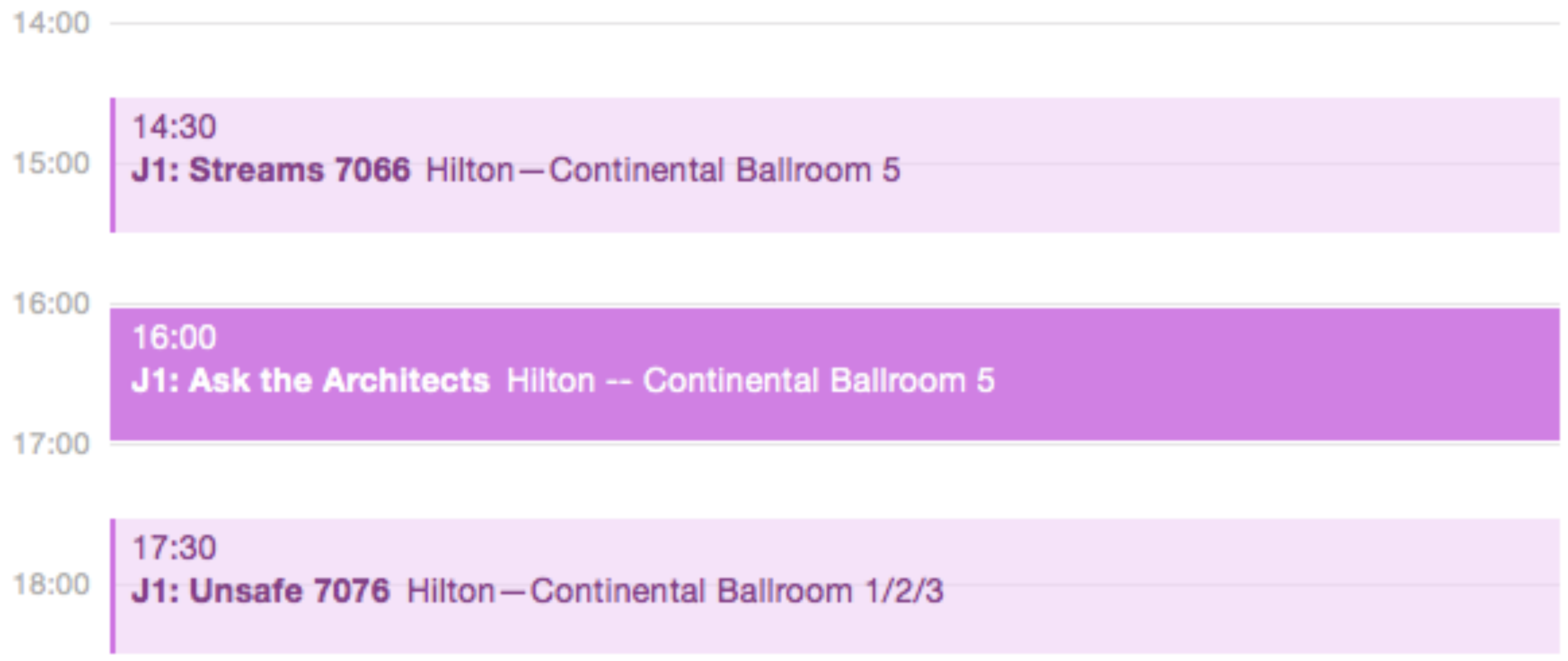
Effective Java Streams

Paul Sandoz
Oracle

@PaulSandoz



14:00

15:00

14:30
J1: Streams 7066 Hilton—Continental Ballroom 5

16:00

16:00
J1: Ask the Architects Hilton -- Continental Ballroom 5

17:00

18:00

17:30
J1: Unsafe 7076 Hilton—Continental Ballroom 1/2/3

```
list.stream().
map(λ).
filter(λ).
reduce(λ)
```

```
brian(λ).
john(λ).
mark(λ)
```

@PaulSandoz

Agenda

- ~~Patterns/Idioms~~ Tips and tricks with interesting stuff
- Effective parallel execution
- Enhancements in Java 9
- Beyond Java 9

Tips and tricks

- Counting
- Concatenating, **`flatMap`** and combining
- Operating over indices
- Composing

Effective parallel execution

- Need approximately 100 microseconds of sequential work across most platforms to break even

- http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html
  By Doug Lea

If it takes 1 nano second to add two integers, then how many integers are approximately needed to break even on parallel summation?

$$10^{-9} * N \sim= 10^{-4}$$

$$N \sim= 10^{5}$$

Effective parallel execution

- Choose good splitting sources with sufficient elements, and good intermediate and terminal operations

- Shooting the Rapids: Maximizing the Performance of Java 8 Streams [CON5931]

