
Let's Get to the Rapids
Understanding Java 8 Stream Performance
Philly ETE
April 2015
@mauricenaftalin

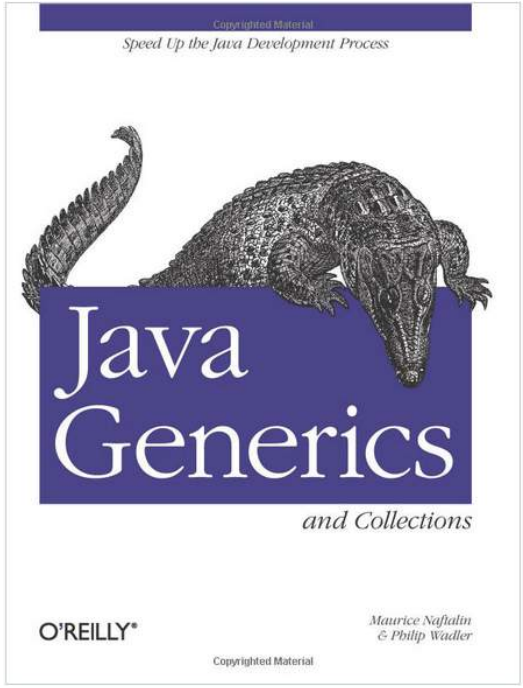
Maurice Naftalin

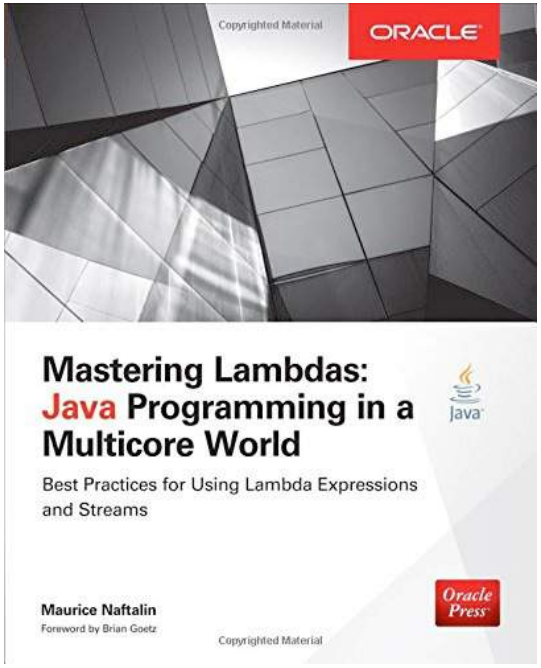
Developer, designer, architect, teacher, learner, writer

