Java 8 Parallel SearchStreamGang Example (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

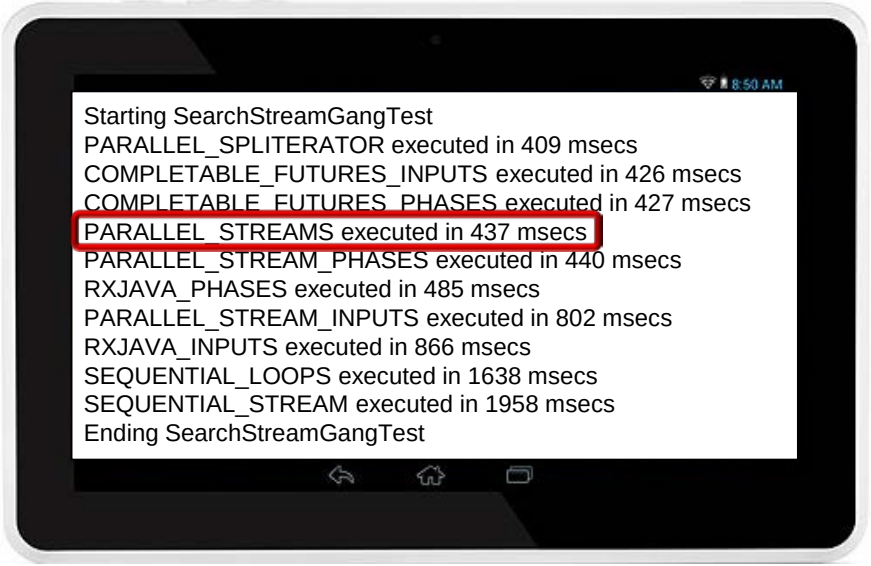
Professor of Computer Science

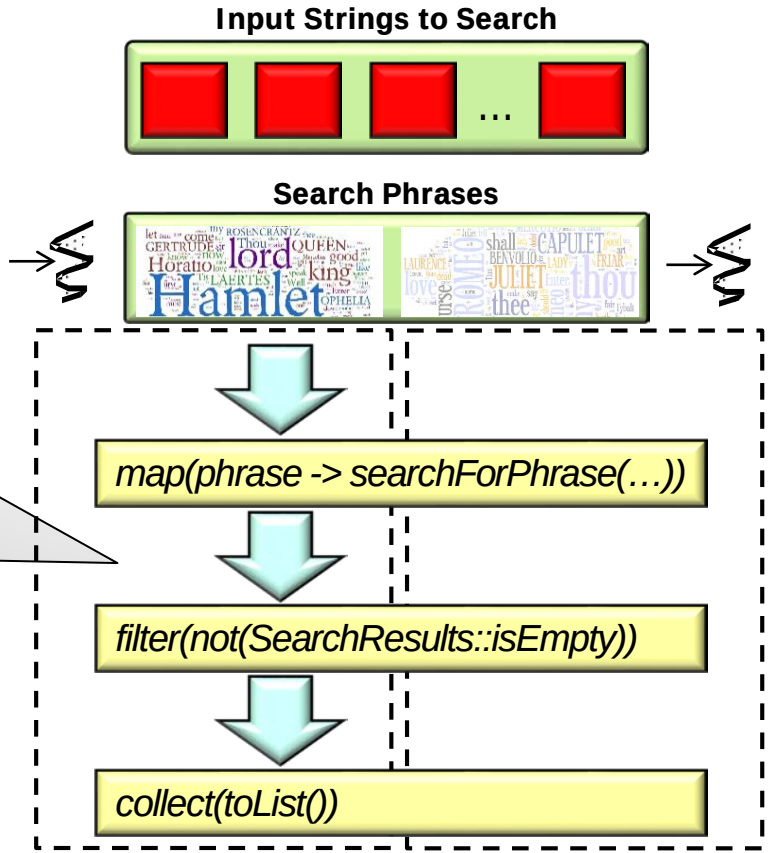
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Know how Java 8 parallel streams are applied in the SearchStreamGang





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

Learning Objectives in this Part of the Lesson

- Know how Java 8 parallel streams are applied in the SearchStreamGang
- Understand the pros & cons of the SearchWithParallelStreams class

