



# Let's Get to the Rapids

## Understanding Java 8 Stream Performance

QCon New York

June 2015

@mauricenaftalin

# Maurice Naftalin

---

Developer, designer, architect, teacher, learner, writer

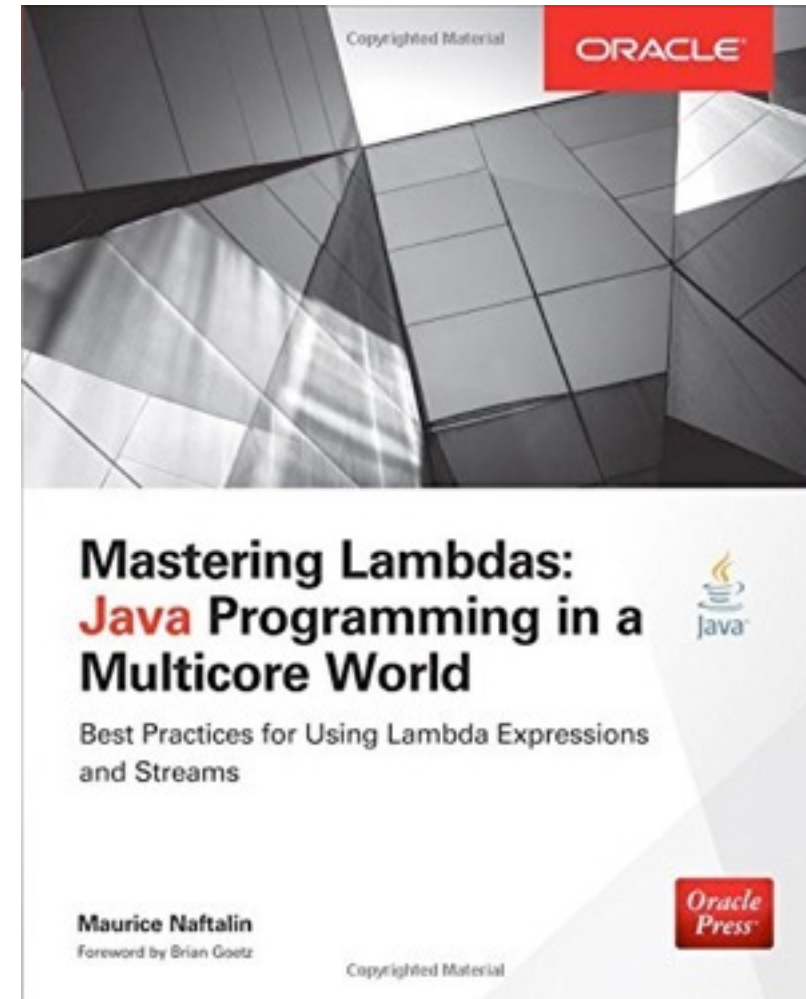
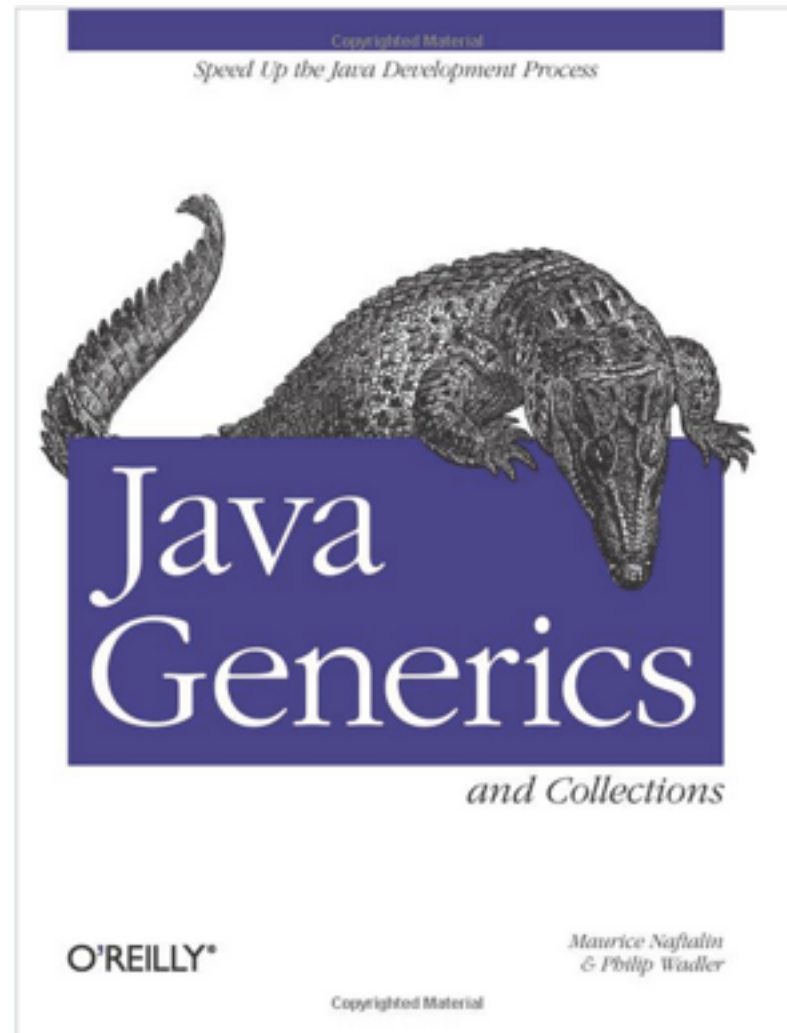


