Perchance to Stream with Java 8

Paul Sandoz
Oracle

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.

It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

Goals of this presentation

- Java SE 8 introduces lambda expressions
  - Among other benefits, lambdas enable better libraries
- Learn how lambda expressions are integrated into the core Collection libraries with the Streams API
- Understand how to get the best out of the Streams API
- This is the first of many potential improvements and optimizations to the Java platform

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Aggregate Operations

- Most business logic is about aggregate operations
- Historically, coded over collections with loops
  - Fundamentally sequential
  - Frustratingly imperative
- Java SE 8's answer: `Stream`
  - With some help from lambdas

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What is a Stream?
At the high level

- Abstraction for specifying aggregate computations
  - Not a data structure; a “view” over data
  - Can be infinite
- Up-levels descriptions of aggregate computations
  - Exposes opportunity for optimization
  - Fusing, laziness, and **parallelism** (map/reduce in the “small”)

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What is a Stream?
At the high level

- The Stream API is the first of (hopefully) many good kinds of stream-like APIs on Java 8
  - Has a particular model for sequential and parallel computation (and a particular implementation)
  - It's not the only model; and nor is **`Stream`** the only stream-like abstraction
- Functional-like patterns are important
  - **`map`**, **`filter`**, **`flatMap`**, **`reduce`**

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream Pipeline Anatomy

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

Stream sources

Many ways to create

- From collections and arrays
  - `Collection.stream()` or `Collection.parallelStream()`
  - `Arrays.stream()` or `Stream.of()`
- Static factories
  - `IntStream.range()`, `Files.walk()`
- Or, roll-your-own stream sources
  - with `java.util.Spliterator`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream sources

Manages three aspects

- Access to stream elements
- Decomposition
  - For parallel operations
- Stream characteristics
  - e.g. `SIZED`, `ORDERED`, `DISTINCT`, `SORTED`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream sources

A sampling of stream sources in the JDK

Source	Decomposibility	Characteristics
`ArrayList` (and arrays)	Excellent	`SIZED`, `ORDERED`
`LinkedList`	Poor	`SIZED`, `ORDERED`
`HashSet`	Good	`SIZED`, `DISTINCT`
`TreeSet`	Good	`SIZED`, `DISTINCT`, `SORTED`, `ORDERED`
`IntStream.range()`	Excellent	`SIZED`, `DISTINCT`, `SORTED`, `ORDERED`
`Stream.iterate()`	Poor	`ORDERED`
`BufferedReader.lines()`	Poor	`ORDERED`
`SplittableRandom.ints()`	Excellent	

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream Pipelines
Intermediate operations

- Are lazy
- Can affect pipeline characteristics
  - **`map()`** preserves **`SIZED`** but not **`DISTINCT`** or **`SORTED`**
- Some parallelize/fuse better than others
  - Stateless operations (**`map()`**, **`filter()`**, **`flatMap()`**) parallelize/fuse perfectly
  - Stateful operations (**`sorted()`**, **`distinct()`**, **`limit()`**) parallelize/fuse to varying degrees

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Intermediate Operations

Pipeline characteristics

	SIZED	ORDERED	DISTINCT	SORTED
`IntStream.range(M, N)`	✔	✔	✔	✔
`.filter(λ)`	✘	⬇✓	⬇✓	⬇✓
`.map(λ)`	⬇✗	⬇✓	✘	✘
`.unordered()`	⬇✗	✘	⬇✗	⬇✗
`.distinct()`	⬇✗	⬇✗	✔	⬇✗
`.sorted()`	⬇✗	✔	⬇✓	✔
`.limit(N)`	✘	⬇✓	⬇✓	⬇✓
`.peek()`	⬇✗	⬇✓	⬇✓	⬇✓

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Intermediation Operations

Operation	Effect	Notes
`filter()`	Removes `SIZED`	
`map()`	Removes `DISTINCT`, `SORTED`	
`flatMap()`	Removes `DISTINCT`, `SORTED`, `SIZED`	
`sorted()`	Injects `SORTED`, `ORDERED`	No-op if already `SORTED`
`distinct()`	Injects `DISTINCT`	No-op if already `DISTINCT`
`limit()`	Removes	Short-circuiting
`peek()`	Preserves all	
`unordered()`	Removes `ORDERED`	

*JDK implementation specific, could preserve all if source is bound to pipeline on intermediate operation rather than terminal operation

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stream Pipelines
Terminal operations

- Invoking a terminal operation executes the pipeline
  - All operations can execute in sequential or parallel
- Terminal operations can take advantage of pipeline characteristics
  - `toArray()` can avoid copying for `SIZED` pipelines by allocating resulting array up-front

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Terminal Operations

Family	Operation	Notes
Aggregation / Summary	`toArray`	
	`reduce`	
	`collect`	
	`sum, min, max, count`	
	`anyMatch, allMatch`	Short-circuiting
Iteration	`forEach`	Not order preserving
Searching	`findFirst`	Short-circuiting
	`findAny`	Short-circuiting, nondeterministic

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Rules of the Stream Game

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

The Rules of the Game

Just a few restrictions

- Behavioral parameters must be ***non-interfering***
  - Don’t interfere/modify the source during execution
  - Non-deterministic results, or `ConcurrentModificationException`
- Behavioral parameters must be ***stateless***
  - Don’t access any state that might change during pipeline execution, otherwise at risk of concurrency hazards
  - Enables pipelines to work *correctly* either in sequential or parallel

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Rules of the Game

- Exception to the rule
  - **`Consumer`** parameter to **`forEach()`** and **`peek()`**
  - Must operate by side-effect

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Streams

Why go parallel?