<<Java Class>>
SearchWithParallelStreams

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

See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithParallelStreams.java

Applying Parallel Streams to SearchStreamGang

Applying Parallel Streams to SearchStreamGang

- SearchWithParallelStreams contains two parallel streams

<<Java Class>>
SearchWithParallelStreams

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

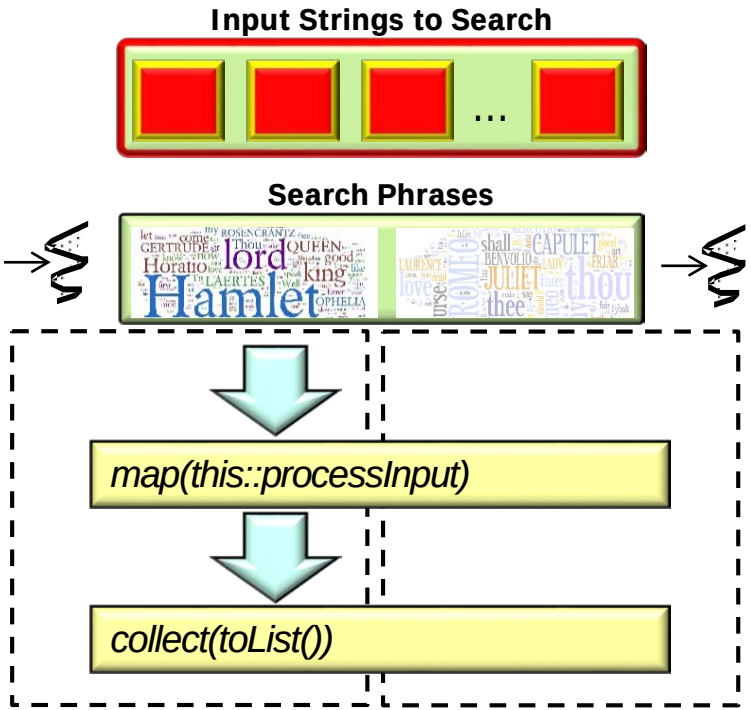
```
getInput()
   .parallelStream()
   .map(this::processInput)
   .collect(toList());
```

```
return mPhrasesToFind
   .parallelStream()
   .map(phrase -> searchForPhrase(phrase, input, title, false))
   .filter(not(SearchResults::isEmpty)
   .collect(toList());
```

See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithParallelStreams.java

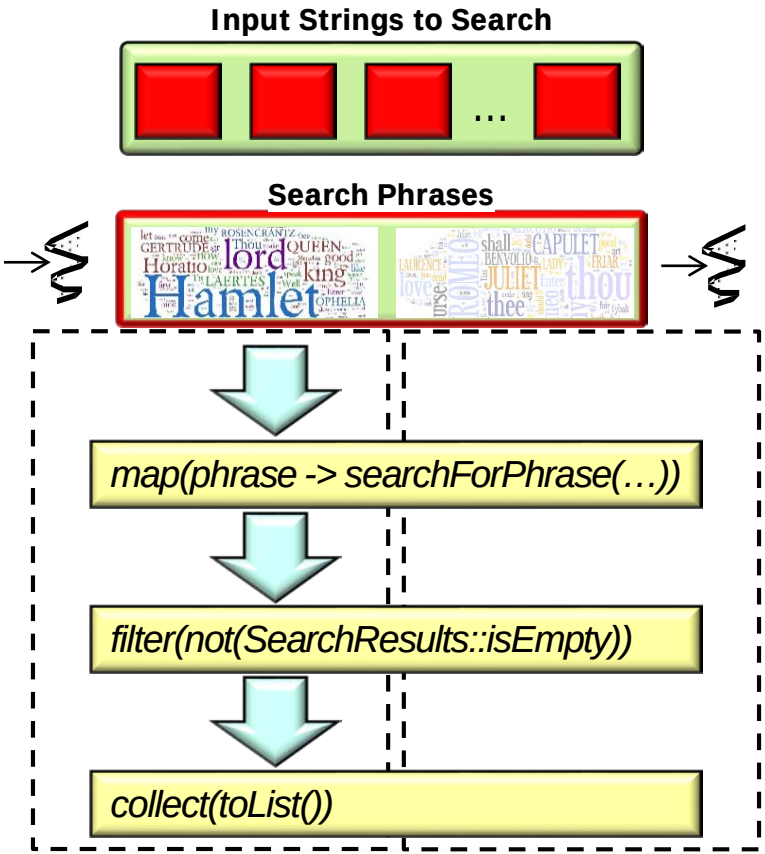
Applying Parallel Streams to SearchStreamGang