# Maurice Naftalin

Repeat offender:


Java 5

Java 8




# The Lambda FAQ

HOME ABOUT THE LAMBDA FAQ LAMBDA RESOURCES ASK THE FAQ

 **Maurice Naftalin's Lambda FAQ**  
Your questions answered: all about Lambdas and friends

Posts Comments




## About the Lambda FAQ

Lambdas are now becoming a familiar part of the Java scenery. Java 8 shipped on March 18th this year, bringing with it the long-awaited feature of lambda expressions (aka *closures*). Together with the associated language and library features—streams and virtual extension methods—they are having a greater impact on how we program in Java than any other change in the history of the platform.

I initially started to learn about the new features so that Phil Wadler and I could consider a second edition of our book [Java Generics and Collections](#). But as I learned more about the subtleties of the changes, it became clear that an entire new book ([Mastering Lambdas](#)) was needed. Writing that has used up my spare cycles for nearly a year, but I'm happy now to turn my attention back to this FAQ. And though of course I'm urging you to buy the book, it's not the end of wisdom on the subject: as I learn more and we get greater experience in using the new features, new understanding can be reflected here.

The new features weren't all easy to understand at first, so this FAQ started with the intention of helping you over some of the obstacles that tripped me up. But then more advanced questions appeared, so I now hope that you will find it useful whether you are already familiar with lambda expressions or encountering them for the first time. All comments and contributions are welcome. I'm very pleased to acknowledge the continuing input from the Oracle Java Language and Tools team, especially [Stuart Marks](#) and [Brian Goetz](#); their collaboration is helping to greatly improve the scope and accuracy of this document. (Of course, all errors and omissions



### Fundamentals

- What is a lambda expression?
- Why are lambda expressions being added to Java?
- What is a functional interface?
- What is the type of a lambda expression?
- Are lambda expressions objects?
- Where can lambda expressions be used?
- What are the scoping rules for lambda expressions?
- Can lambda expressions be used to define recursive functions?
- Can lambda expressions use

IONthink

[www.lambdafaq.org](http://www.lambdafaq.org)

@mauricenaftalin

# 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:

we're going to write this:

