Java Parallel Stream Internals:  
Partitioning

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

- Understand parallel stream internals, e.g.
- Know what can change & what can’t
- Partition a data source into “chunks”

Partitioning a Parallel Stream
Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java parallel stream into chunks

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)
We’ve shown how a spliterator can traverse elements in a source

List<String> quote = Arrays.asList("This ", "above ", "all- ", "to ", "thine ", "own ", "self ", "be ", "true", ",\n", ...");

for(Spliterator<String> s = 
    quote.spliterator();
    s.tryAdvance(System.out::print)
    != false;
)
    continue;
We now outline how a parallel spliterator can \textit{partition} all elements in a source.
We now outline how a parallel spliterator can *partition* all elements in a source.

The streams framework calls a spliterator’s `trySplit()` method, not a user’s app.
Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

```java
Spliterator<T> trySplit() {
    if (input is <= minimum size)
        return null
    else {
        split input in 2 (even-sized) chunks
        return a spliterator for "left chunk"
    }
}
```

*trySplit()* attempts to split the input evenly (if it’s not <= the minimum size)
• We now outline how a parallel spliterator can *partition* all elements in a source

A spliterator usually needs no synchronization nor does it need a “join” phase!
We now outline how a parallel spliterator can partition all elements in a source

Partitioning a Parallel Stream

```java
Spliterator<T> trySplit() {
    if (input is <= minimum size)
        return null
    else {
        split input in 2 (even-sized) chunks
        return a spliterator for “left chunk”
    }
}
```

trySplit() is called recursively until all chunks are <= to the minimize size.
We now outline how a parallel spliterator can partition all elements in a source.

```
Spliterator<T> trySplit() {
    if (input is <= minimum size)
        return null
    else {
        split input in 2 (even-sized) chunks
        return a spliterator for “left chunk”
    }
}
```

trySplit() is finished when a chunk is <= to the minimum size.
We now outline how a parallel spliterator can *partition* all elements in a source

```java
Spliterator<T> trySplit() {
    if (input is <= minimum size)
        return null
    else {
        split input in 2 (even-sized) chunks
        return a spliterator for “left chunk”
    }
}
```

When null is returned the streams framework processes this chunk sequentially.
Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

```java
ArrayListSpliterator<E> trySplit() {
    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    // divide range in half unless too small
    return lo >= mid ? null : new ArrayListSpliterator<E>(list, lo, index = mid, ...);
}

boolean tryAdvance(Consumer<? super E> action) {
    ...
    if (index < getFence()) {
        action.accept((E) list.elementData[i]); ...
        return true;
    } return false;
}
```

See [openjdk/8u40-b25/java/util/ArrayList.java](openjdk/8u40-b25/java/util/ArrayList.java)
Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

```java
ArrayListSpliterator<E> trySplit() {
    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    // divide range in half unless too small
    return lo >= mid ? null : new ArrayListSpliterator<E>(list, lo, index = mid, ...);
}

Split the array evenly each time until there’s nothing left to split
```

```java
boolean tryAdvance(Consumer<? super E> action) {
    ...
    if (index < getFence()) {
        action.accept((E) list.elementData[i]); ...
        return true;
    }
    return false;
}
```
Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

```java
ArrayListSpliterator<E> trySplit() {
    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    // divide range in half unless too small
    return lo >= mid ? null : new ArrayListSpliterator<E>(list, lo, index = mid, ...);
}

boolean tryAdvance(Consumer<? super E> action) {
...  
    if (index < getFence()) {
        action.accept((E) list.elementData[i]); ...
        return true;
    }
    return false;
}
```

Try to consume a single element on each call
Partitioning a Parallel Stream

- Other Java collections do not split evenly & efficiently, e.g., LinkedList

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

boolean tryAdvance(Consumer<? super E> action) { ...
    Node\<E\> p;
    if (getEst() > 0 && (p = current) != null) {
        --est; E e = p.item; current = p.next;
        action.accept(e); return true;
    } return false;
}

See openjdk/8u40-b25/java/util/LinkedList.java
Partitioning a Parallel Stream

- Other Java collections do not split evenly & efficiently, e.g., LinkedList

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

\[Split\ the\ list\ into\ "batches",\ rather\ than\ evenly\ in\ half\]

```java
boolean tryAdvance(Consumer<? super E> action) { ...  
    Node<? super E> p;  
    if (getEst() > 0 && (p = current) != null) {  
        --est; E e = p.item; current = p.next;  
        action.accept(e); return true;  
    } return false;  
}
```
Partitioning a Parallel Stream

- Other Java collections do not split evenly & efficiently, e.g., LinkedList

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

boolean tryAdvance(Consumer<? super E> action) { ...
    Node<E> p;
    if (getEst() > 0 && (p = current) != null) {
        --est; E e = p.item; current = p.next;
        action.accept(e); return true;
    }
    return false;
}
```

Try to consume a single element on each call
Partitioning a Parallel Stream

• We’ll cover the implementation details of parallel spliterators in upcoming lessons.

```
Partitioning a Parallel Stream
```

```
Input Strings to Search
```

```
Search Phrases
```

```
map(phrase -> searchForPhrase(…))
```

```
filter(not(SearchResults::isEmpty))
```

```
collect(toList())
```

See “Java SearchWithParallelSpliterator Example: trySplit()”
End of Java Parallel Stream Internals: Partitioning