- Get results faster... maybe
  - Many chips and cores per machine; let's keep 'em hot!
  - GPU/SIMD (OpenJDK project Sumatra)
- !Do not assume parallel is always faster!
  - Easy to do, but not always the right thing to do
  - Sometimes slower than sequential
- Fork/Join is great, but too low-level
  - Stream framework uses Fork/Join under the hood

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Streams
Going parallel

- Almost always takes more work
- Sadly, not a magic bullet
  - Just saying (again :-)) `.parallel()` is not always faster
- Many factors will affect performance
  - Data size/decomposition/packing, #cores, cost-per-element
  - Sharing cores with other applications
  - Primitive streams included for performance reasons: boxing hurts!

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Streams

Going parallel

- Parallelism is explicit but unobtrusive
  - Sequential is the default
  - Parallelism may be faster but introduces non-determinism
  - Where possible sequential and parallel execution produce the "same" result
- A stream pipeline is created with an orientation of sequential or parallel
  - Can be changed with methods `parallel()` and `sequential()`
  - Applies to the whole pipeline; "Last call wins"

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Pipelines

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Parallel Streams
Details

- Parallel execution is greedy
  - Uses all the resources the Fork/Join (common) pool has to offer
- Parallelism of a pipeline cannot currently be configured
  - We need to refine this in a future Java release
- Perhaps the runtime should work out the best possible mode of execution
  - After N sequential executions, if gathered statistics are favorable then try a speculative parallel execution?
  - Sounds simple :-)

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Performance

Should i go parallel?

- Web apps (ironically) should not
  - Web/Apps servers should by default disable parallelism of Fork/Join common pool

```
System.setProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism",
    "0");
```

- Qualitative considerations
  - How good is the stream source decomposition?
  - Is the terminal operation merge step expensive?
  - What are the stream characteristics?

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Simple Performance Model

Quantitative considerations

```
    N   =  size of source data set
    Q   =  cost per-element through the stream pipeline
N * Q  ~=  cost of pipeline
```

- The larger **N * Q** is then higher likelihood of good parallel performance
- Easier to know **N** than **Q**
  - But can reason qualitatively about **Q**

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Intuition and Measurement

- For small data sets (**`N`** is small) sequential usually wins
- Simpler pipelines are easier "guesstimate"
  - "Do you have 10K elements or more", **`N > 10K`**, **`Q = 1`**
- Complex pipelines are harder to reason about
  - Stream source is derived from an **`Iterator`** (biased to low **`Q`**)
  - Pipeline contains a **`limit()`** operation
  - Complex reduction using **`groupingBy()`**

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If in doubt, measure!

- The JVM is very dynamic
  - Runtime compilation and optimization may affect measurement
  - This can be surprising
- Use a benchmark tool
  - Java Micro/Macro-benchmark/Measurement Harness, `jmh`
    - http://openjdk.java.net/projects/code-tools/jmh/

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh`
Measurement Examples

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh` measurement examples

- Dell Laptop
- JVM flags
  - `-XX:-TieredCompilation`
- Down-clocked CPU to 2GHz
  - `sudo cpufreq-set ...`
  - Intel's CPU behaves differently in single vs multi-threaded modes, which makes sequential and parallel results hard to compare

    "... nobody knows how it works deeply"