  Wednesday, Oct 28, 3:00 p.m. | Hilton—Continental Ballroom 4
  Maurice Naftalin & Kirk Pepperdine

Flat mapping enhancements in Java 9

- **`Optional.stream`** and **`Stream.ofNullable`** for better integration with **`flatMap`**
- **`Collectors.flatMapping`** for collecting zero or more items from a **`Stream`**

Stream returning enhancements in Java 9

- **`java.net.NetworkInterface`**

```
Enumeration<InetAddress> getInetAddresses()
Enumeration<NetworkInterface> getSubInterfaces()
static Enumeration<NetworkInterface> getNetworkInterfaces()
->
Stream<InetAddress> inetAddresses()
Stream<NetworkInterface> subInterfaces()
static Stream<NetworkInterface> networkInterfaces()
```

- **`java.security.PermissionCollection`**

```
Enumeration<Permission> elements()
->
Stream<Permission> elementsAsStream()
```

Larger enhancements in Java 9

- New operations **`{Int, Long, Double}`** **`Stream.takeWhile/dropWhile`**
- Parallel performance improvement of **`Files.lines`**
- Stream over results from **`java.util.regex.Matcher/Scanner`**

`Stream.take/dropWhile`

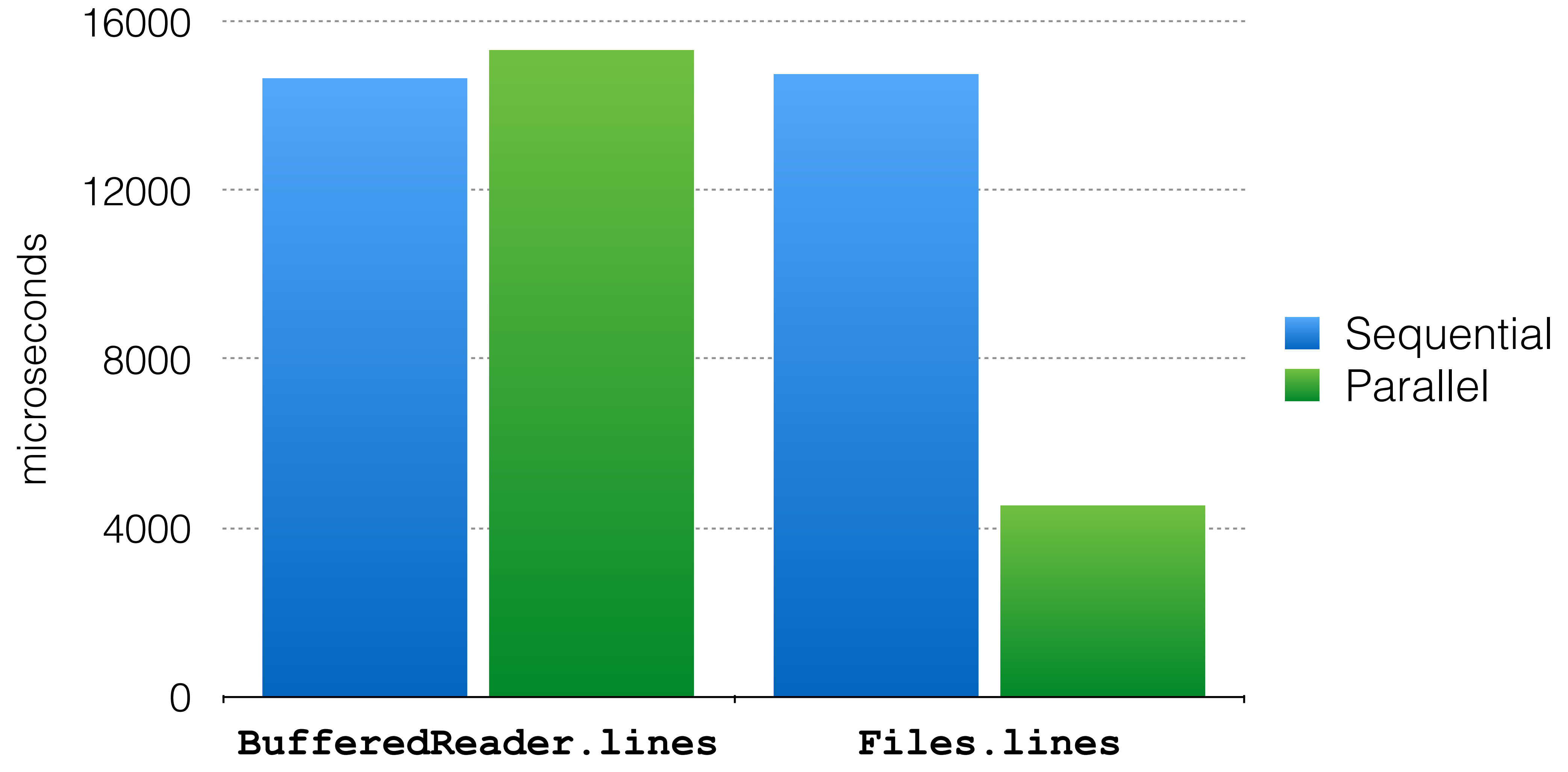
- Does the "obvious" thing for ordered streams
- Non-deterministic for unordered streams
  - Can take or drop any matching subset
- Parallel implementations are stateful and may perform as poorly as, or worse than, **`limit`**/**`skip`**

