Transitioning to Parallelism & 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

• Know how aggregate operations from Java sequential streams are applied in the parallel streams framework
Transitioning from Sequential Streams to Parallel Streams
Transitioning from Sequential Streams to Parallel Streams

- A Java stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)

```
List <String>
Stream <String>
Stream <SearchResults>
Stream <SearchResults>
List <SearchResults>
```

- `List <String>`
- `Stream <String>`
- `Stream <SearchResults>`
- `Stream <SearchResults>`
- `List <SearchResults>`

```
Stream
<SearchResults>
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())
```

Transitioning from Sequential Streams to Parallel Streams

• A Java stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)

```
Stream<SearchResults>
Stream<String>
List<SearchResults>
List<String>
```

Aggregate operations use internal iteration & behaviors to process elements in a stream
Transitioning from Sequential Streams to Parallel Streams

• By default, a stream executes sequentially, so all its aggregate operations run behaviors in a single thread of control.
When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in the common fork-join pool.

Transitioning from Sequential Streams to Parallel Streams

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html
When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in the common fork-join pool.

Threads in the fork-join pool (non-deterministically) process different chunks.

Transitioning from Sequential Streams to Parallel Streams

```
List <String>
Stream <String>
Stream <SearchResults>
Stream <SearchResults>
List <SearchResults>
```

```
parallelStream()
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())
```
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in the common fork-join pool.

```
List <String>

Stream <String>

Stream <SearchResults>

List <SearchResults>
```

```
parallelStream()

map(phrase -> searchForPhrase(...))

filter(not(SearchResults::isEmpty))

collect(toList())
```

*Intermediate operations iterate over & process behaviors on these chunks in parallel*
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in the common fork-join pool.

```
List <String>
Stream <String>
Stream <SearchResults>
Stream <SearchResults>
List <SearchResults>
```

```
parallelStream()
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())
```

A terminal operation then combines the chunks into a single result.
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in the common fork-join pool.

(Stateless) Java lambda expressions & method references are used to pass behaviors.
When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in the common fork-join pool.

Transitioning from Sequential Streams to Parallel Streams

Ideally, minuscule changes are needed to transition from sequential to parallel stream.
Transitioning from Sequential Streams to Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>allMatch(Predicate&lt;? super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns whether all elements of this stream match the provided predicate.</td>
</tr>
<tr>
<td>boolean</td>
<td>anyMatch(Predicate&lt;? super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns whether any elements of this stream match the provided predicate.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream.Builder&lt;T&gt;</td>
<td>builder()</td>
</tr>
<tr>
<td></td>
<td>Returns a builder for a Stream.</td>
</tr>
<tr>
<td>&lt;R, A&gt; R</td>
<td>collect(Collectors&lt;super T, A, R&gt; collector)</td>
</tr>
<tr>
<td></td>
<td>Performs a mutable reduction operation on the elements of this stream using a Collector.</td>
</tr>
<tr>
<td></td>
<td>Performs a mutable reduction operation on the elements of this stream.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>concat(Stream&lt;? extends T&gt; a, Stream&lt;? extends T&gt; b)</td>
</tr>
<tr>
<td></td>
<td>Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.</td>
</tr>
<tr>
<td>long</td>
<td>count()</td>
</tr>
<tr>
<td></td>
<td>Returns the count of elements in this stream.</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>distinct()</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code>) of this stream.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>empty()</td>
</tr>
<tr>
<td></td>
<td>Returns an empty sequential Stream.</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>filter(Predicate&lt;? super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the elements of this stream that match the given predicate.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findAny()</td>
</tr>
<tr>
<td></td>
<td>Returns an <code>Optional</code> describing some element of the stream, or an empty <code>Optional</code> if the stream is empty.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findFirst()</td>
</tr>
<tr>
<td></td>
<td>Returns an <code>Optional</code> describing the first element of this stream, or an empty <code>Optional</code> if the stream is empty.</td>
</tr>
<tr>
<td>&lt;R&gt; Stream&lt;R&gt;</td>
<td>flatMap(Function&lt;? super T, ? extends Stream&lt;? extends R&gt;&gt; mapper)</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)
The same aggregate operations can be used for sequential & parallel streams.

*Example*: `SearchStreamGang` uses the same aggregate operations for both `SearchWithSequentialStreams` & `SearchWithParallelStreams` implementations.

```java
<<Java Class>>
SearchWithSequentialStreams

- processStream(): List<List<SearchResults>>
- processInput(String): List<SearchResults>
```

```java
<<Java Class>>
SearchWithParallelStreams

- processStream(): List<List<SearchResults>>
- processInput(CharArraySequence): List<SearchResults>
```

**Transitioning from Sequential Streams to Parallel Streams**

- `stream()` vs. `parallelStream()`
- `map(phrase -> searchForPhrase(…))`
- `filter(not(SearchResults::isEmpty))`
- `collect(toList())`
The same aggregate operations can be used for sequential & parallel streams
Java streams can thus treat parallelism as an optimization & leverage all available cores!

Transitioning from Sequential Streams to Parallel Streams

• The same aggregate operations can be used for sequential & parallel streams

• Java streams can thus treat parallelism as an optimization & leverage all available cores!

• Behaviors run by aggregate operations must be designed carefully to avoid accessing unsynchronized shared mutable data.

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html
End of Transitioning to Parallelism & Parallel Streams