- SearchWithParallelStreams contains two parallel streams
  - **processStream()**
    - Uses a parallel stream to search a list of input strings in parallel



Applying Parallel Streams to SearchStreamGang

- SearchWithParallelStreams contains two parallel streams
  - **processStream()**
  - **processInput()**
    - Uses a parallel stream to search each input string to locate all occurrences of phases



Java 8 Parallel Stream processStream() Implementation

Java 8 Parallel Stream processStream() Implementation

- This processStream() implementation has one minuscule change

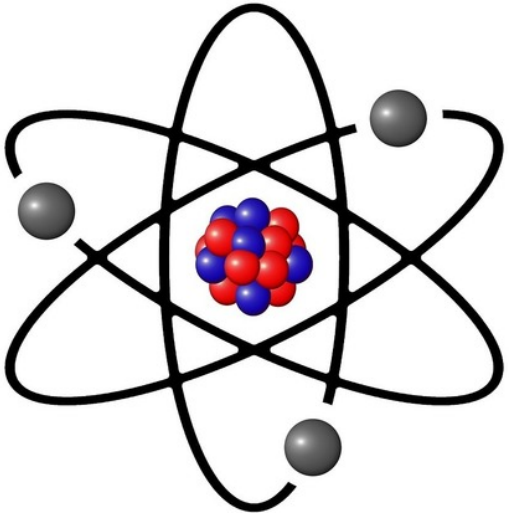
```
protected List<List<SearchResults>> processStream() {
  List<CharSequence> inputList =
    getInput();

  return getInput()

    .parallelStream()

    .map(this::processInput)

    .collect(toList());
}
```



Java 8 Parallel Stream processStream() Implementation

- This processStream() implementation has one minuscule change

```
protected List<List<SearchResults>> processStream() {
  List<CharSequence> inputList =
    getInput();

  return getInput()

    .parallelStream()

    .map(this::processInput)

    .collect(toList());
}
```

Creates a parallel stream that searches a list of input strings in parallel

Java 8 Parallel Stream processStream() Implementation

- This processStream() implementation has one minuscule change

```
protected List<List<SearchResults>> processStream() {
  List<CharSequence> inputList =
    getInput();

  return getInput()

    .parallelStream()

    .map(this::processInput)

    .collect(toList());
}
```

Searches each input string to locate all occurrences of phases

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

Java 8 Parallel Stream processStream() Implementation

- This processStream() implementation has one minuscule change

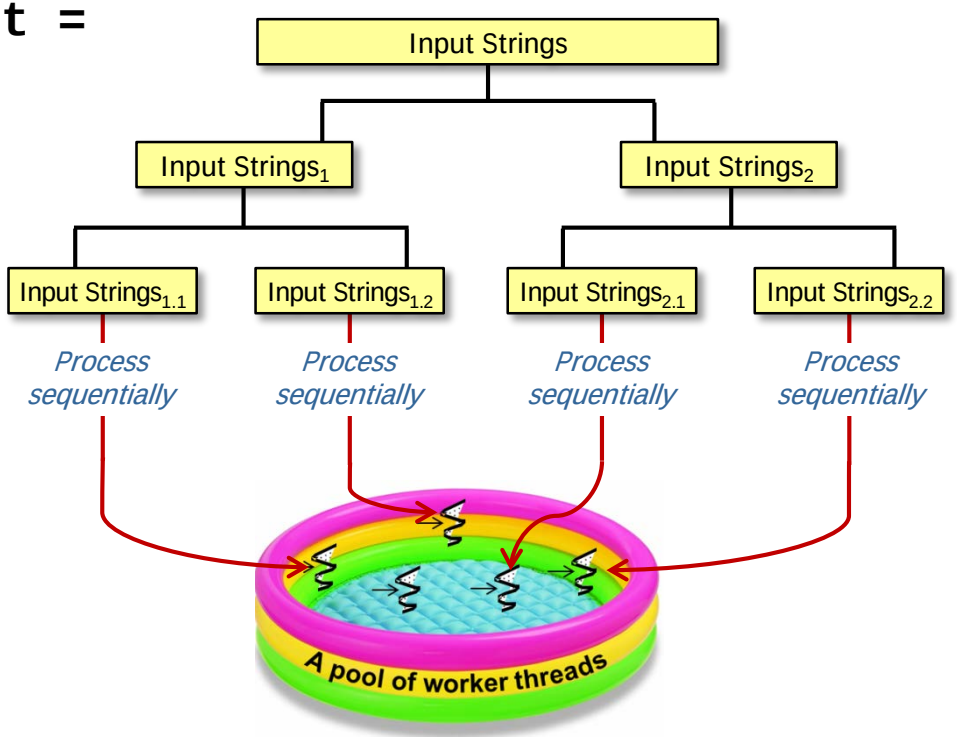
```
protected List<List<SearchResults>> processStream() {
  List<CharSequence> inputList =
    getInput();

  return getInput()

    .parallelStream()

    .map(this::processInput)

    .collect(toList());
}
```