@mauricenaftalin

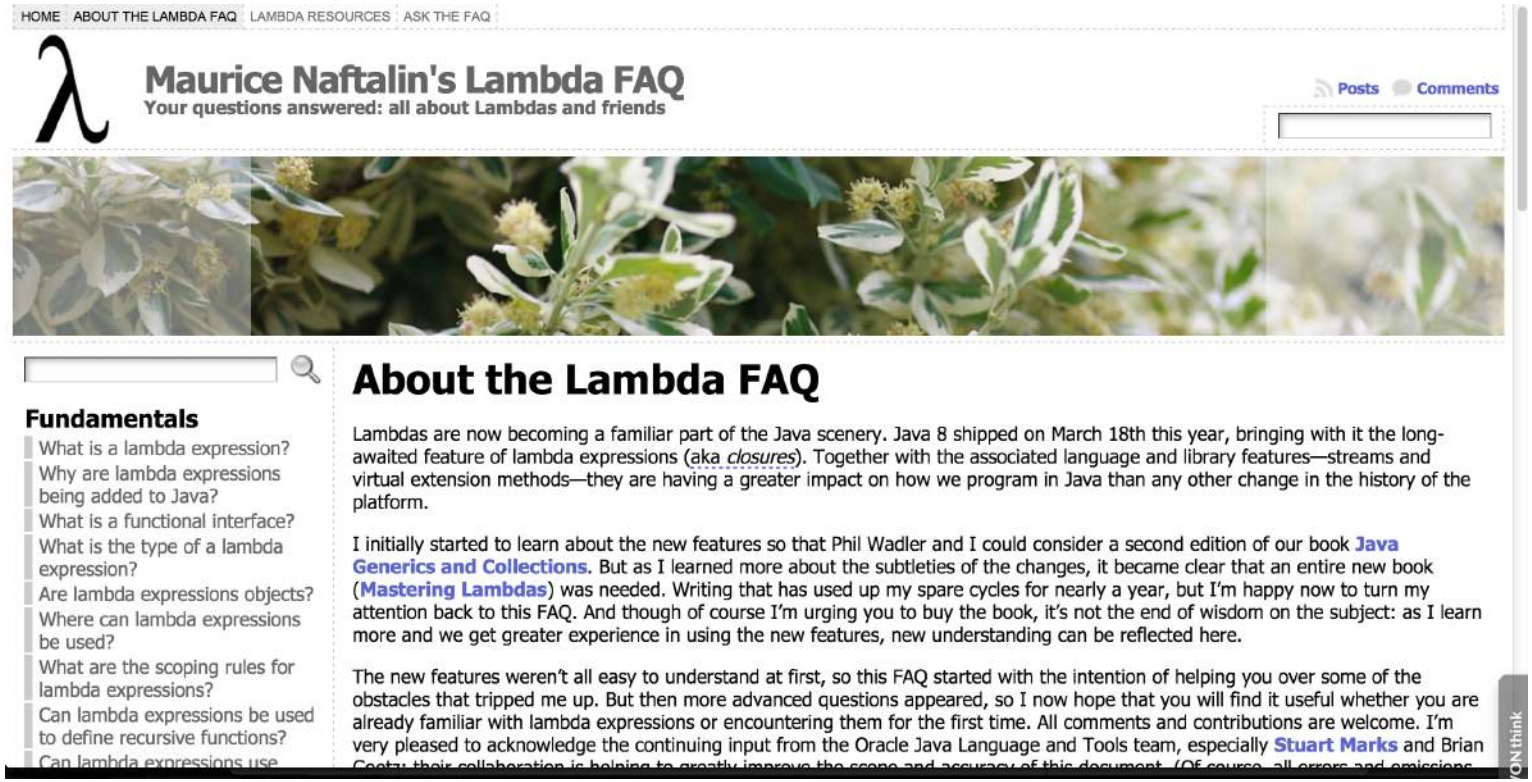
Maurice Naftalin

Repeat offender: Java 5 Java 8





The Lambda FAQ

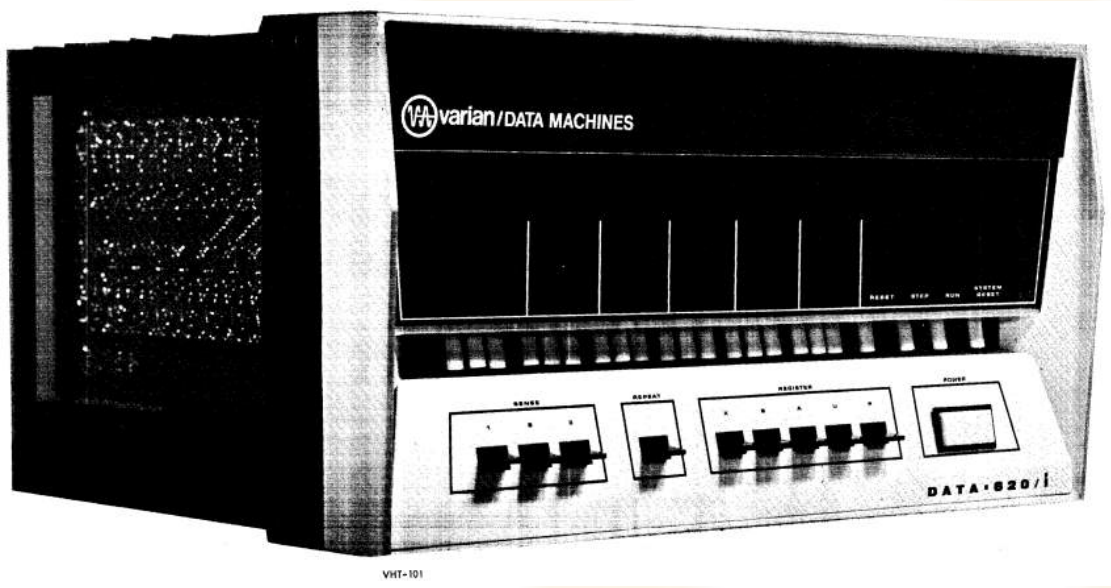


www.lambdafaq.org

@mauricenaftalin

First Computer I Used

Varian 620/i



Fast operation:	1.8-microsecond memory cycle.
Large instruction repertoire:	107 standard, 18 optional; with approximately 200 additional instruction configurations which can be microcoded.
Word length:	16- or 18-bit configurations.
Modular memory:	4096 word minimum, 32,768 words maximum.

Agenda

- Background
    - Java 8 Streams
    - Parallelism
    - Microbenchmarking
- Case study
- Conclusions

Streams – Why?

- Bring functional style to Java
- Exploit hardware parallelism – “explicit but unobtrusive”

Streams – Why?

- Intention: replace loops for aggregate operations

instead of writing this:

```
List<Person> people = …
Set<City> shortCities = new HashSet<>();

for (Person p : people) {
   City c = p.getCity();
   if (c.getName().length() < 4 ) {
       shortCities.add(c);
   }
}
```

Streams – Why?

- Intention: replace loops for aggregate operations

instead of writing this:

```
Set<City> shortCities = new HashSet<>();

for (Person p : pe
   City c = p.getC
   if (c.getName()
      shortCities
   }
}
```

we're going to write this:

```
List<Person> people = …
Set<City> shortCities = people.stream()
    .map(Person::getCity)
    .filter(c -> c.getName().length() < 4)
    .collect(toSet());
```

Streams – Why?