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

```
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:

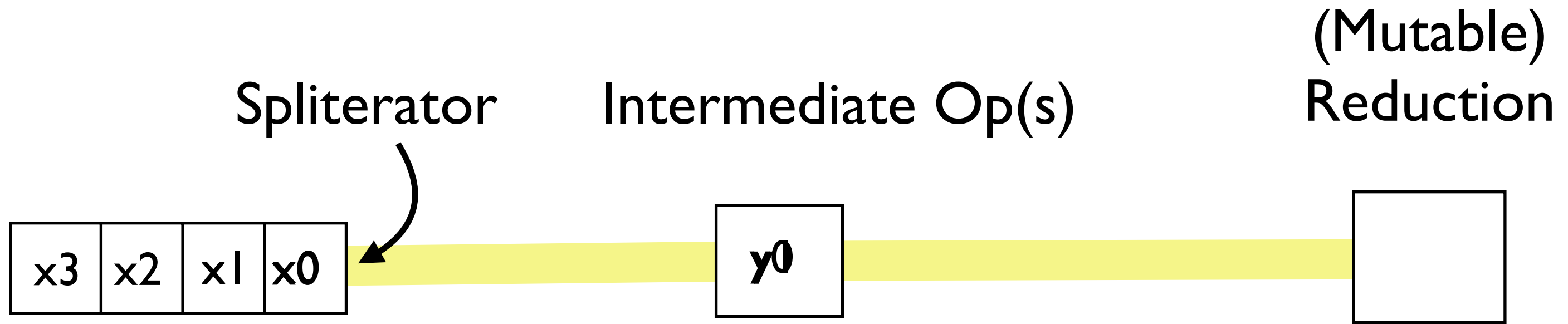
we're going to write this:

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

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

# Visualizing Stream Operations

---



# Practical Benefits of Streams?

---

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

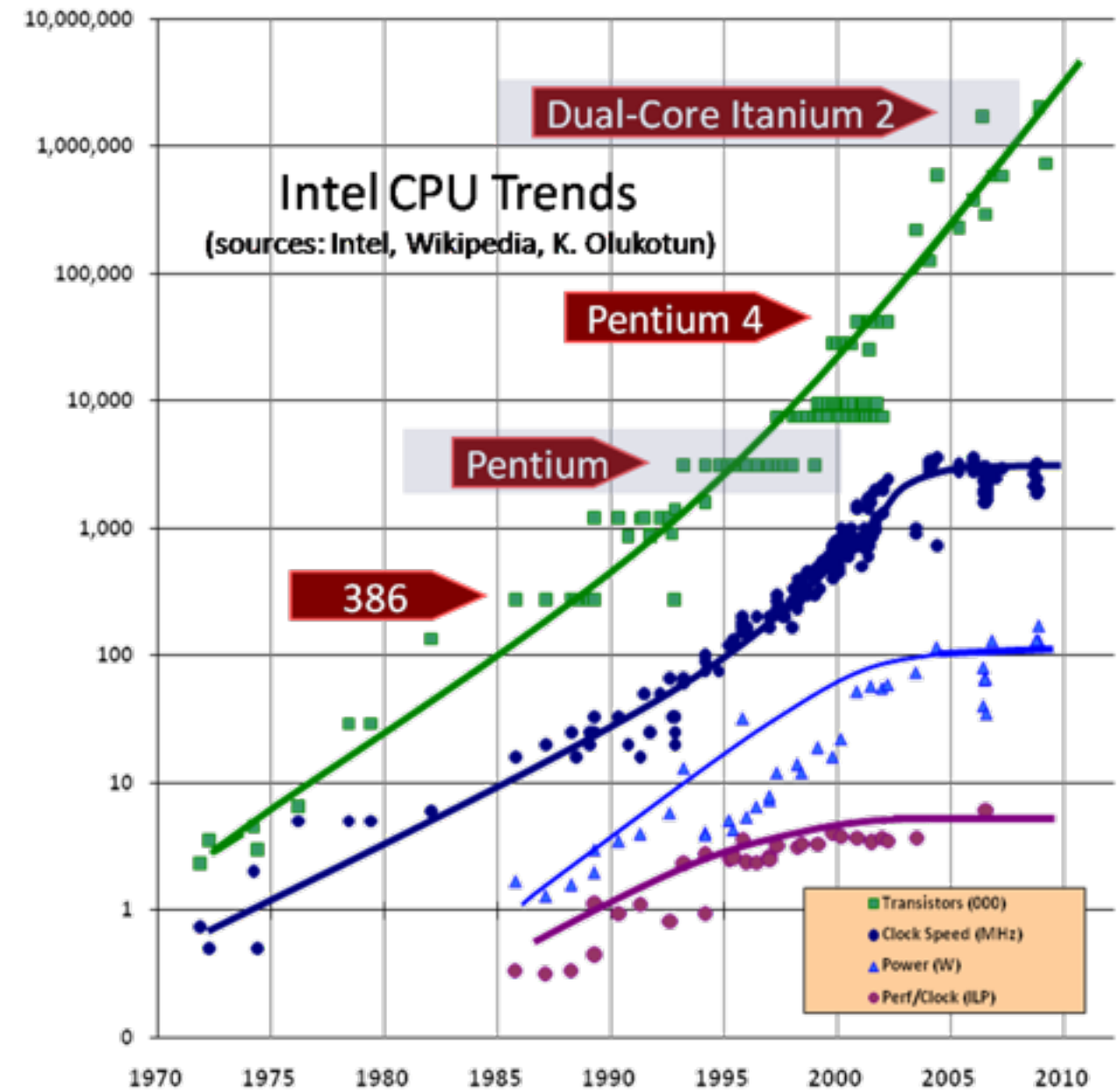
Automatic parallelism is useful, in *certain situations*

