially sequentially ioin join ioin

See developer.ibm.com/articles/j-java-streams-3-brian-goetz



See en.wikipedia.org/wiki/MapReduce

- A Java parallel stream implements **DataSource**
 - a "map/reduce" variant optimized for multi-core processors
 - It's actually a three phase "split-apply-combine" data processing strategy



The split-apply-combine phases are:

 CollectionData

 Split - Recursively partition a

1. Split – Recursively partition a data source into "chunks"

CollectionData

trySplit()

trySplit()

trySplit()

Collection Data₁



CollectionData₂

Collection Data_{2,2}

CollectionData_{1,2}

See en.wikipedia.org/wiki/Divide_and_conquer_algorithm

The colit-apply-combine phaces are:

CollectionData

 The split-apply-combine phases are: CollectionData **1. Split** – Recursively partition a trySplit() data source into "chunks" CollectionData₁ CollectionData₂ trySplit() trySplit() CollectionData_{1.2} Collection Data_{1.1} CollectionData_{2.1} CollectionData_{2.2} Each chunk is an independent & "atomic" subset of the data source

CollectionData₁

trySplit()

- The split-apply-combine phases are: CollectionData
- 1. Split Recursively partition a

data source into "chunks"

Spliterators partition collections in Java

```
CollectionData<sub>1.1</sub>
CollectionData<sub>1.2</sub>

CollectionData<sub>2.1</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

Spliterator<T> trySplit();
   long estimateSize();
   int characteristics();
}
```

trySplit()

Collection Data₂

trySplit()

- The split-apply-combine phases are:

 CollectionData
- The split-apply-combine phases are:
 1. Split Recursively partition a

data source into "chunks"

Spliterators partition collections in Java

```
CollectionData<sub>1.1</sub>

CollectionData<sub>1.2</sub>

CollectionData<sub>2.1</sub>

CollectionData<sub>2.1</sub>

CollectionData<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

CollectionData<sub>2.2</sub>

Discreption Data<sub>2.1</sub>

CollectionData<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

CollectionData<sub>2.2</sub>

Interpretion Data<sub>2.2</sub>

public interface Spliterator<T> trySplit();

Interpretion Data<sub>2.2</sub>

collectionData<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

Interpretion Data<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

Interpretion Data<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

CollectionData<sub>2.2</sub>

public interface Spliterator<T> {
   boolean tryAdvance(Consumer<? Super T> action);

CollectionData<sub>2.2</sub>
```

CollectionData₁

trySplit()

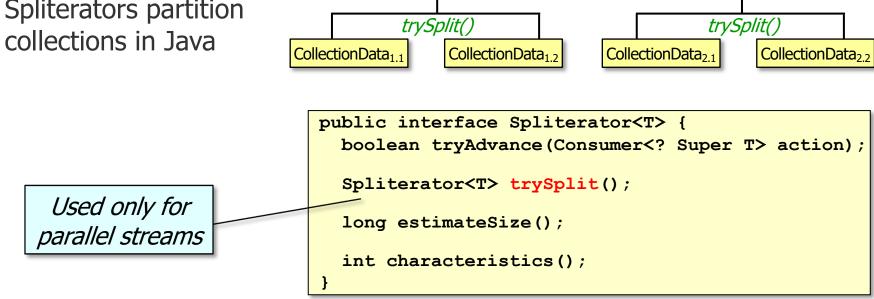
trySplit()

Collection Data₂

trySplit()

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance

- CollectionData
- The split-apply-combine phases are: **1. Split** – Recursively partition a
 - data source into "chunks" Spliterators partition
 - collections in Java



CollectionData₁

trySplit()

Collection Data₂

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit

InputString₁

- The split-apply-combine phases are: InputString
 - 1. Split Recursively partition a

data source into "chunks"

- Spliterators partition collections in Java
- Each Java collection

has a spliterator

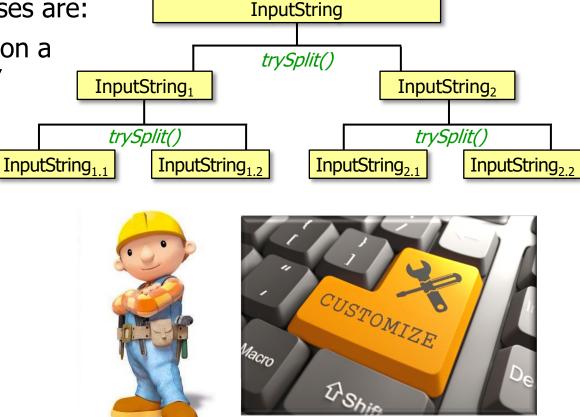
interface Collection<E> {
 ...
 default Spliterators.spliterator() {
 return Spliterators.spliterator(this, 0);
 }
 default Stream<E> parallelStream() {

return StreamSupport.stream(spliterator(), true);

trySplit()