- Intention: replace loops for aggregate operations
  - more concise, more readable, composable operations, parallelizable

instead of writing this:

```
Set<City> shortCities = new HashSet<>();

for (Person p : pe
   City c = p.getC
   if (c.getName()
       shortCities.add(c);
   }
}
```

we're going to write this:

```
Set<City> shortCities = people.parallelStream()
    .map(Person::getCity)
    .filter(c -> c.getName().length() < 4)
    .collect(toSet());
```

Practical Benefits of Streams?

Functional style will affect (nearly) ***all collection processing***

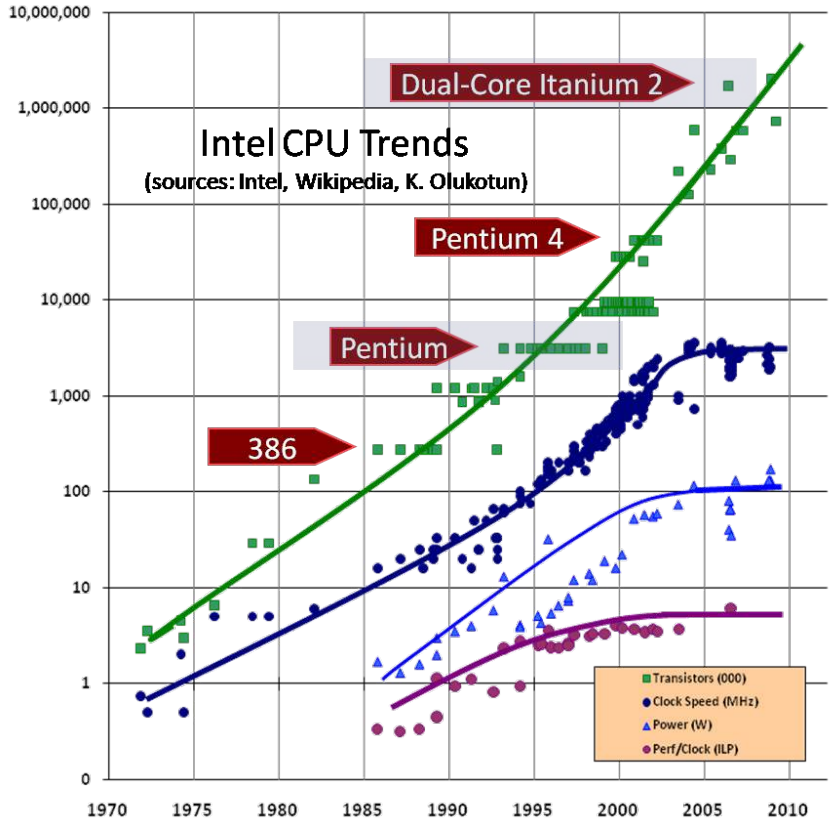
Automatic parallelism is useful, in ***certain situations***

- but everyone cares about performance!

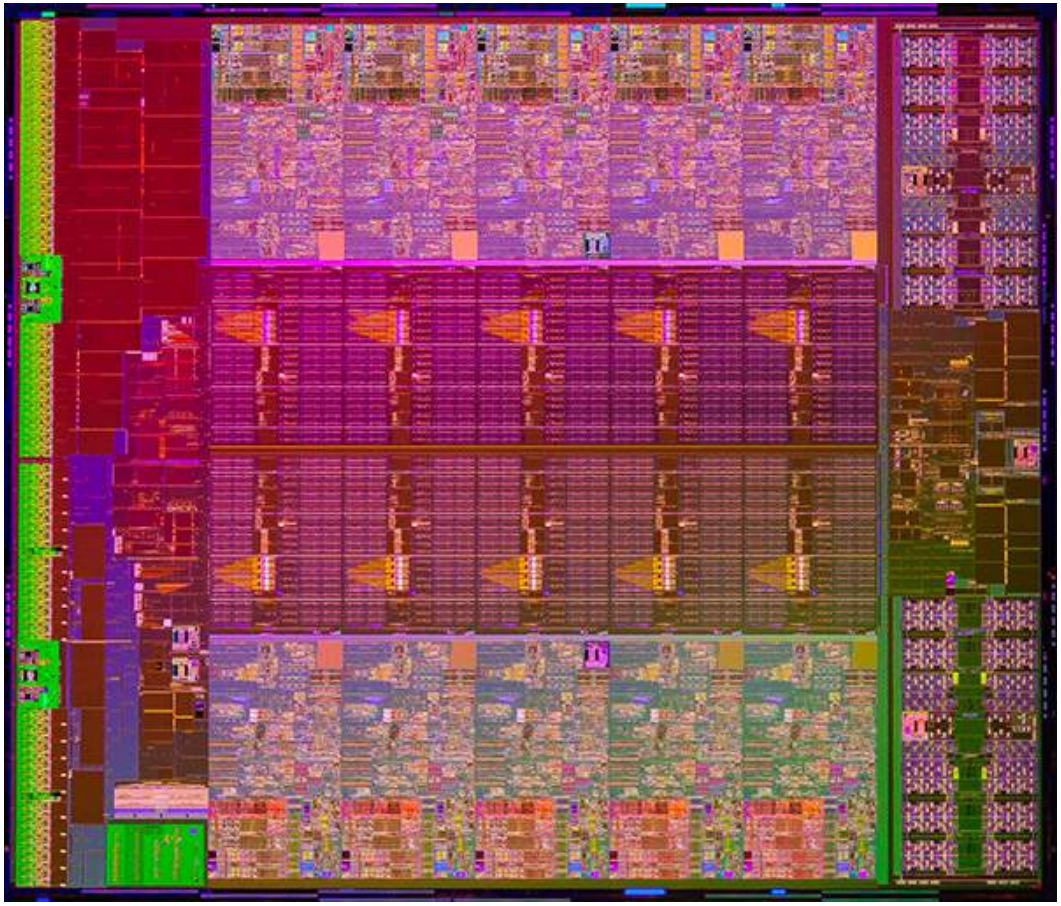
Parallelism – Why?