- but everyone cares about performance!

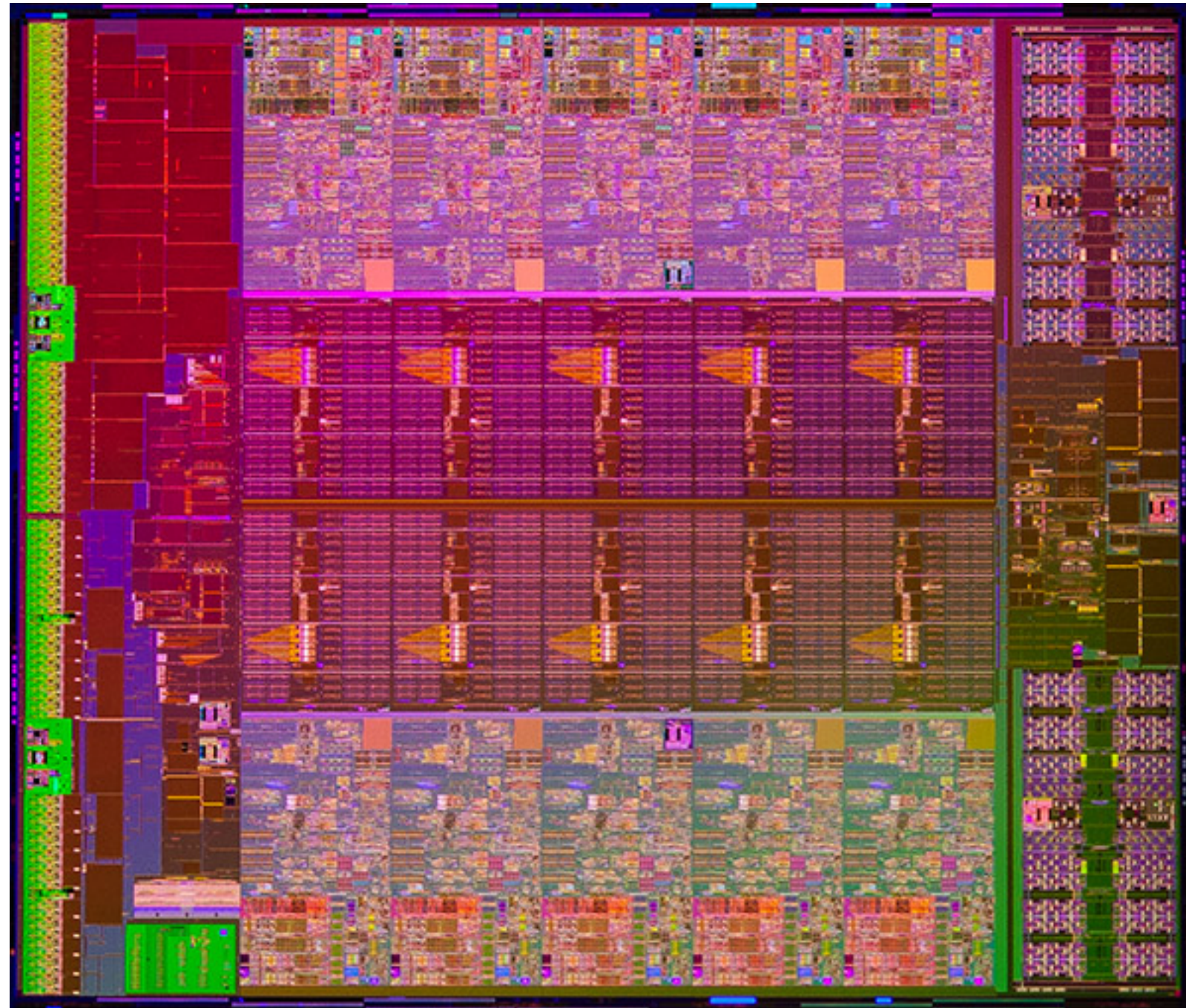
# Parallelism – Why?

The Free Lunch Is Over

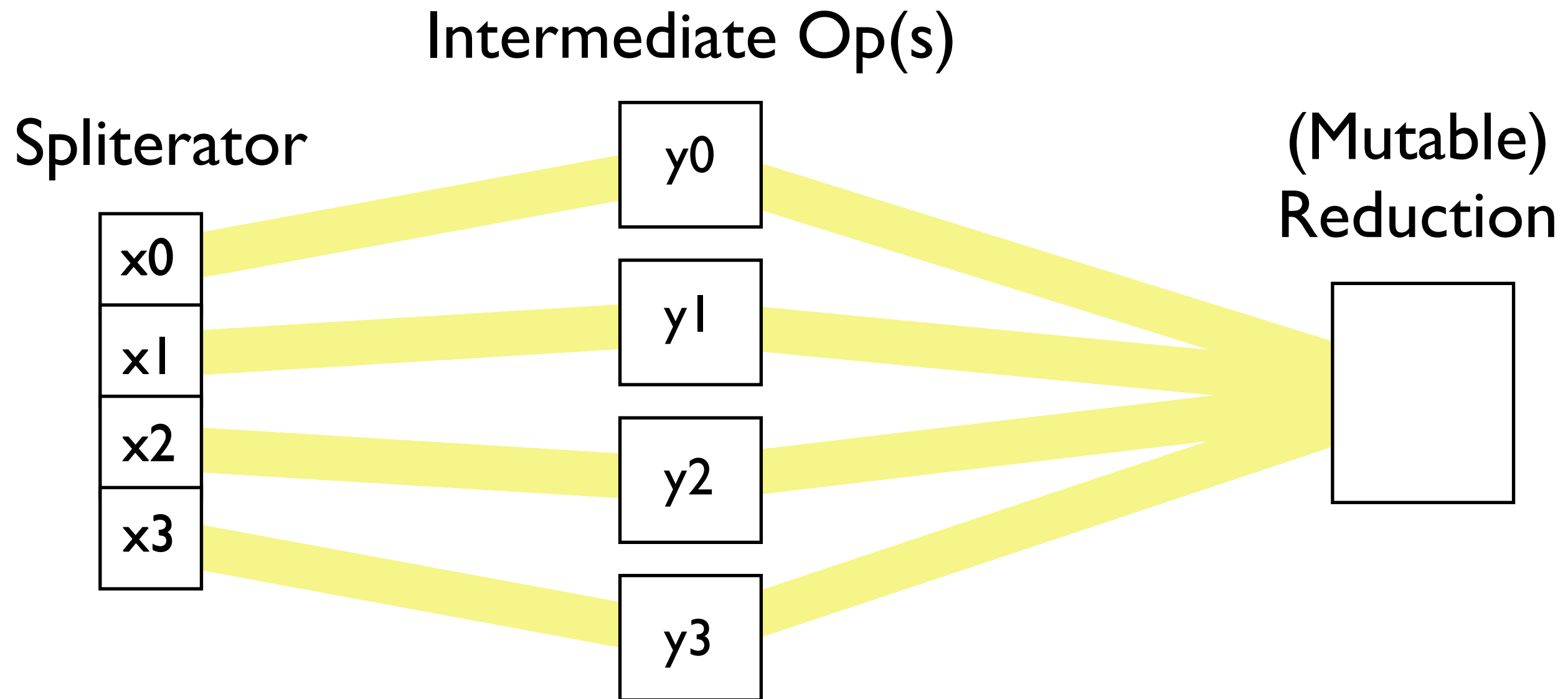
<http://www.gotw.ca/publications/concurrency-ddj.htm>



# Intel Xeon E5 2600 10-core



# Visualizing Stream Operations



# What to Measure?

---

How do code changes affect system performance?

Controlled experiment, production conditions

- difficult!

So: controlled experiment, lab conditions

- beware the substitution effect!

# 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 Optimize Code?

---