Parallel performance of **`Files.lines`**

- Memory maps the file for **`UTF-8`**, **`ISO 8859-1`** and **`US ASCII`**
  - Character sets where line feeds are easily identifiable via random access of file contents
- Efficient splitting of the mapped memory region
  - Divides ~ in half to the closest line feed from the mid-point

Performance

Processing a file of 100,000 lines
each of 80 characters



Results produced using **`jmh`** on a MacBook Pro (2012 model)

Beyond Java 9

- Improve parallel production of lists and maps
  - **`s.collect(toList())`**
- Leverage value types and generics over values
  - Simpler more powerful API and implementation
  - Easier to introduce extensions such as map-based streams or an SPI for pluggable operations

Expression with performance

- Want to express **`IntStream <: Stream<int>`**
- Without explicit specialisation of the implementation (as is the case today)
- With stream sources that pack and align in memory for better cache coherency
- With stream pipelines that inline the main processing loop (“loop specialization")

Latency numbers

https://gist.github.com/jboner/2841832

L1 cache reference	0.5	ns		
Branch mispredict	5	ns		
L2 cache reference	7	ns		
Mutex lock/unlock	25	ns		
Main memory reference	100	ns		
Compress 1K bytes with Zippy	3,000	ns		
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms
Read 4K randomly from SSD*	150,000	ns	0.15	ms
Read 1 MB sequentially from memory	250,000	ns	0.25	ms
Round trip within same datacenter	500,000	ns	0.5	ms
Read 1 MB sequentially from SSD*	1,000,000	ns	1	ms
Disk seek	10,000,000	ns	10	ms
Read 1 MB sequentially from disk	20,000,000	ns	20	ms
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms

Boxes, alignment and GC

```java
// Create an array of Boxed integer
Integer[] arr = new Integer[10];
for (int i = 0; i < 10; i++) {
    arr[i] = new Integer(i);
}
```

```
 ADDRESS       SIZE TYPE              PATH                           VALUE
740012698        16 java.lang.Integer <r4>                            3
7400126a8       424 (something else)  (somewhere else)                (something else)
740012850        16 java.lang.Integer <r6>                            5
740012860        16 java.lang.Integer <r8>                            7
740012870        48 (something else)  (somewhere else)                (something else)
7400128a0        16 java.lang.Integer <r10>                           9
7400128b0    382920 (something else)  (somewhere else)                (something else)
740070078        16 java.lang.Integer <r2>                            1
740070088        16 java.lang.Integer <r3>                            2
740070098     16456 (something else)  (somewhere else)                (something else)
7400740e0        16 java.lang.Integer <r9>                            8
7400740f0        16 java.lang.Integer <r7>                            6
740074100        16 java.lang.Integer <r5>                            4
740074110    169808 (something else)  (somewhere else)                (something else)
74009d860        16 java.lang.Integer <r1>                            0
```

`Stream<any T>`

- Prototype in valhall repo
  http://openjdk.java.net/projects/valhalla/
  http://hg.openjdk.java.net/valhalla
- Temporary home in package
  `java.anyutil.stream`
- Code significantly reduced

Why “don’t you just” add a method to zip two streams in Java 8 or 9?

```
<A, B, C> Stream<C> zip(Stream<A> a,
                        Stream<B> b,
                        BiFunction<A, B, C> zipper)

<A, C> Stream<C> zip(Stream<A> a,
                     IntStream b,
                     BiFunction<A, Integer, C> zipper)

<A, C> Stream<C> zip(Stream<A> a,
                     LongStream b,
                     BiFunction<A, Long, C> zipper)

<A, C> Stream<C> zip(Stream<A> a,
                     DoubleStream b,
                     BiFunction<A, Double, C> zipper)
```