The Free Lunch Is Over

http://www.gotw.ca/publications/concurrency-ddj.htm



Intel Xeon E5 2600 10-core



Microbenchmarking

Really hard to get meaningful results from a dynamic runtime:

- timing methods are flawed
  - System.currentTimeMillis() and System.nanoTime()
- compilation can occur at any time
- garbage collection interferes
- runtime optimizes code after profiling it for some time
- then may deoptimize it
  - optimizations include dead code elimination

Microbenchmarking

Don’t try to eliminate these effects yourself!

Use a benchmarking library

- Caliper
- JMH (Java Benchmarking Harness)

Ensure your results are statistically meaningful

Get your benchmarks peer-reviewed

Case Study: `grep -b`

`grep -b:` "The offset in bytes of a matched pattern is displayed in front of the matched line."

The Moving Finger writes; and, having writ,
Moves on: nor all thy Piety nor Wit
Shall bring it back to cancel half a Line
Nor all thy Tears wash out a Word of it.

rubai51.txt

```
$ grep -b 'W.*t' rubai51.txt
44:Moves on: nor all thy Piety nor Wit
122:Nor all thy Tears wash out a Word of it.
```

Why Shouldn't We Care?

Because we don't have a problem

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

- The OS is struggling!

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

- The OS is struggling!

Else there's a problem in our process, but not in the code

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

- The OS is struggling!

Else there's a problem in our process, but not in the code

- GC is using all the cycles!

Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

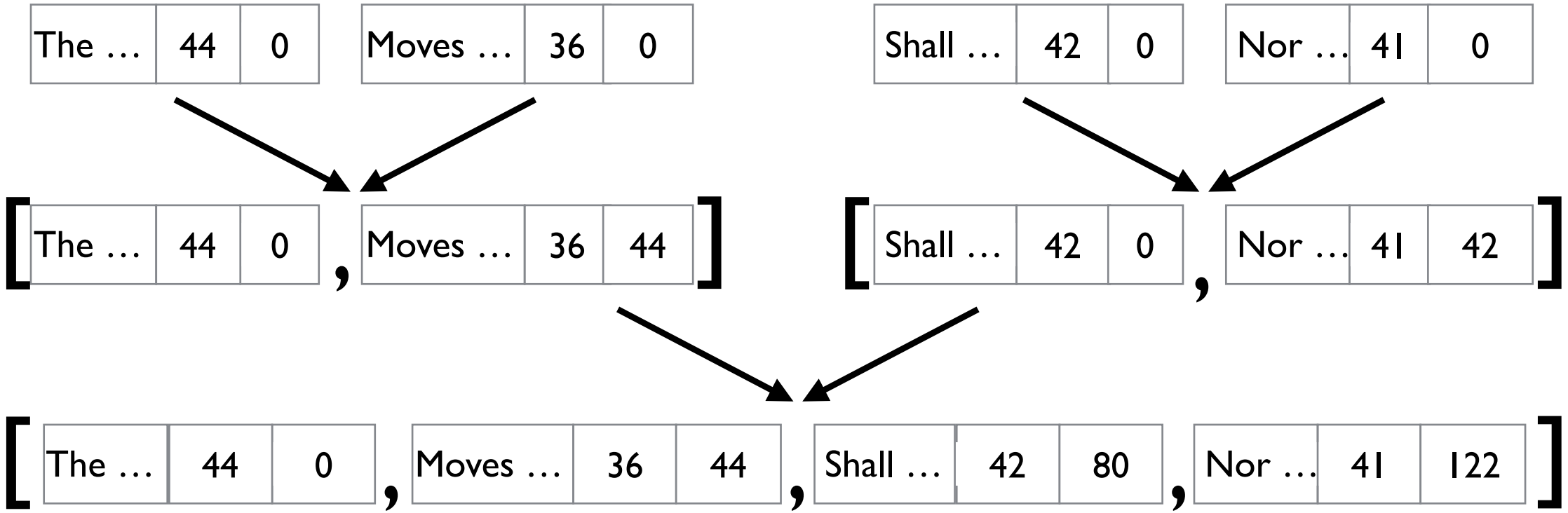
- The OS is struggling!