"Chunks" of input strings are processed in parallel in the common fork-join pool

Java 8 Parallel Stream processStream() Implementation

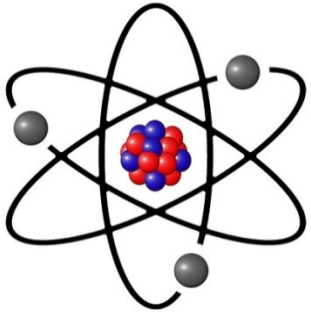
Java 8 Parallel Stream processInput() Implementation

- Likewise, this processInput() implementation has just one minuscule change

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(phase, input, title, false))
    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```



Java 8 Parallel Stream processInput() Implementation

- Likewise, this processInput() implementation has just one minuscule change

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(phase, input, title, false))
    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

Create a parallel stream that searches each input string to locate all occurrences of phases

Java 8 Parallel Stream processInput() Implementation

- Likewise, this processInput() implementation has just one minuscule change

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(phase, input, title, false))
    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

In this implementation strategy the spliterator is used to break the input into "chunks" that are processed sequentially

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

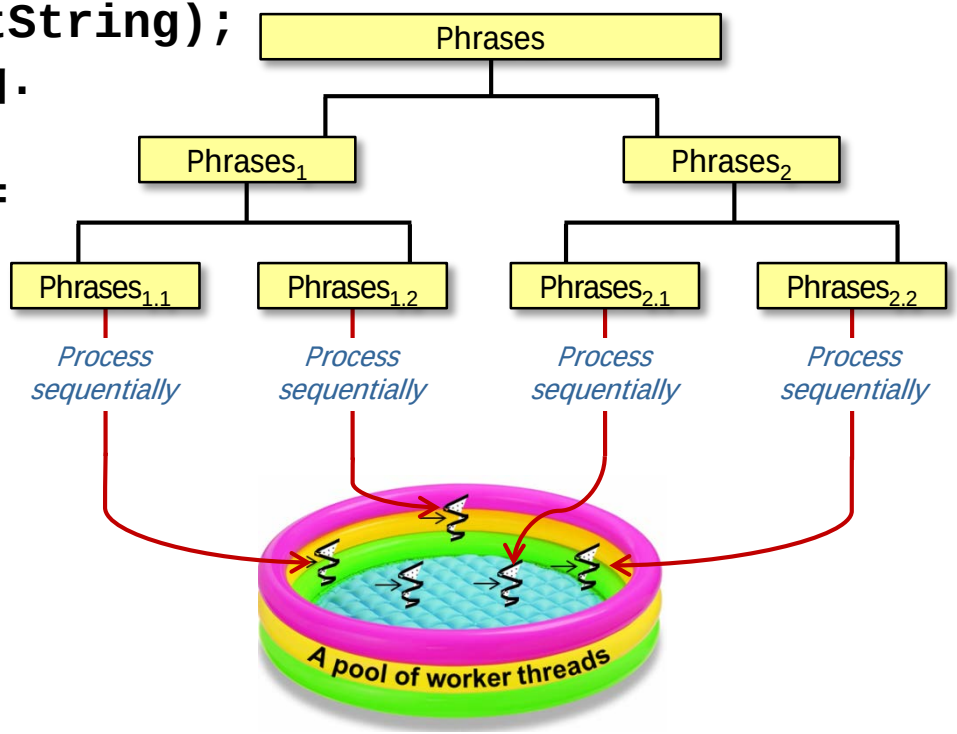
Java 8 Parallel Stream processInput() Implementation

- Likewise, this processInput() implementation has just one minuscule change

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.