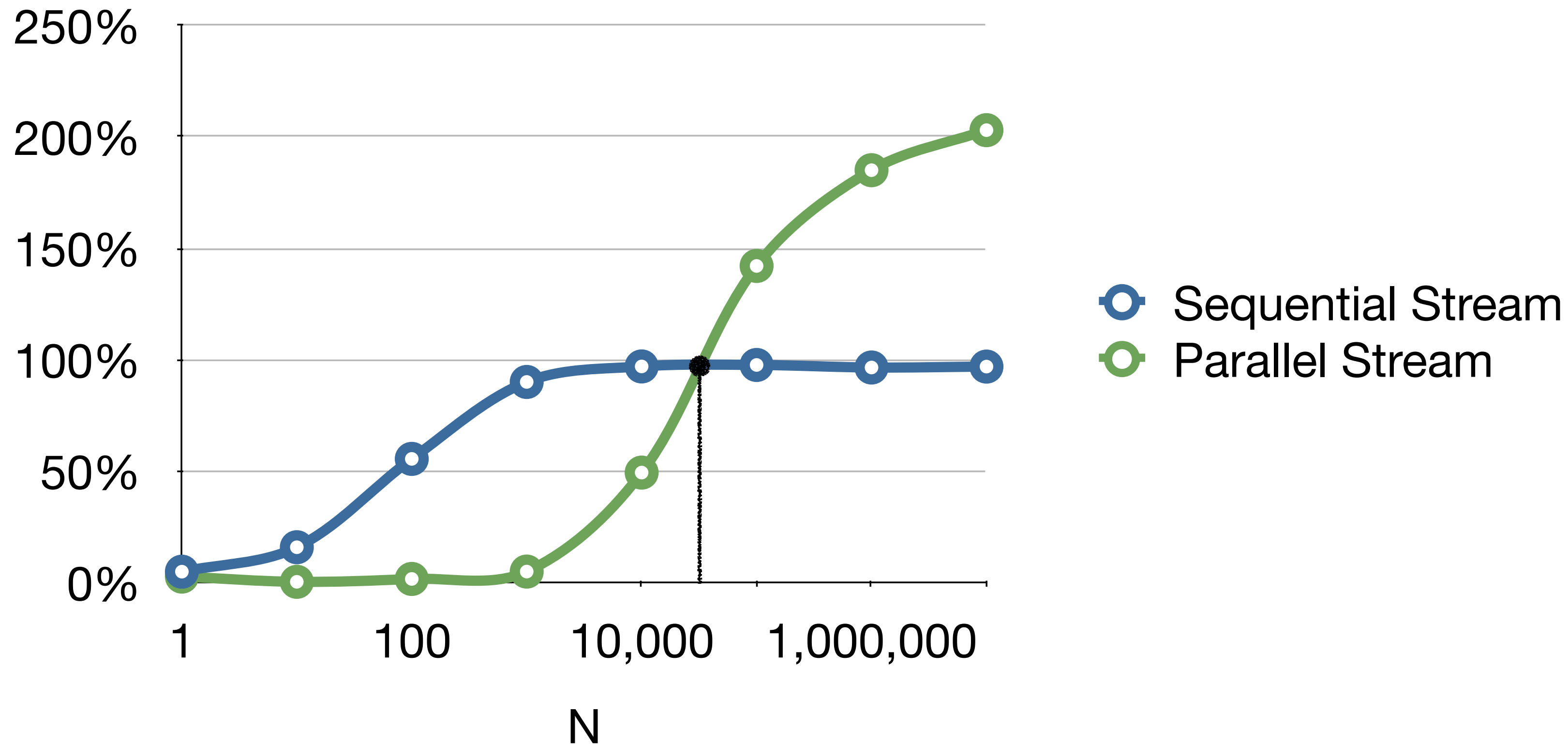
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Low Q: Map and sum of `int[]`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