Else there's a problem in our process, but not in the code

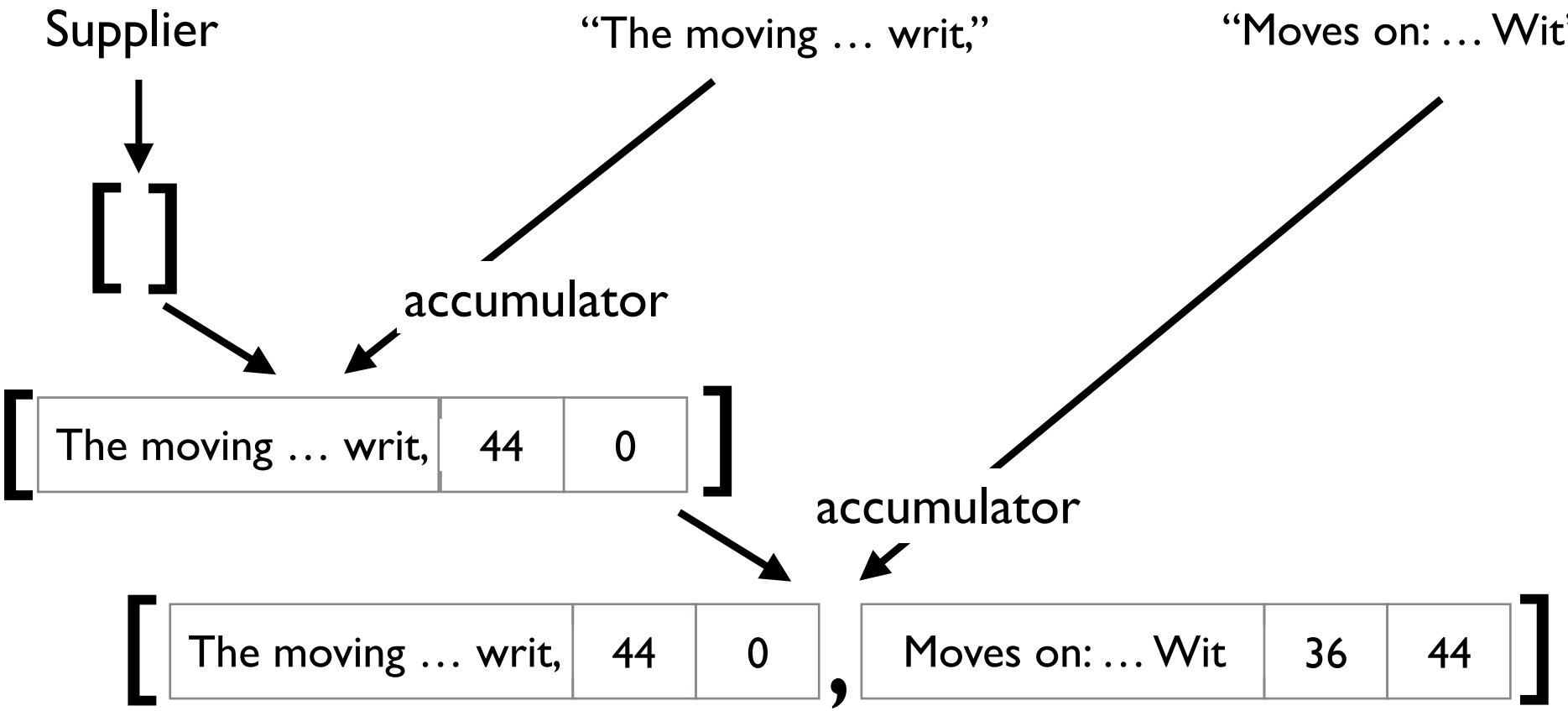
- GC is using all the cycles!