  List<SearchResults> results =
    .parallelStream()
    .map(phase ->
         searchForPhrase)
    .filter(not(SearchResults

    .collect(toList());
  return results;
}
```



"Chunks" of phrases are processed in parallel in the common fork-join pool

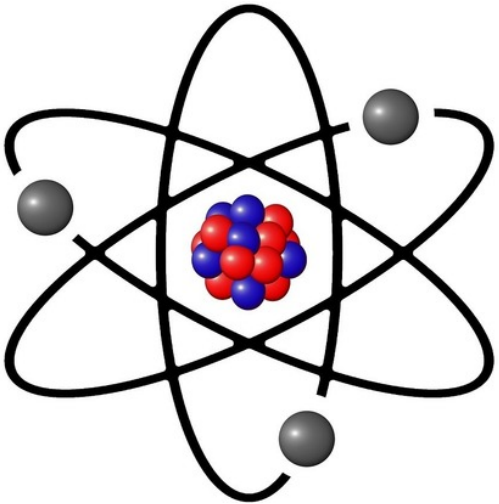
Pros of the SearchWith ParallelStreams Class

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!

<<Java Class>>
SearchWithParallelStreams

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



See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
}
```

Here's processStream() from SearchWithSequentialStream that we examined earlier

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
}
```

vs

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .parallelStream()
    .map(this::processInput)
    .collect(toList());
}
```

Here's processStream() in SearchWithParallelStreams

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!

```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
}
```

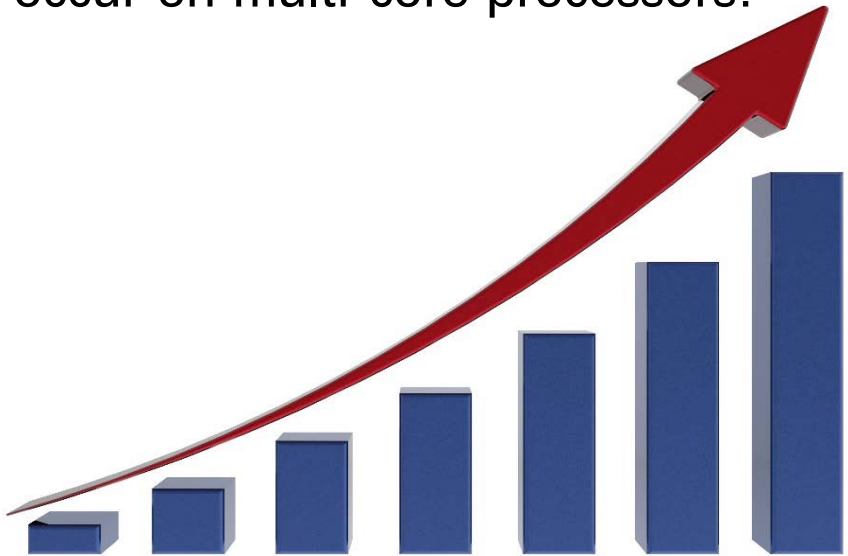
vs

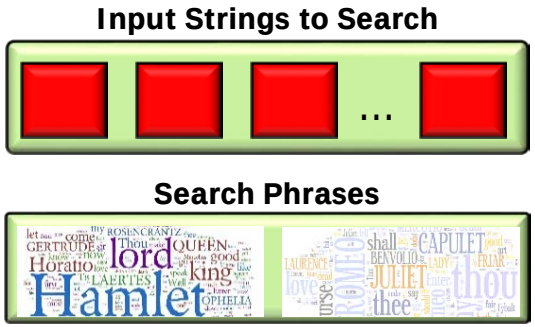
```
List<List<SearchResults>>
                processStream() {
  return getInput()
    .parallelStream()
    .map(this::processInput)
    .collect(toList());
}
```