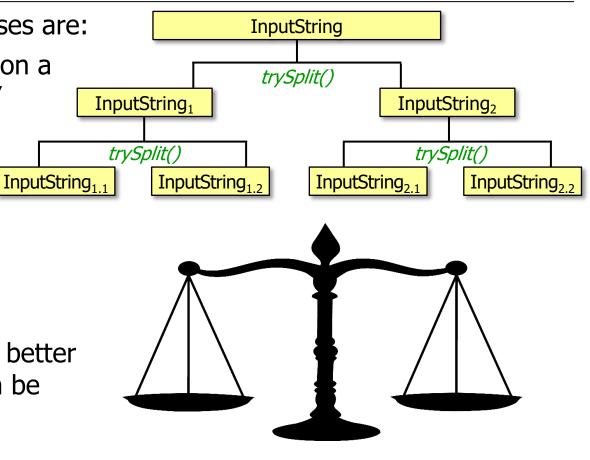
InputString₂

- The split-apply-combine phases are:
 - 1. Split Recursively partition a data source into "chunks"
 - Spliterators partition collections in Java
 - Each Java collection has a spliterator
 - Programmers can define custom spliterators



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

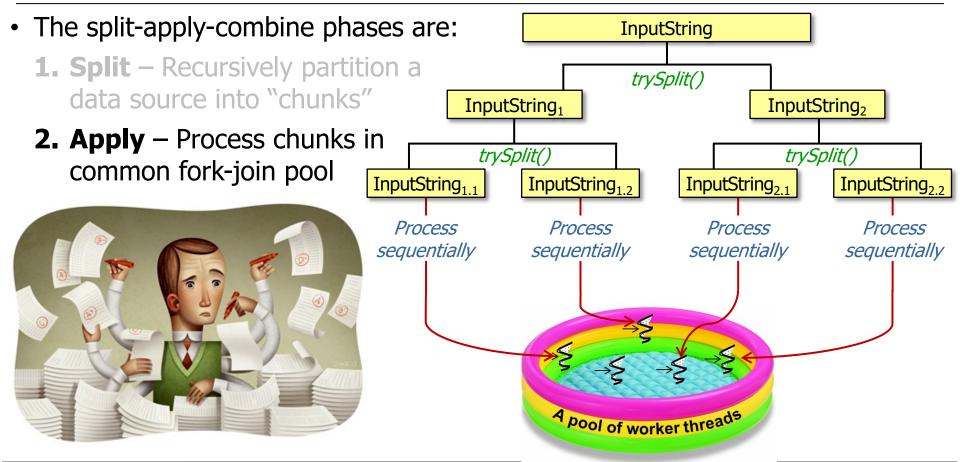
- The split-apply-combine phases are:
 - 1. Split Recursively partition a data source into "chunks"
 - Spliterators partition collections in Java
 - Each Java collection has a spliterator
 - Programmers can define custom spliterators
 - Parallel streams perform better on data sources that can be split efficiently & evenly



See www.airpair.com/java/posts/parallel-processing-of-io-based-data-with-java-streams

 The split-apply-combine phases are: InputString 1. Split – Recursively partition a data source into "chunks" InputString₁ InputString₂ 2. Apply – Process chunks in common fork-join pool InputString_{1 1} InputString_{1,2} InputString_{2,1} InputString_{2,2} **Process Process Process Process** sequentially sequentially sequentially sequentially

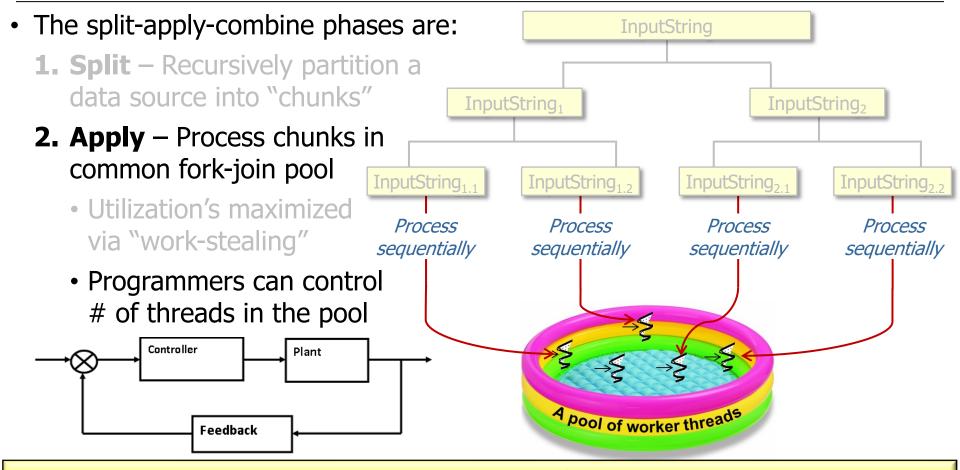
A pool of worker threads



Splitting & applying run simultaneously (after certain limits met), not sequentially