Because we don't have a problem

# Why Shouldn't We Optimize Code?

---

Because we don't have a problem

- No performance target!

# Why Shouldn't We Optimize Code?

---

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

# Why Shouldn't We Optimize Code?

---

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 Optimize Code?

---

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 Optimize Code?

---

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 Optimize Code?

---

Because we don't have a problem

- No performance target!

Else there is a problem, but not in our process

- The OS is struggling!

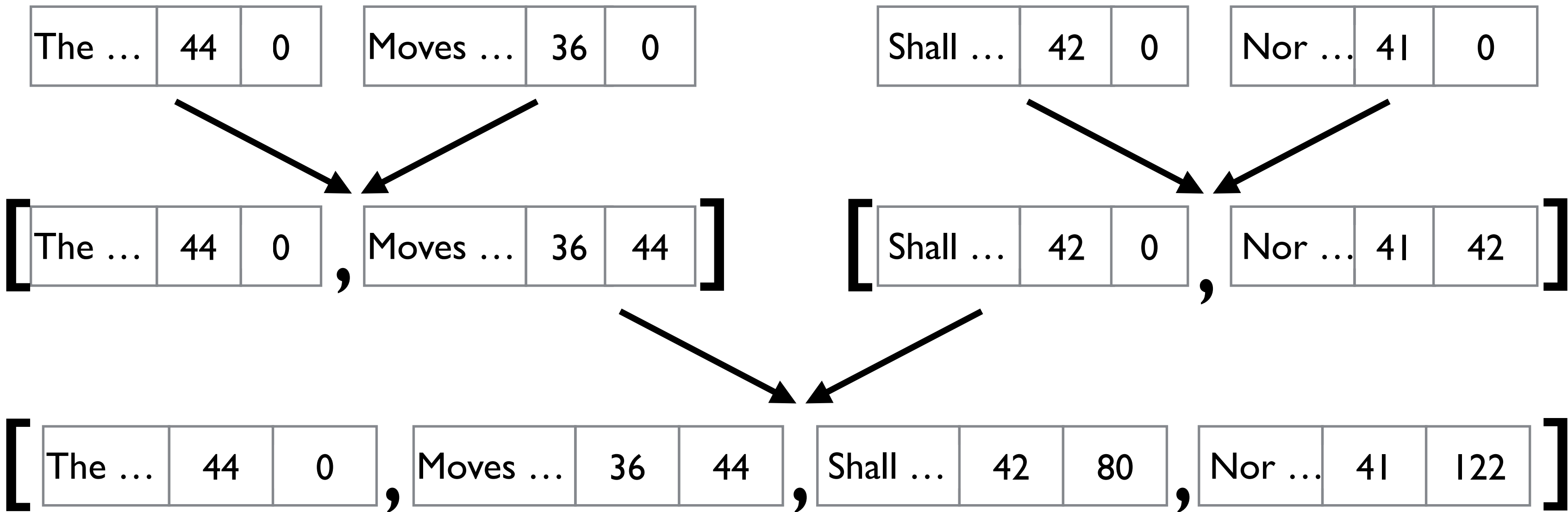
Else there's a problem in our process, but not in the code

- GC is using all the cycles!

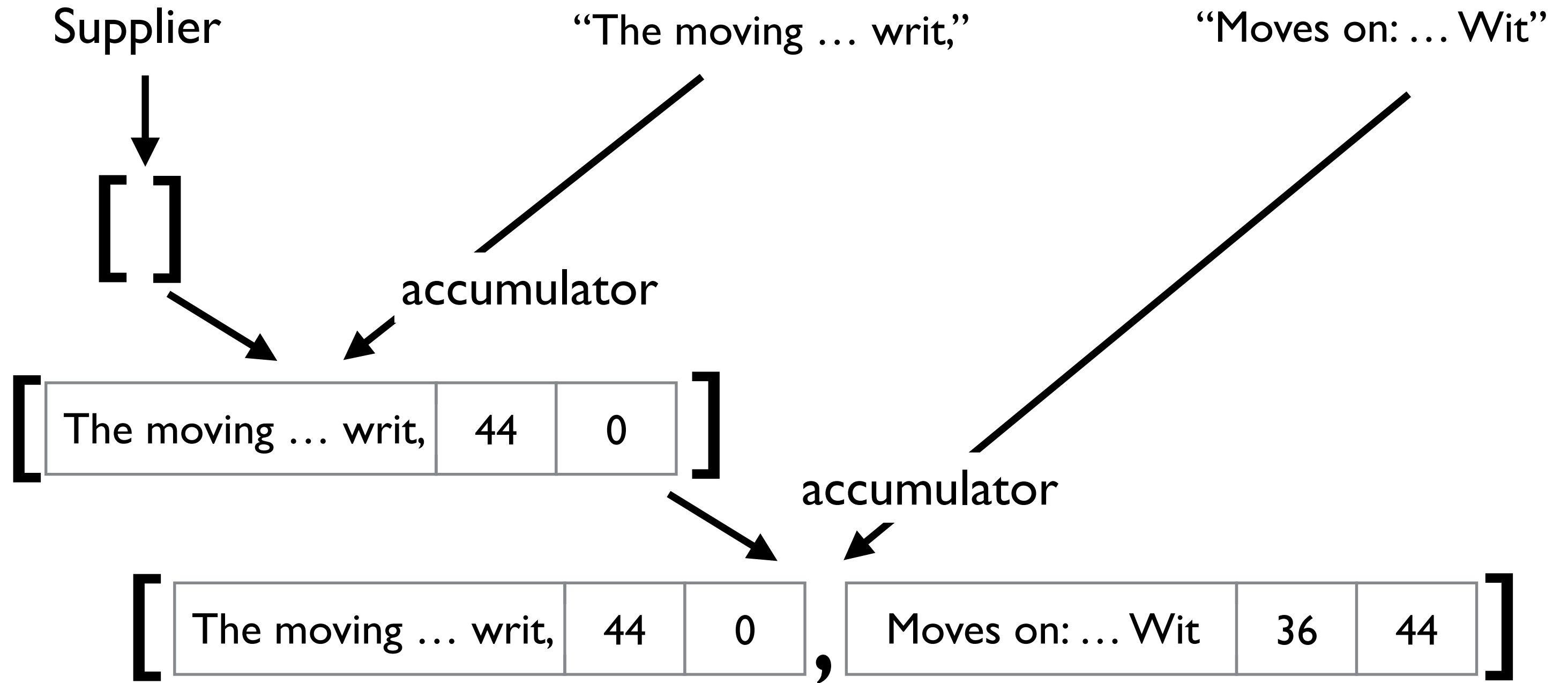
Else there's a problem in the code... somewhere

- *now* we can consider optimising!

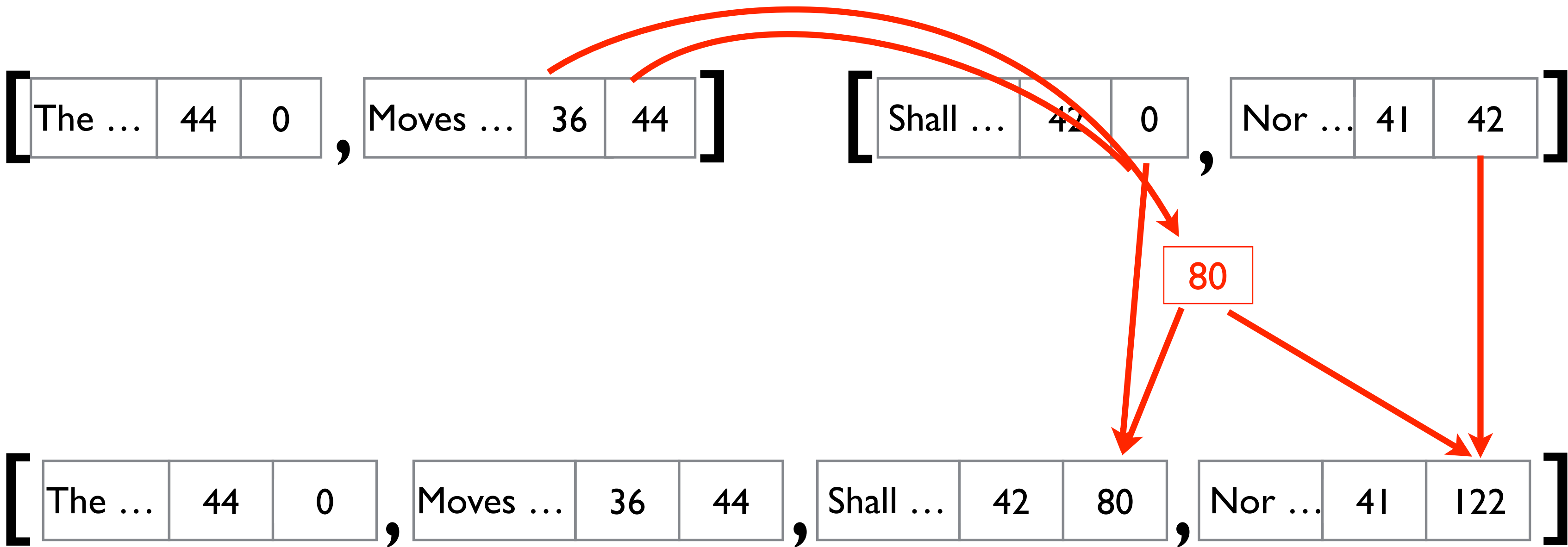
# grep -b: Collector combiner



# grep -b: Collector accumulator



# grep -b: Collector solution



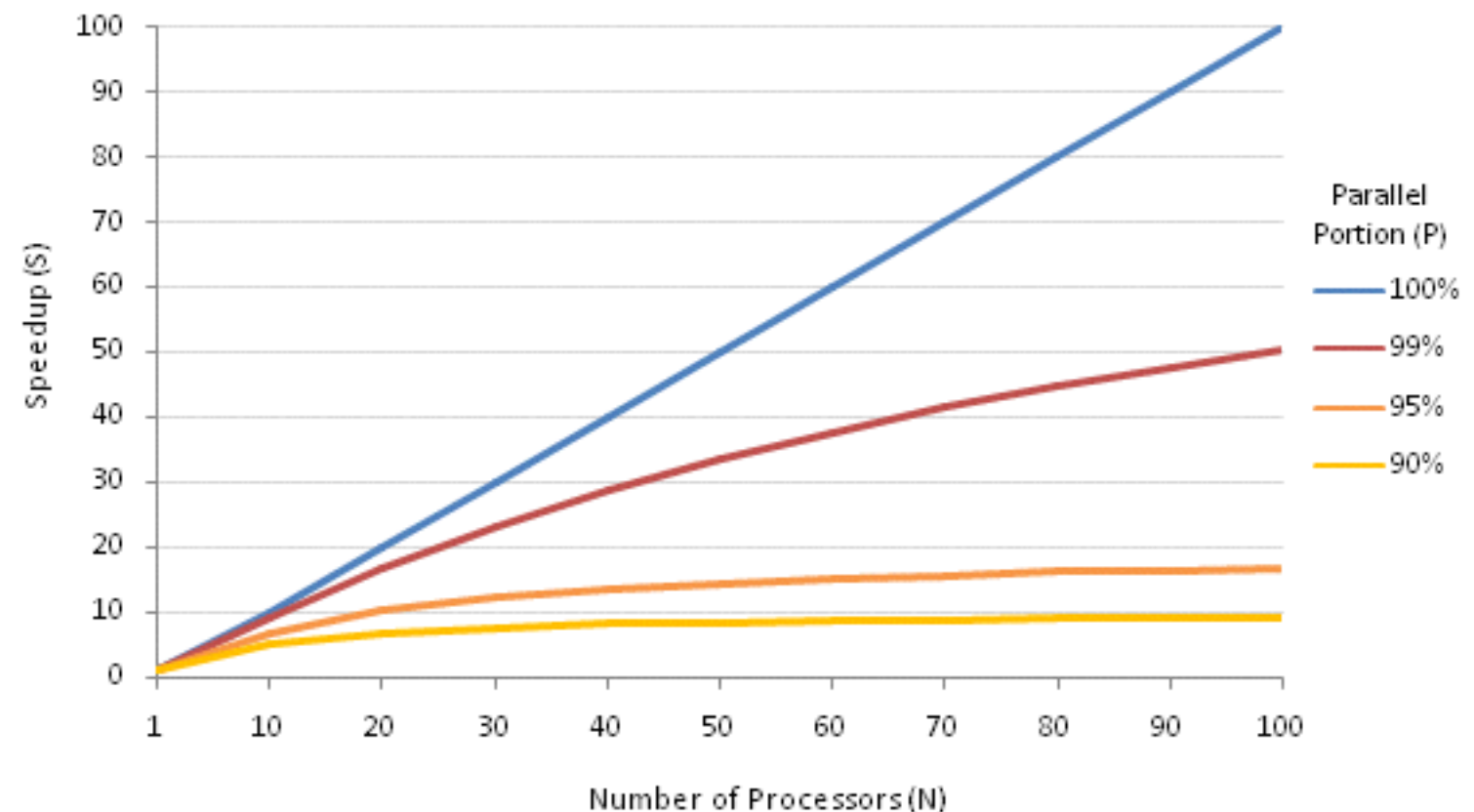