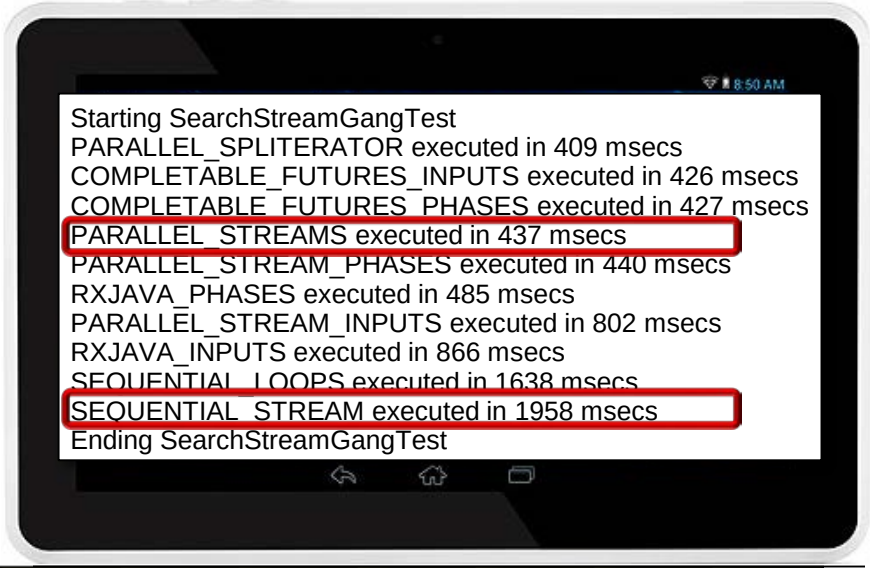
Changing all the stream() calls to parallelStream() calls is the only difference between implementations!!

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!
  - Moreover, substantial speedups can occur on multi-core processors!