Else there's a problem in the code, but not a stream problem

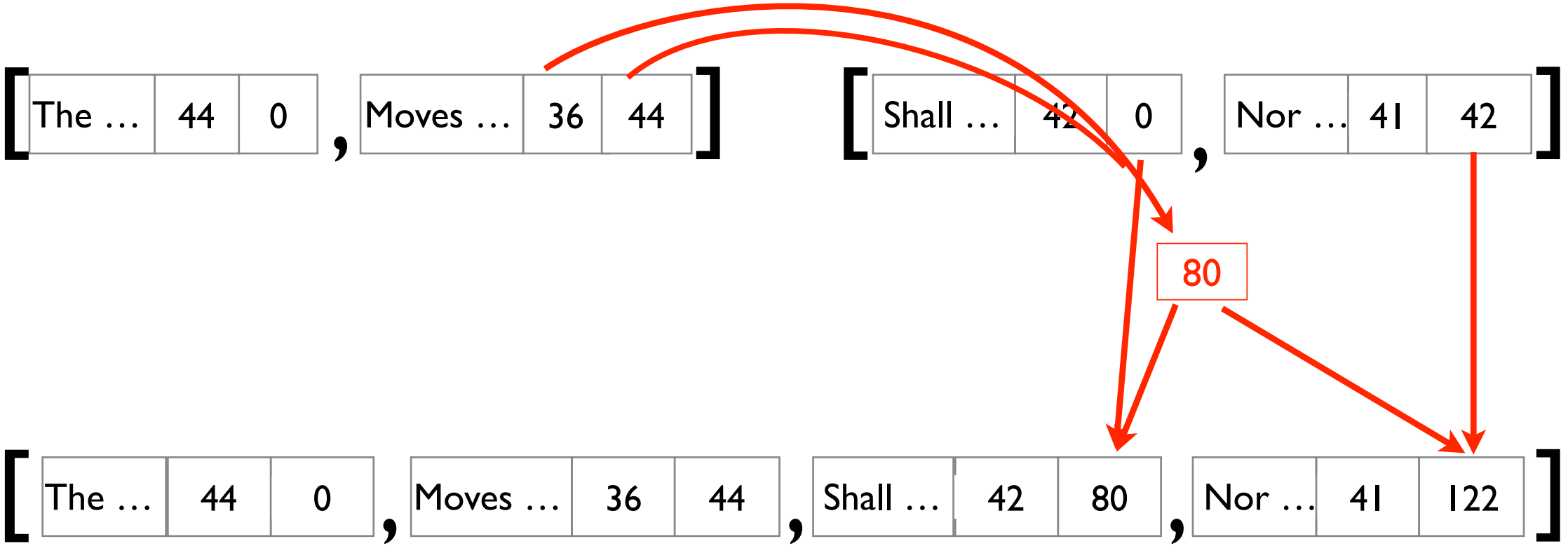
grep -b: Collector solution



grep -b: Collector solution



grep -b: Collector solution



Why Shouldn't We Care?

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

- The OS is struggling!

Else there's a problem in our process, but not in the code

- GC is using all the cycles!

Else there's a problem in the code, but not a stream problem

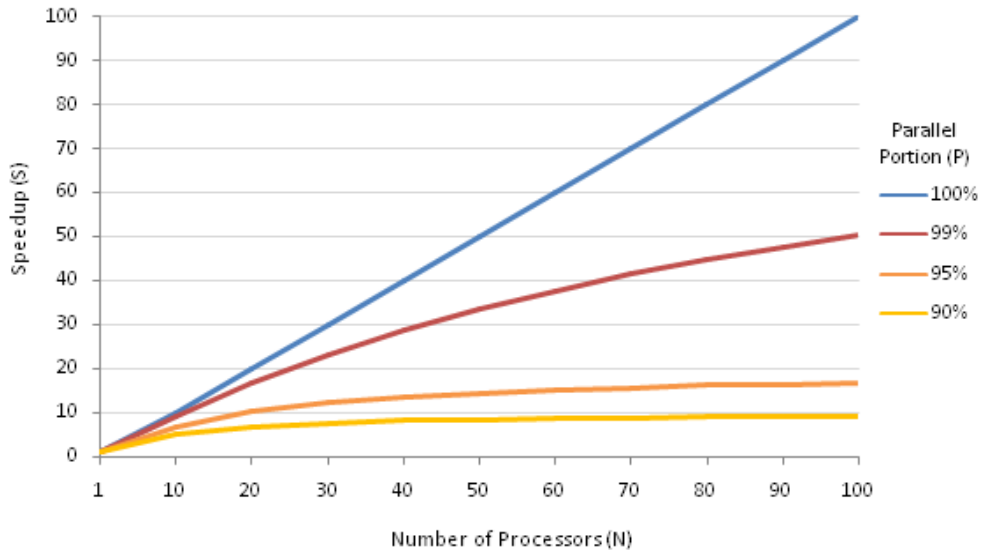
- The bottleneck is elsewhere!

What's wrong?

- Possibly very little
  - overall performance comparable to Unix `grep -b`
- Can we improve it by going parallel?