# What's wrong?

---

- 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 contains the sum of the preceding ones.
  - Combiner is  $O(n)$
- The source is streaming IO (`BufferedReader.lines()`)
- Amdahl's Law strikes:



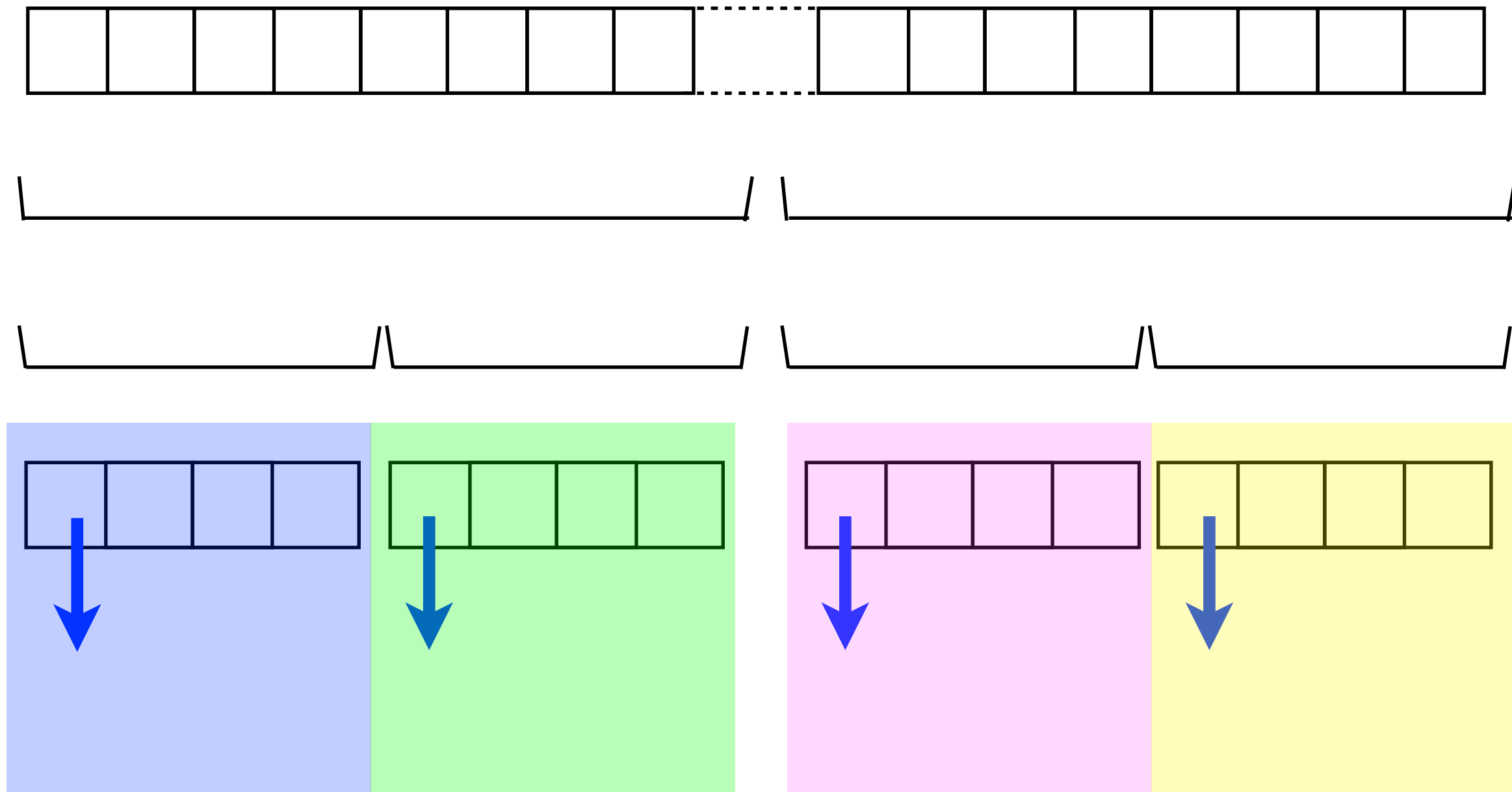
# A Parallel Solution for grep -b

---

Need to get rid of streaming IO – inherently serial  
Parallel streams need *splittable* sources

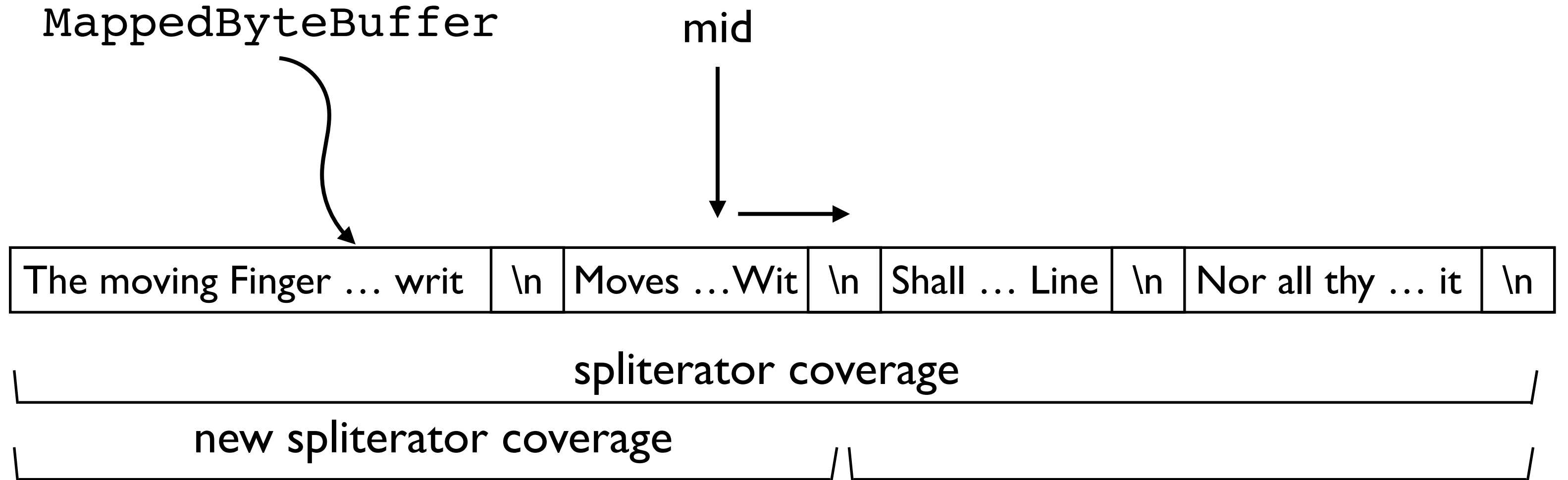
# Stream Sources

Implemented by a `Splitter`





# LineSplitter



# Parallelizing `grep -b`

---

- Splitting action of `Linesplitter` is  $O(\log n)$
- Collector no longer needs to compute index
- Result (relatively independent of data size):
  - sequential stream  $\sim 2x$  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 ( $\sim 100\mu\text{s}$ ):

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

Often quoted as  $N \times Q$

size of data set



processing cost per element



# 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 very 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/>