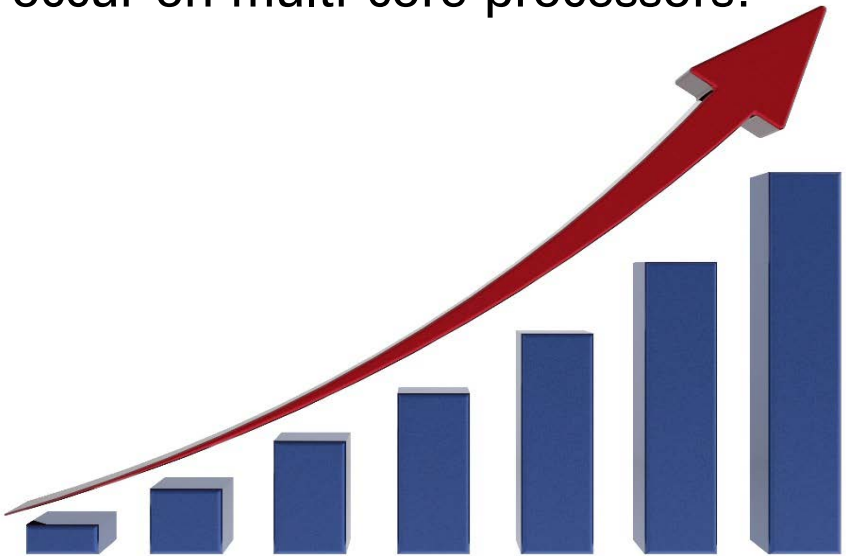


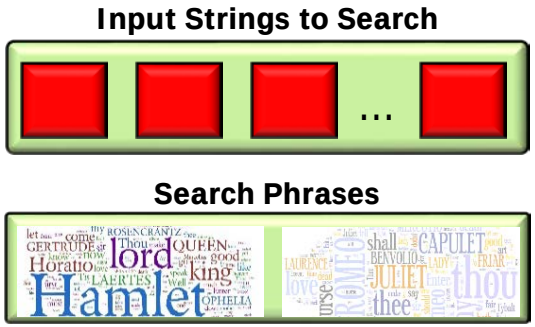


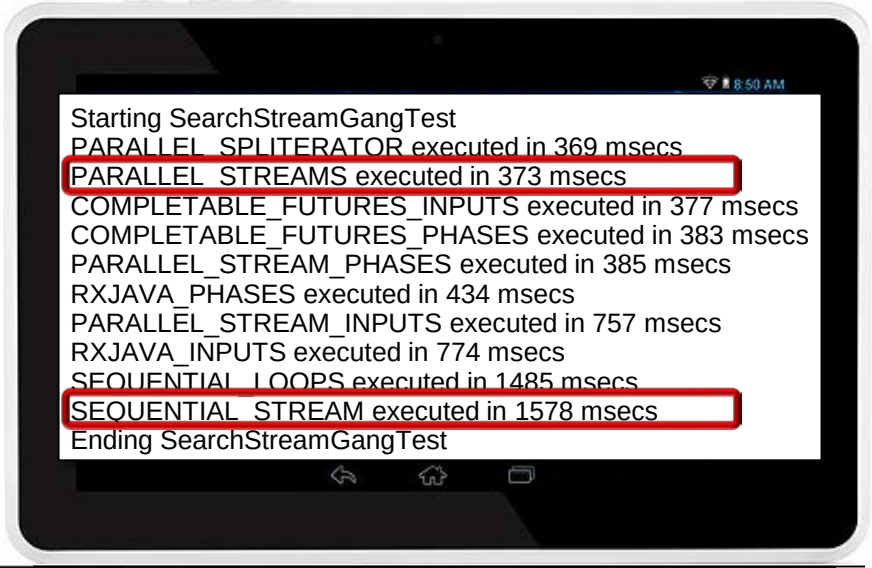
Tests conducted on a 2.7GHz quad-core Lenovo P50 with 32 Gbytes of RAM

Pros of the SearchWithParallelStreams Class

- This example shows that the difference between sequential & parallel streams is often minuscule!
  - Moreover, substantial speedups can occur on multi-core processors!





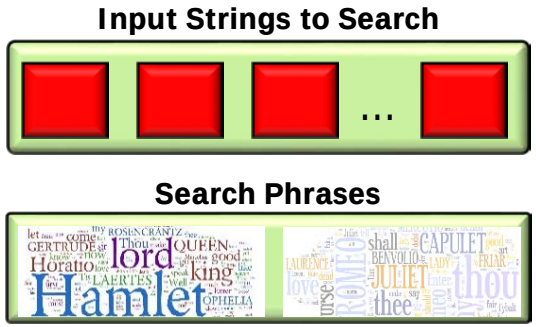


Tests conducted on a 2.9GHz quad-core MacBook Pro with 16 Gbytes of RAM

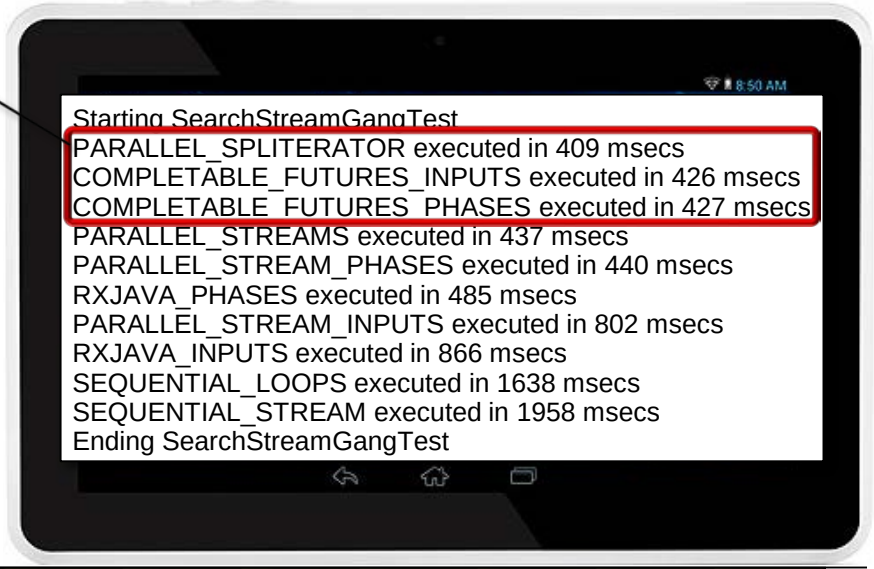
Cons of the SearchWith ParallelStreams Class

Cons of the SearchWithParallelStreams Class

- Just because two minuscule changes are needed doesn't mean this is the best implementation!



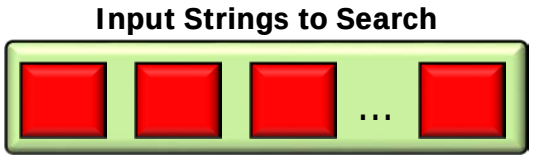
Other Java 8 concurrency strategies are even more efficient..