Serial vs. Parallel

- The problem is a *prefix sum* – every element is the sum of the preceding ones.
  - Combiner is O(n)
- The source is streaming IO (`BufferedReader.lines()`)
- Amdahl's Law strikes:



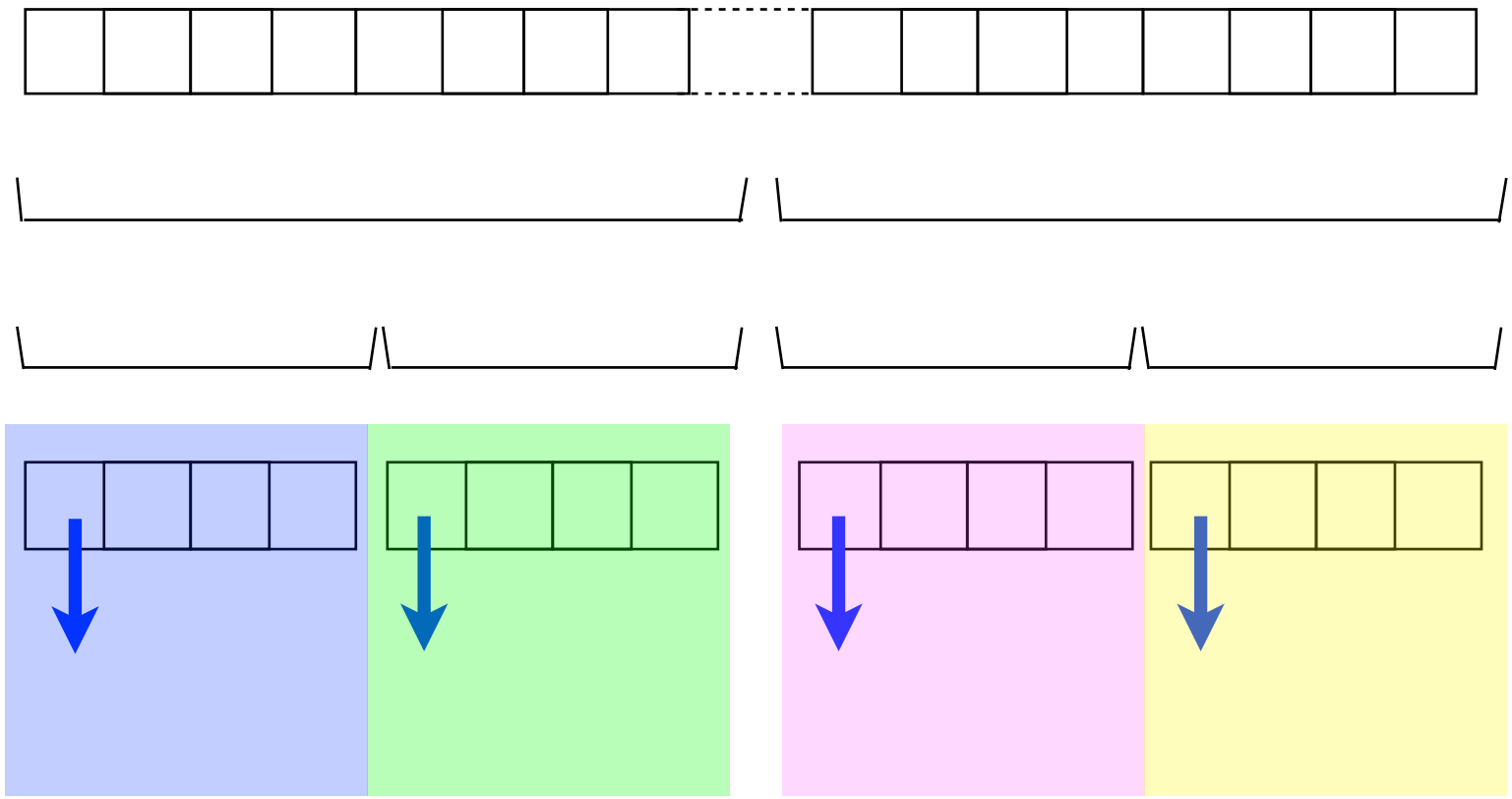
A Parallel Solution for grep -b

Need to get rid of streaming IO – inherently serial

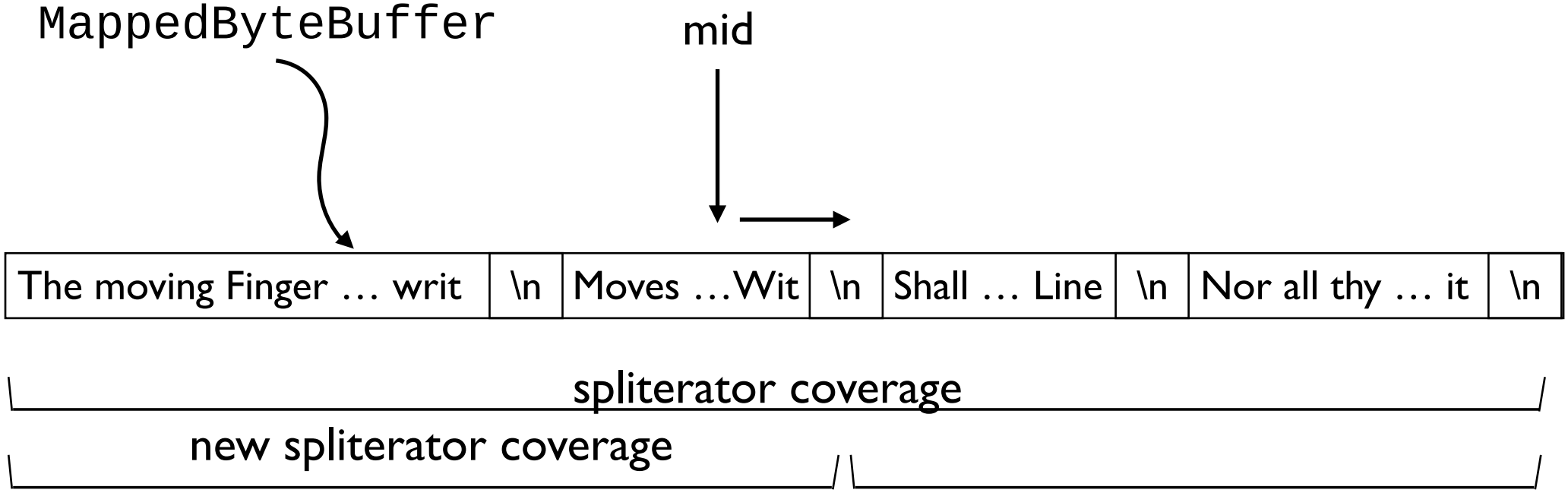
Parallel streams need *splittable* sources

