An Overview of Parallelism & 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

- Know how aggregate operations from Java sequential streams are applied seamlessly in the Java parallel streams framework.

```
Input x

Intermediate operation (behavior f)
Output f(x)

Intermediate operation (behavior g)
Output g(f(x))

Terminal operation (reducer)
```

Stream factory operation ()

Output f(x)

Output g(f(x))
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 <SearchResults>
Stream <SearchResults>
Stream <String>
List <String>
```

- `Stream <SearchResults>`
  - **stream()**
  - **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”)

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

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

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

List `<String>`

Stream `<String>`

Stream `<SearchResults>`

Stream `<SearchResults>`

List `<SearchResults>`

Search Phrases

`stream()`

`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 “chunks” that run in the common fork-join pool

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html)
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple “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

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

Intermediate operations cleverly 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 “chunks” that run in the common fork-join pool

```
List <String>

Stream <String>

Stream <SearchResults>

Stream <SearchResults>

List <SearchResults>
```

A terminal operation triggers processing & combines the chunks into a single result

```
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 “chunks” that run in the common fork-join pool

(Stateless) Java lambda expressions & method references are used to pass behaviors
Transitioning from Sequential Streams to Parallel Streams

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

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

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

```
stream() vs. parallelStream()
```

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

`List <String>`

`Stream <String>`

`Stream <SearchResults>`

`Stream <SearchResults>`

`List <SearchResults>`

`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>&lt;R&gt; R</td>
<td>collect(Supplier&lt;? -&gt; R&gt; supplier, BiConsumer&lt;? -&gt; R&gt; accumulator, BiConsumer&lt;? -&gt; R&gt; combiner)</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 Object.equals(Object)) 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 Optional describing some element of the stream, or an empty Optional if the stream is empty.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findFirst()</td>
</tr>
<tr>
<td></td>
<td>Returns an Optional describing the first element of this stream, or an empty Optional 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>
The same aggregate operations can be used for sequential & parallel streams

e.g., SearchStreamGang uses the same aggregate operations for both SearchWithSequentialStreams & SearchWithParallelStreams implementations

<<Java Class>>

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

<<Java Class>>

SearchWithParallelStreams
- processStream(): List<List<SearchResults>>
- processInput(CharSequence): List<SearchResults>

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

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
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..
- An easy way to avoid shared mutable data is to use stateless behaviors

See en.wikipedia.org/wiki/Side_effect_(computer_science)
End of An Overview of Parallelism & Java Parallel Streams