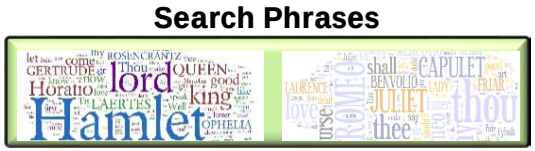


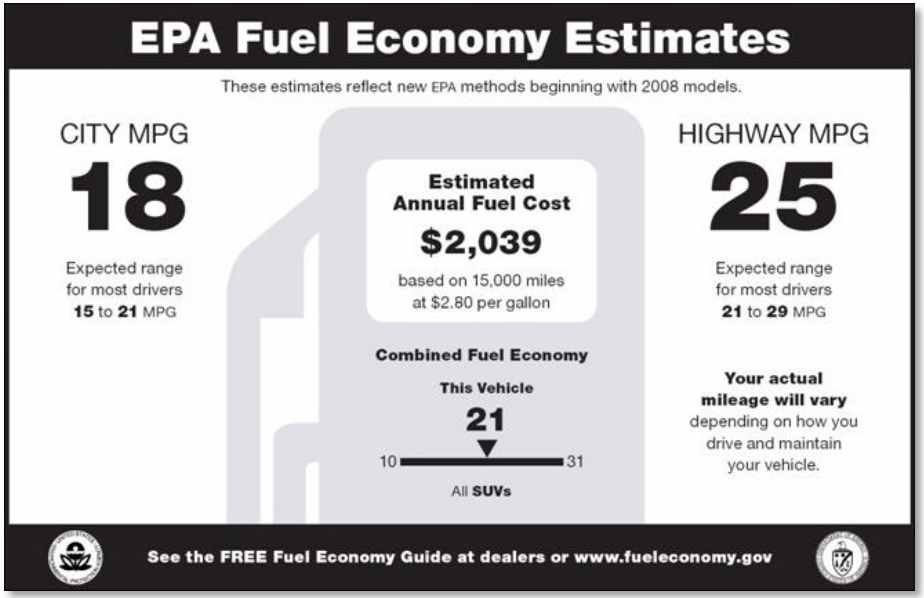
Tests conducted on a 2.7GHz quad-core Lenovo P50 with 32 Gbytes of RAM

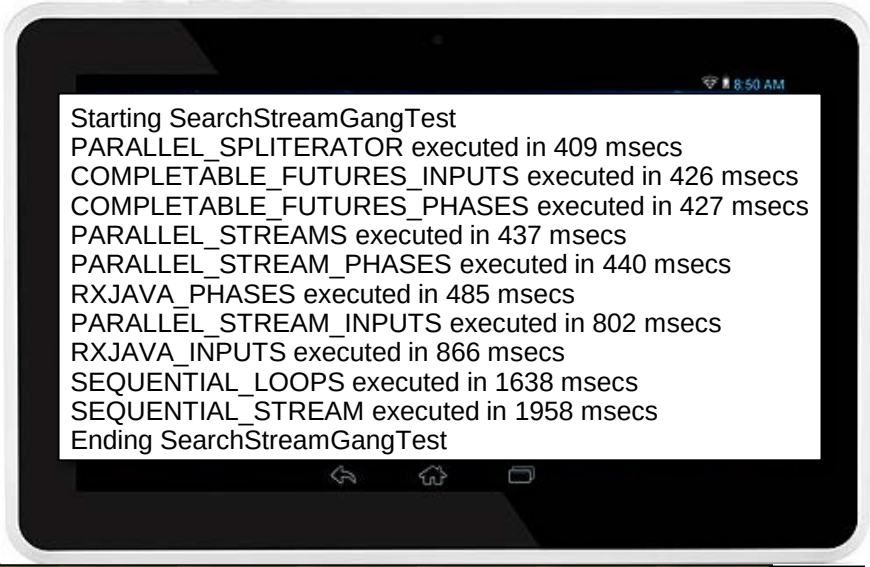
Cons of the SearchWithParallelStreams Class

- Just because two minuscule changes are needed doesn't mean this is the best implementation!









There's no substitute for systematic benchmarking & experimentation

End of Java 8 Parallel SearchStreamGang Example (Part 1)