Stream Sources

Implemented by a `Spliterator`



LineSpliterator



Parallelizing `grep -b`

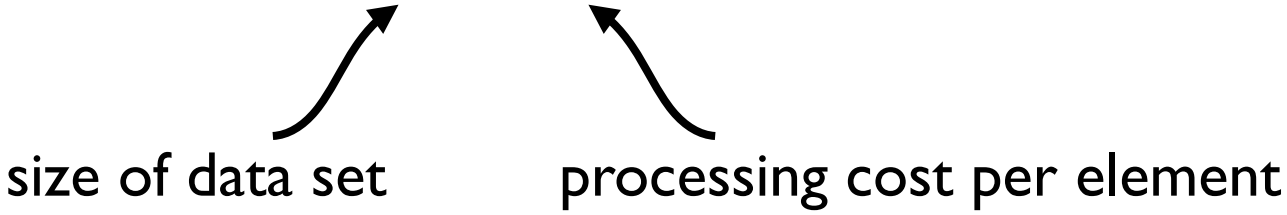
- Splitting action of `LineSpliterator` is O(log n)
- Collector no longer needs to compute index
- Result (relatively independent of data size):
  - sequential stream ~2x as fast as iterative solution
  - parallel stream >2.5x as fast as sequential stream
    - on 4 hardware threads

When to go Parallel

The workload of the intermediate operations must be great enough to outweigh the overheads (~100µs):

- splitting
- concurrent collection
- initializing the fork/join framework

Often quoted as N x Q



Intermediate Operations

Parallel-unfriendly intermediate operations:

stateful ones

- need to store some or all of the stream data in memory
  - `sorted()`

those requiring ordering

- `limit()`

Collectors Cost Extra!

Depends on the performance of accumulator and combiner functions

- toList(), toSet(), toCollection – performance normally dominated by accumulator
  - but allow for the overhead of managing multithread access to non-threadsafe containers for the combine operation
- toMap(), toConcurrentMap() – *map merging is slow*. Resizing maps, especially concurrent maps, is particularly expensive. Whenever possible, presize all data structures, maps in particular.

Parallel Streams in the Real World

Threads for executing parallel streams are (all but one) drawn from the common Fork/Join pool

- Intermediate operations that block (for example on I/O) will prevent pool threads from servicing other requests
- Fork/Join pool assumes by default that it can use all cores
  - Maybe other thread pools (or other processes) are running?

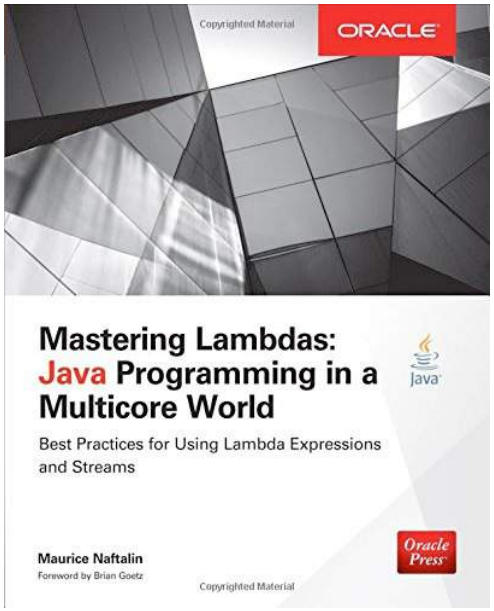
Conclusions

Performance ***mostly doesn't matter***

But if you ***must***…

- sequential streams normally beat iterative solutions
- parallel streams can utilize all cores, providing
  - the data is efficiently splittable
  - the intermediate operations are sufficiently expensive and are CPU-bound
  - there isn't contention for the processors

Resources



http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf

http://openjdk.java.net/projects/code-tools/jmh/

@mauricenaftalin