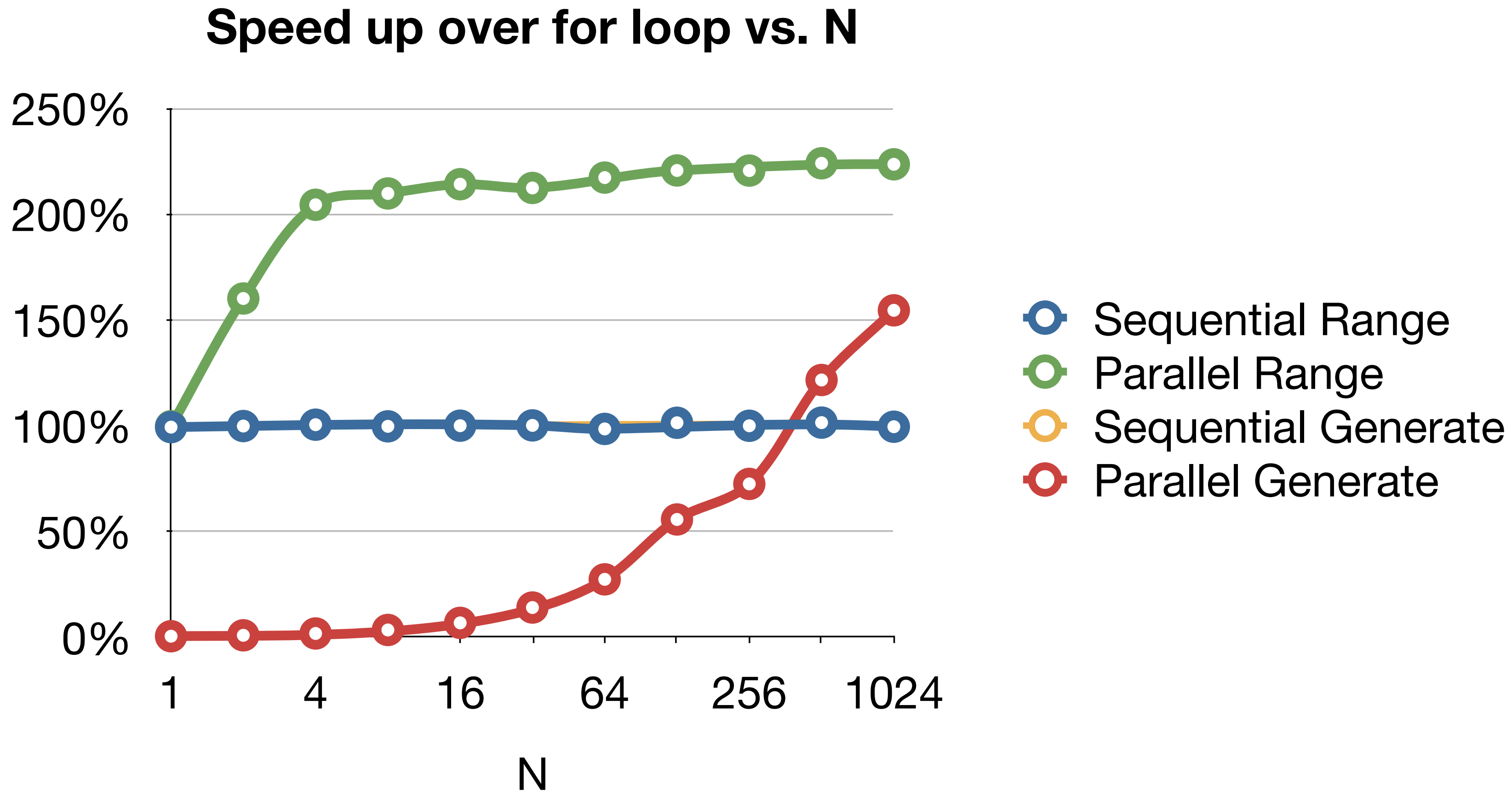
@PaulSandoz

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

High Q: Generating probable primes



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

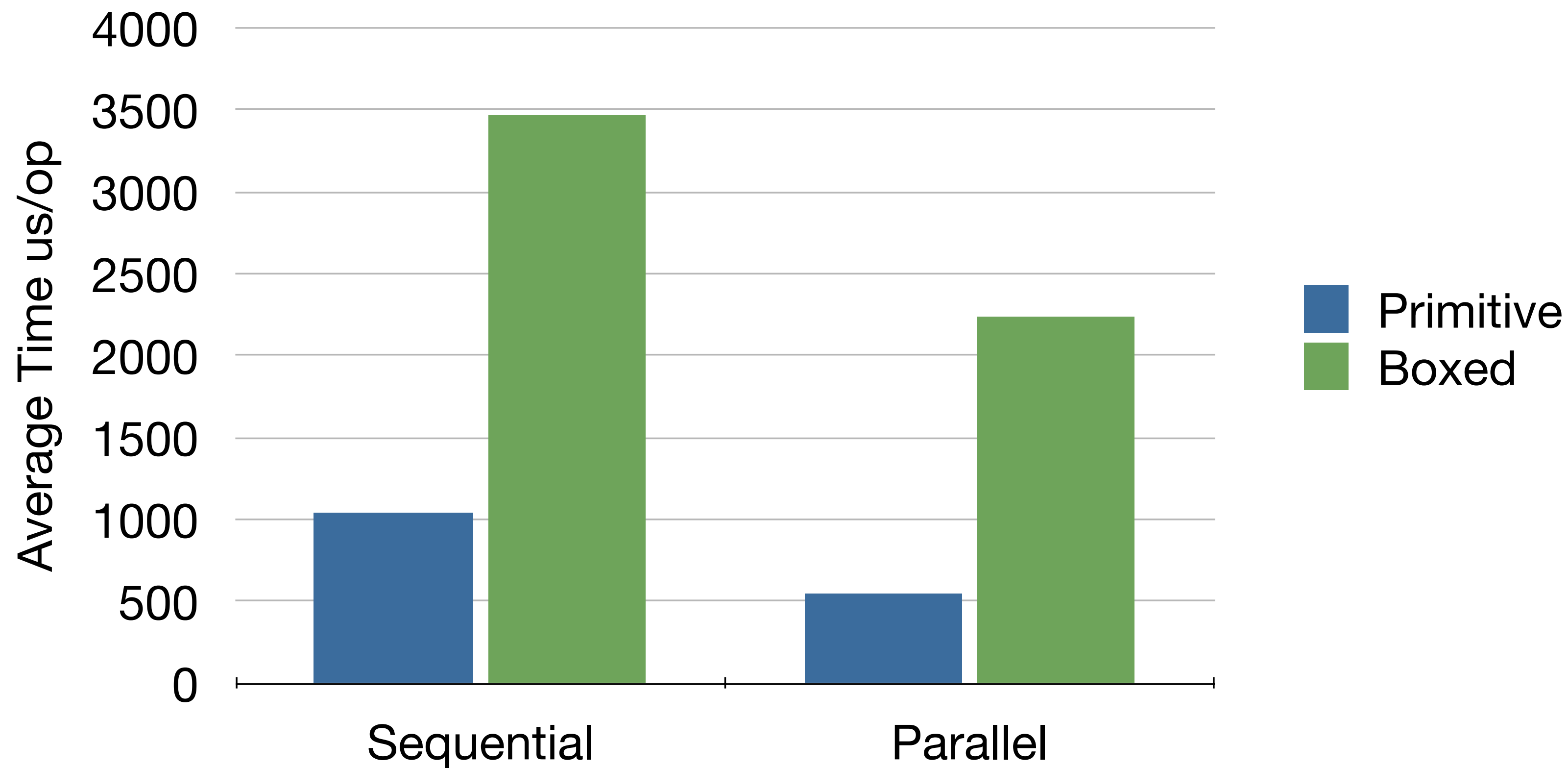
@PaulSandoz

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Boxing

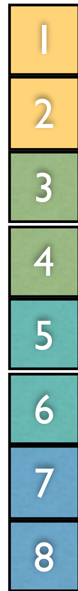


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

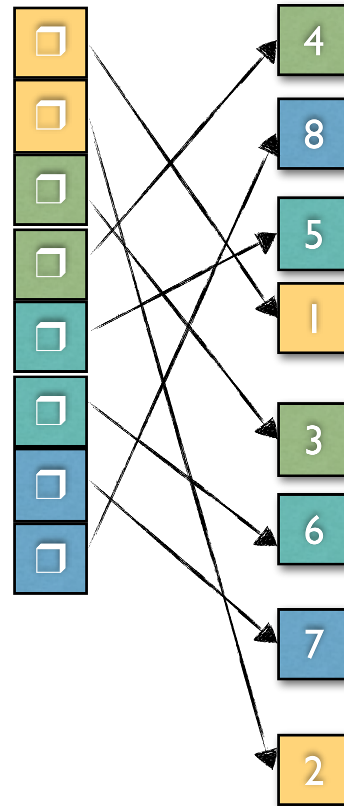
@PaulSandoz

Boxing; The Case For Value Types

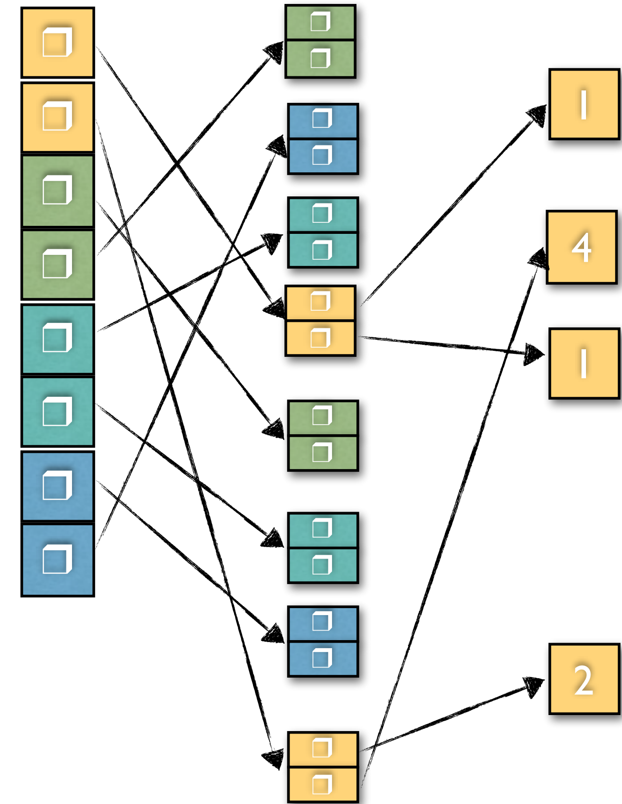
`int[]`