• The split-apply-combine phases are: InputString **1. Split** – Recursively partition a data source into "chunks" InputString₁ InputString₂ 2. Apply – Process chunks in common fork-join pool InputString_{1,1} InputString_{2,2} InputString_{1,2} InputString_{2,1} Utilization's maximized **Process Process Process Process** via "work-stealing" sequentially sequentially sequentially sequentially A pool of worker threads

See lesson on "Java Parallel Stream Internals: Mapping onto the Common ForkJoinPool"



See lesson on "Java Parallel Stream Internals: Configuring the Common Fork-Join Pool"

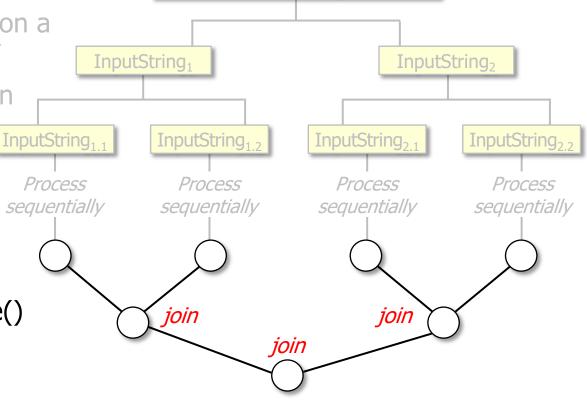
• The split-apply-combine phases are: **InputString** 1. Split – Recursively partition a data source into "chunks" InputString₁ InputString₂ 2. Apply – Process chunks in common fork-join pool InputString_{1 1} InputString_{1,2} InputString_{2,2} InputString_{2,1} **3. Combine** – Join partial **Process Process Process Process** results to a single result sequentially sequentially sequentially sequentially join

See upcoming lessons on "Java Parallel Stream Internals: Combining Results"

- The split-apply-combine phases are:

 InputString
 Apply Process chunks in

 InputString₁
 InputString₂
 - common fork-join pool3. Combine Join partial results to a single result
 - Performed by terminal operations
 - e.g., collect() & reduce()



See www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples

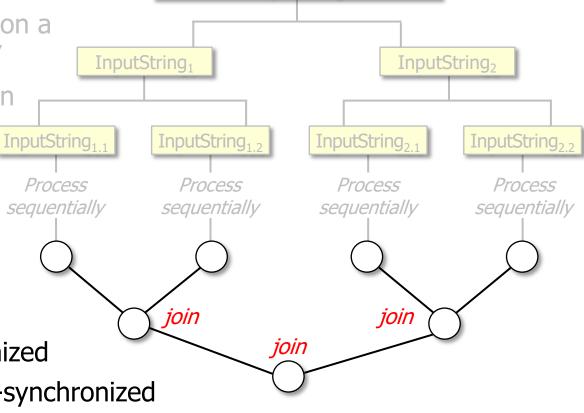
- The split-apply-combine phases are:

 InputString
 - The split-apply-combine phases are
 Split Recursively partition a
 - data source into "chunks"

 2. Apply Process chunks in
 - common fork-join pool3. Combine Join partial results to a single result
 - operations
 - · Collectors can either be

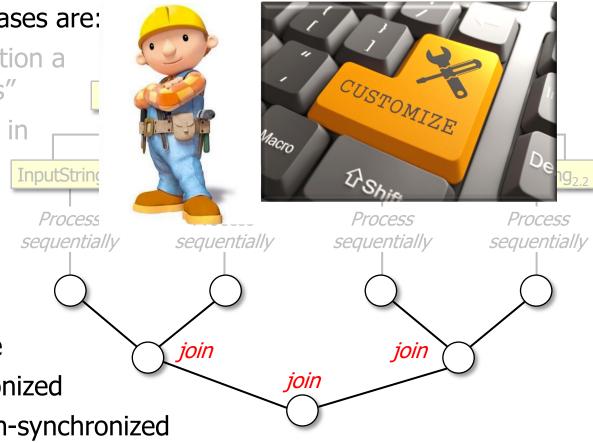
Performed by terminal

- Concurrent synchronized
- Non-concurrent non-synchronized



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

- The split-apply-combine phases are:
 - Split Recursively partition a data source into "chunks"
 - 2. Apply Process chunks in common fork-join pool
 - **3. Combine** Join partial results to a single result
 - Performed by terminal operations
 - Collectors can either be
 - Concurrent synchronized
 - Non-concurrent non-synchronized



Programmers can define custom collectors

End of How Java Parallel Streams Work "Under the Hood"