Overview of Java Parallel Streams: Transitioning to Parallelism

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.

Diagram:

- **Input x**
  - Intermediate operation (behavior f)
    - Output f(x)
  - Intermediate operation (behavior g)
    - Output g(f(x))
- Terminal operation (reducer)

Legend:

- Stream factory operation ()
- Input x
- 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 <String>
Stream <String>
Stream <SearchResults>
Stream <SearchResults>
List <SearchResults>
```

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

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

**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.

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

```
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.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html](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 substream “chunks” that run in the common fork-join pool.

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

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

- `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.

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 8 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 substream “chunks” that run in the common fork-join pool.

```
List <String>  stream() vs. parallelStream()

Stream <String>  map(phrase -> searchForPhrase(…))

Stream <SearchResults>  filter(not(SearchResults::isEmpty))

Stream <SearchResults>  collect(toList())

List <SearchResults>  
```

Ideally, minuscule changes 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>
</table>
| boolean           | allMatch(Predicate<? super T> predicate)  
Returns whether all elements of this stream match the provided predicate. |
| boolean           | anyMatch(Predicate<? super T> predicate)  
Returns whether any elements of this stream match the provided predicate. |
| static <T> Stream.Builder<T> | builder()  
Returns a builder for a Stream. |
| <R,A> R           | collect(Collectors.<super T,A,R> collector)  
Performs a mutable reduction operation on the elements of this stream using a Collector. |
Performs a mutable reduction operation on the elements of this stream. |
| static <T> Stream<T> | concat(Stream<? extends T> a, Stream<? extends T> b)  
Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. |
| long              | count()  
Returns the count of elements in this stream. |
| Stream<T>         | distinct()  
Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| static <T> Stream<T> | empty()  
Returns an empty sequential Stream. |
| Stream<T>         | filter(Predicate<? super T> predicate)  
Returns a stream consisting of the elements of this stream that match the given predicate. |
| Optional<T>       | findAny()  
Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| Optional<T>       | findFirst()  
Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. |
| <R> Stream<R>     | flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)  
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. |
Transitioning from Sequential Streams to Parallel Streams

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

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

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!

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!
- Naturally, behaviors run by these aggregate operations must be designed carefully to avoid accessing unsynchronized shared state..

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