`Integer[]`



`Pair<Integer, Integer>[]`

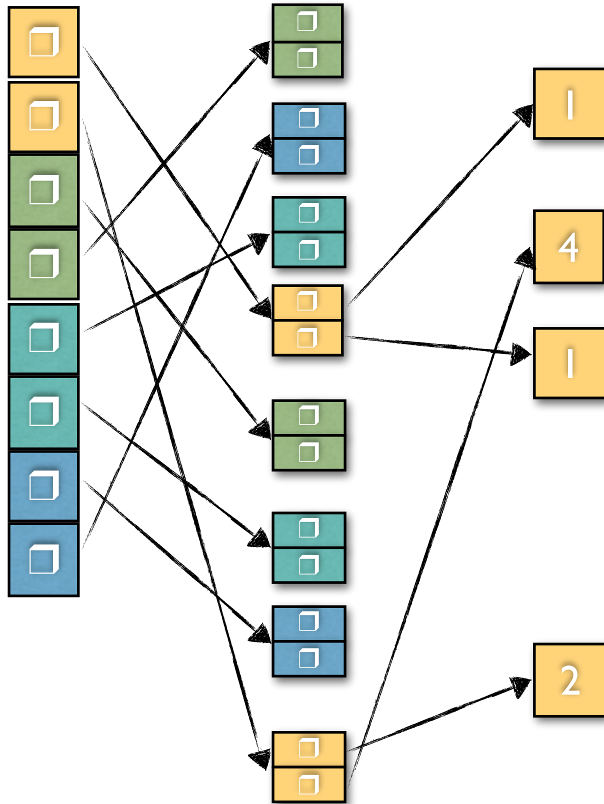


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

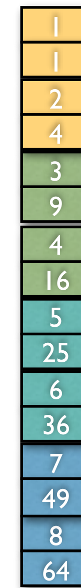
@PaulSandoz

Boxing; The Case For Value Types

`Pair<Integer, Integer>[]`



`Pair<int, int>[]`

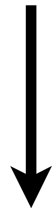


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Value Type Stream Nirvana?

Wearing my tin foil hat... pure speculation right now

```
Stream<Pair<int, int>> s = ...;
s.filter(p -> p.x > p.y)
 .map(p -> Pair.of(p.x + p.x, p.y)
 ...
```