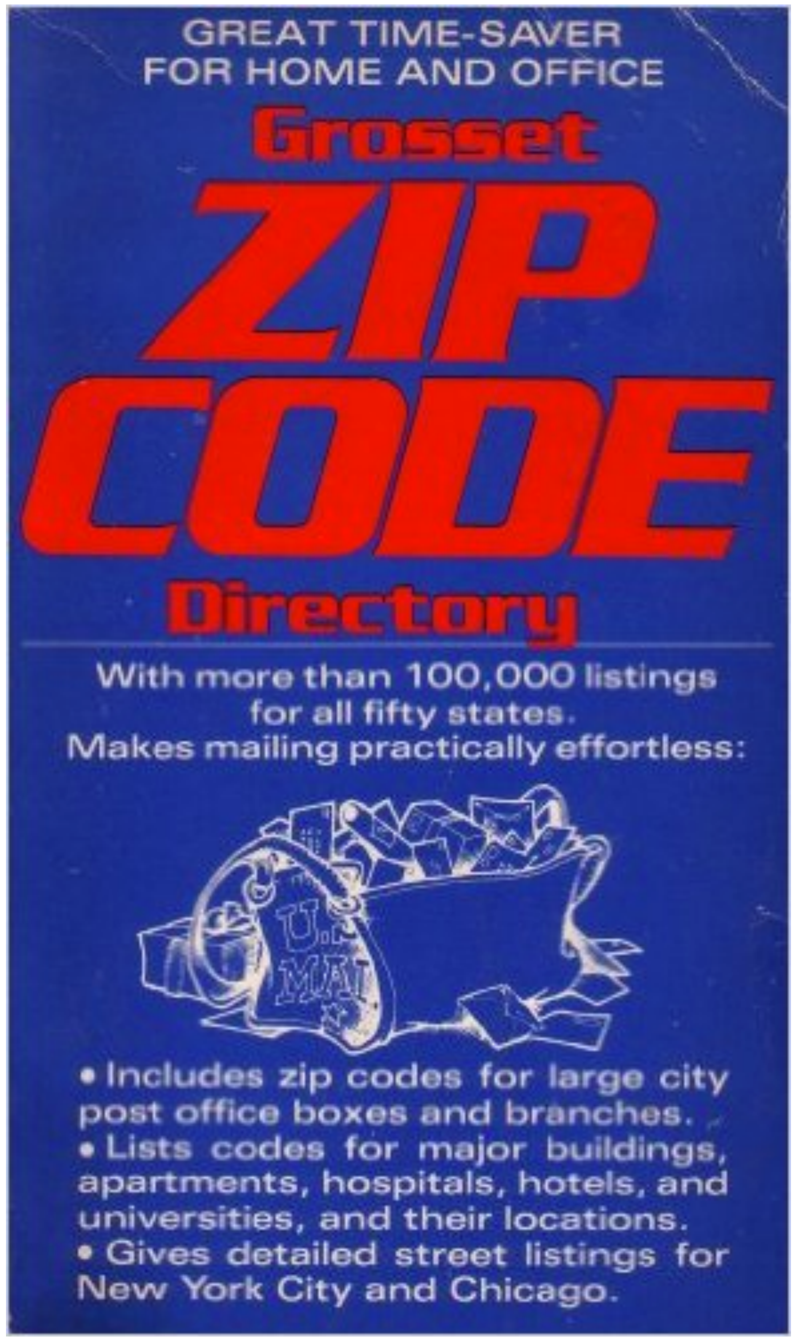
```
IntStream zip(IntStream a,
              IntStream b,
              BiFunction<Integer, Integer, Integer> zipper)

IntStream zip(IntStream a,
              LongStream b,
              BiFunction<Integer, Long, Integer> zipper)

IntStream zip(IntStream a,
              DoubleStream b,
              BiFunction<Integer, Double, Integer> zipper)
```

```
IntStream zip(LongStream a,
              LongStream b,
              BiFunction<Long, Long, Integer> zipper)
```

```
IntStream zip(LongStream a,
              DoubleStream b,
              BiFunction<Long, Double, Integer> zipper)
```


GREAT TIME-SAVER
FOR HOME AND OFFICE

Grosset
ZIP
CODE
Directory

With more than 100,000 listings
for all fifty states.
Makes mailing practically effortless:

- Includes zip codes for large city post office boxes and branches.
- Lists codes for major buildings, apartments, hospitals, hotels, and universities, and their locations.
- Gives detailed street listings for New York City and Chicago.

Grosset Zip Code Directory: U.S. Postal Zip Code Directory by Grosset Dunlap, Ottenheimer Publishers, Filmways Company. Paperback 1977 Printing by Grosset Dunlap. 490 Pages. ASIN B000J0GSK2. MPN GD14732. In English. Special Limited Edition.

Zipping streams

- **`{Int,Long,Double}Stream.zip`** was not added in Java 8/9
  - Method and functional interface explosion
- Easy to support in Valhalla with fewer methods and functional interfaces
  - Support for tuples would be nice too but…

In legal safety

?

Hackergarten, Java Hub
Track #2
10am-12pm Wed