Hotspot runtime unboxing and inlining

```
Stream<Pair<int, int>> s = ...;
s.filter((x, y) -> x > y)
 .map((x, y) -> return (x + x, y))
 ...
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

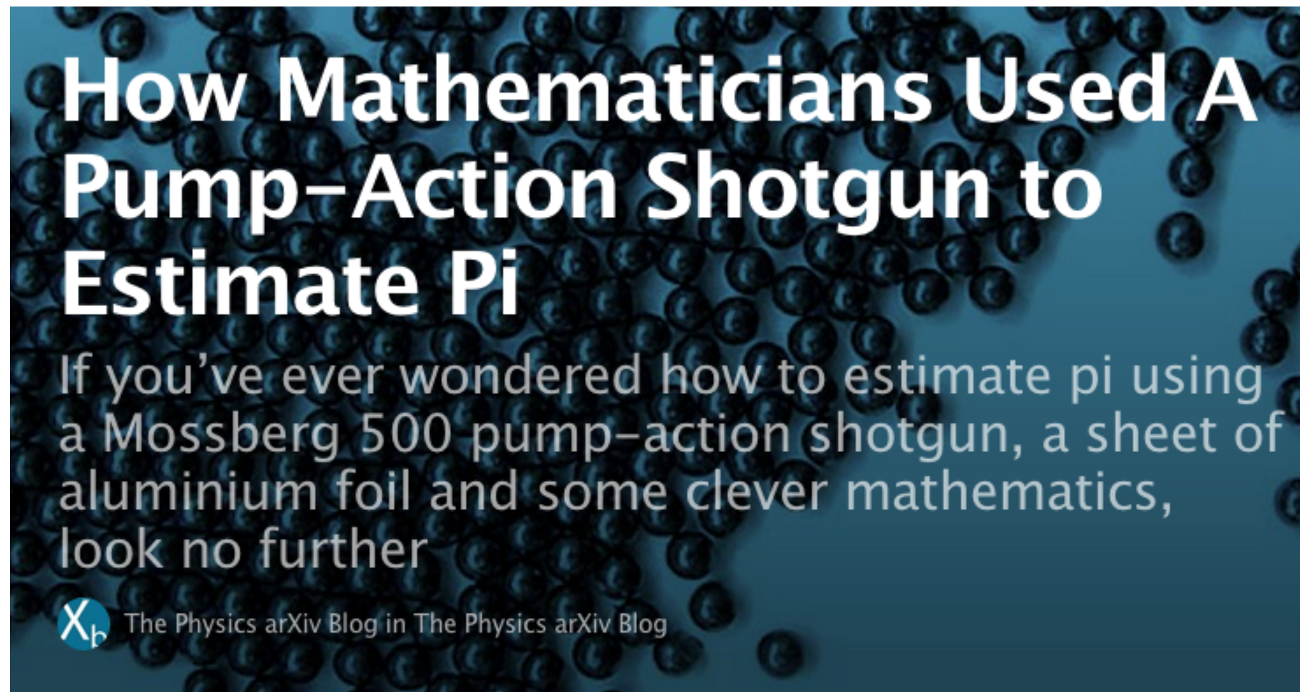
`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Monte Carlo calculation of π

The slow, but arguably, more fun way

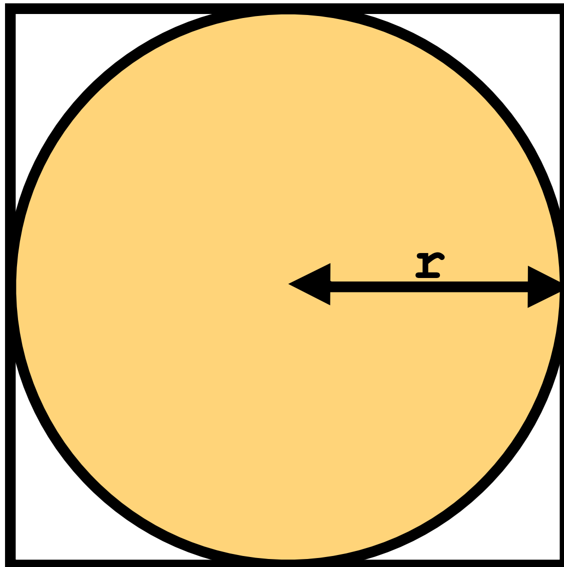


https://medium.com/the-physics-arxiv-blog/c1eb776193ef

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Monte Carlo calculation of π

The fast easy way with streams



N = # randomly generated (x,y) samples in square

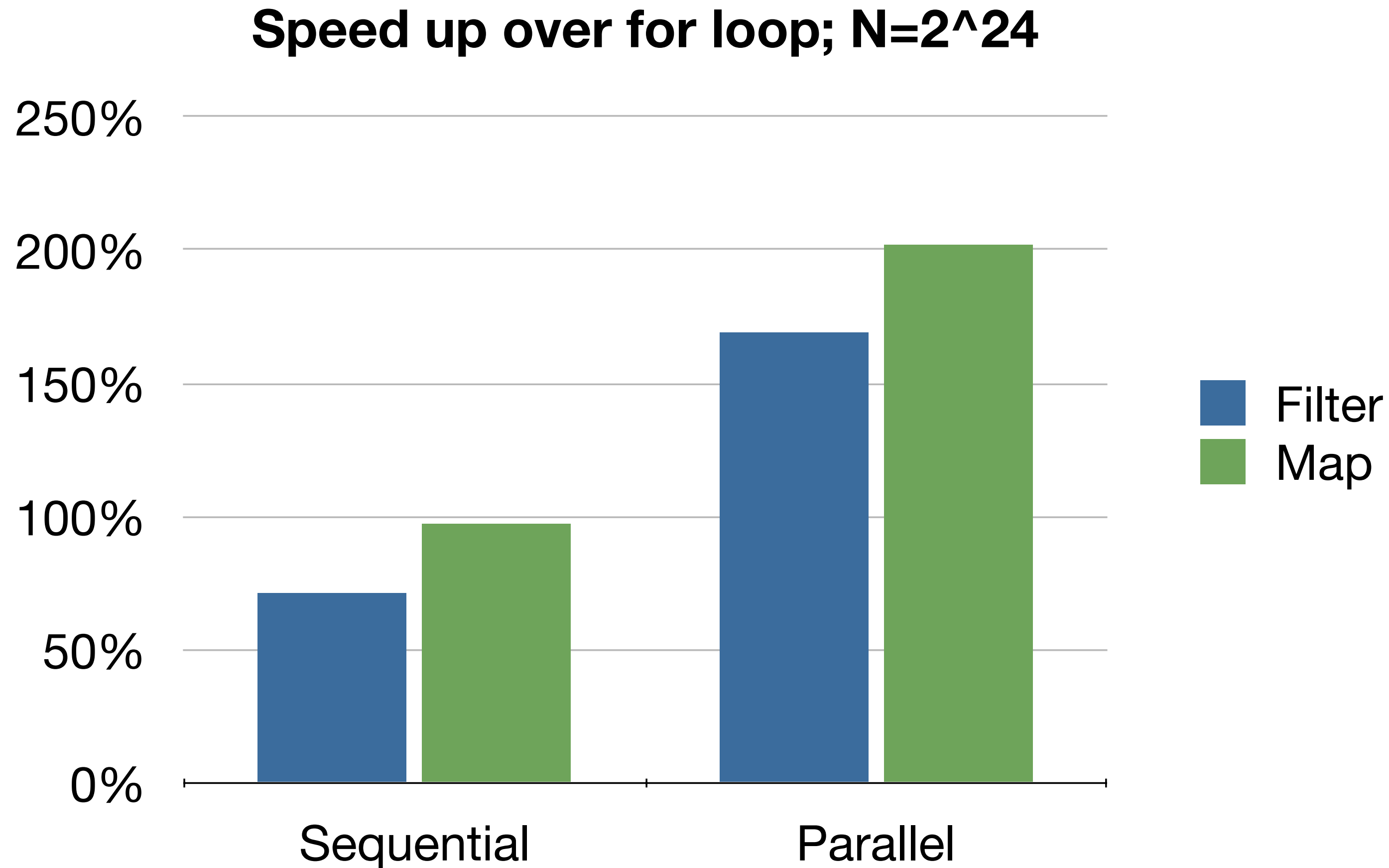
m = # (x,y) samples in circle

$$m/N \sim= \pi r^2/4r^2$$

$$4m/N \sim= \pi$$

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Monte Carlo calculation of π



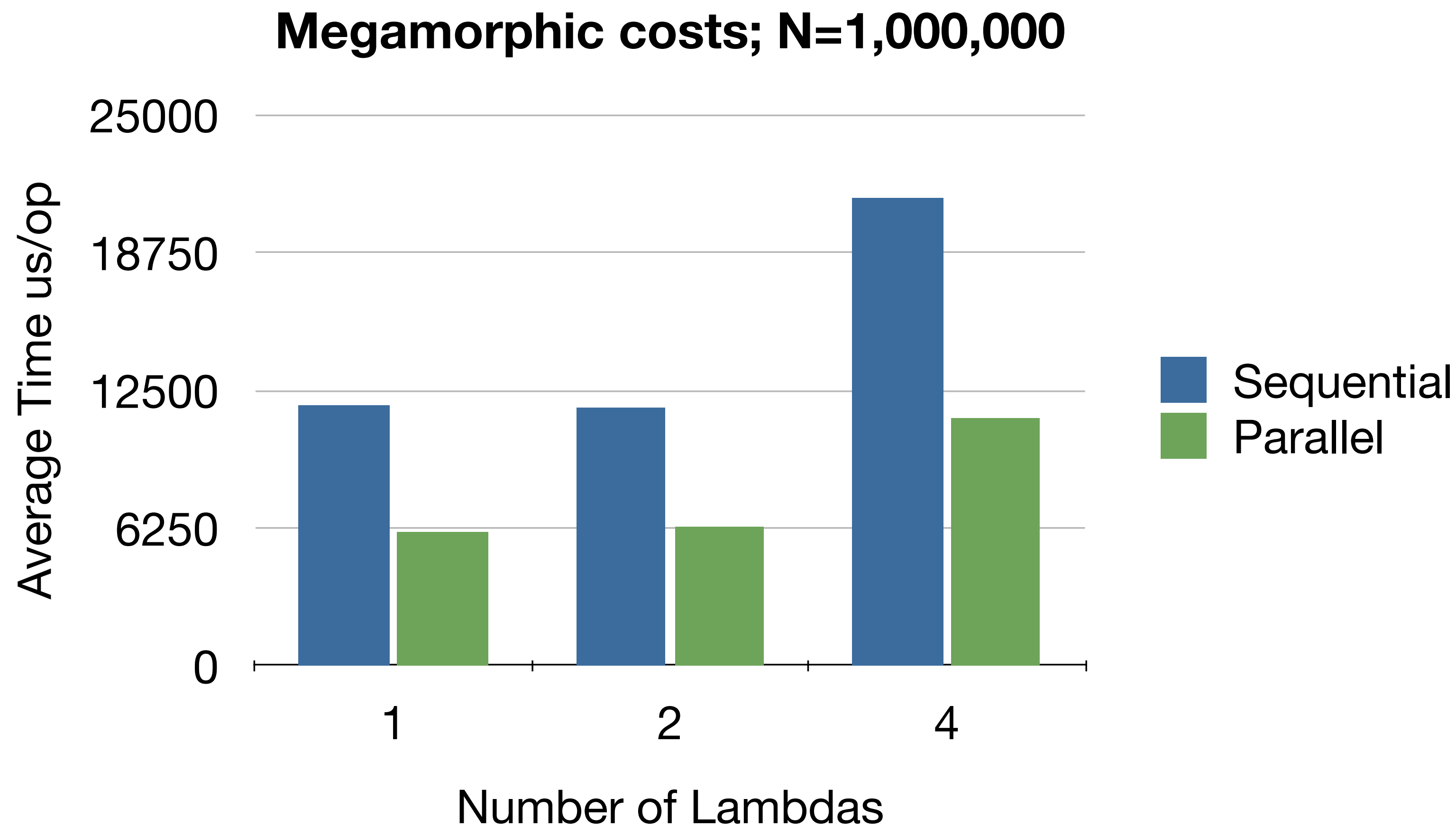
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Megamorphic call sites



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

Fixing the inlining problem

- Inlining is the "mother of all optimizations", but lambdas and streams are stressing the JIT out!
- Cliff Click in April 2011
  - http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem
  - "Inlining is not happening in a crucial point in hot code..."
  - "Inlining isn't happening because The Problem is a hard one to solve"
- Hard work is required to improve HotSpot and/or stream pipeline optimization

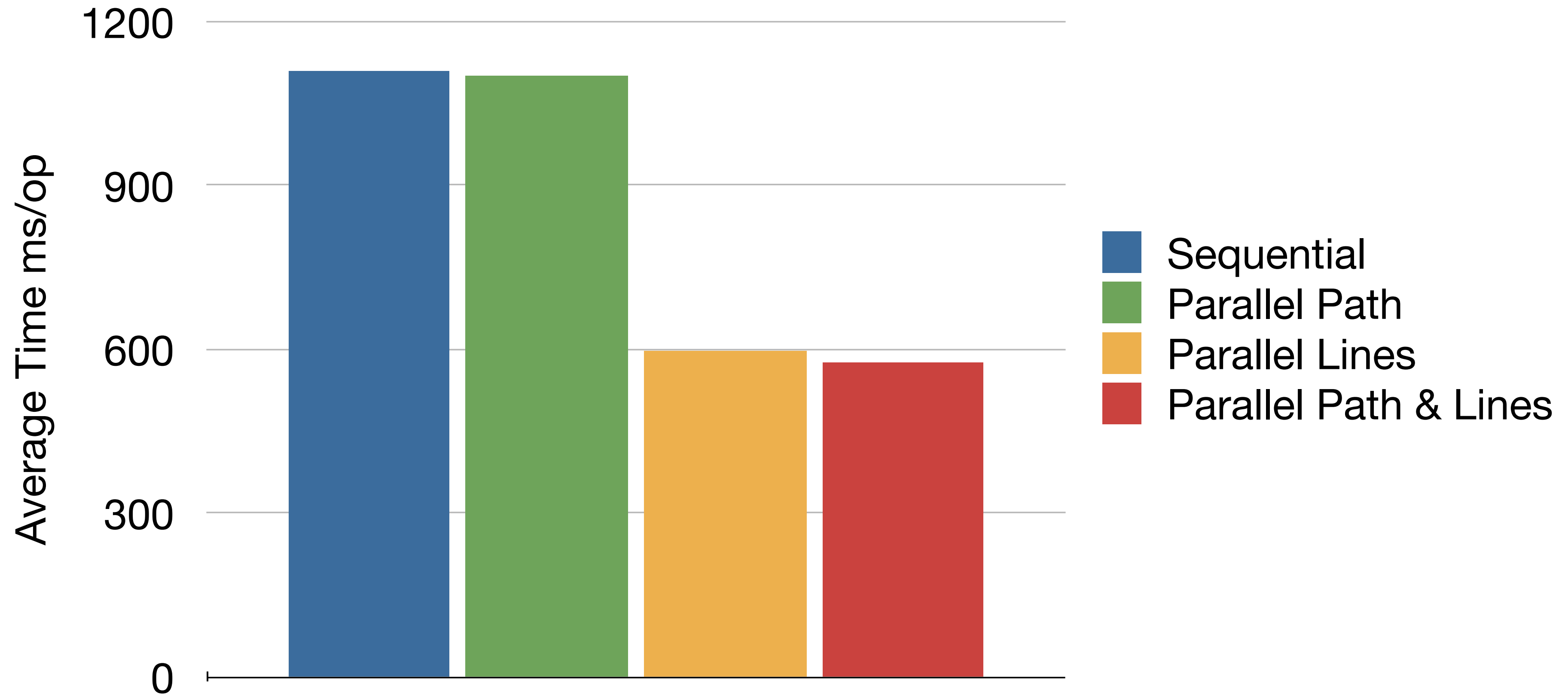
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

`jmh` measurement examples

- Low Q: Map and sum of `int[]` (`IntStream`)
- High Q: Generating probable primes
- Boxing: `int[]` (`IntStream`) vs. `Integer[]` (`Stream`)
- Monte Carlo calculation of π
- Megamorphic call sites
- Grep files

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Grep files



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

@PaulSandoz

Summary

- Streams is a powerful framework
  - More compact, readable, performant, less error-prone
- Easy path to parallelism
  - But parallelism is not always easy
- Deeply integrated with Collections
  - But not restricted to Collections
- Potential optimization improvements in the future

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Questions?

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.