Ripping Apart Java 8 Parallel Streams

Kirk Pepperdine and Heinz Kabutz

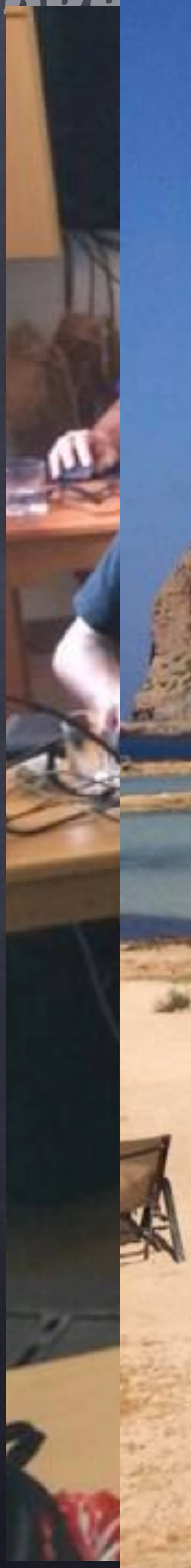
KODEWERK

JavaSpecialists.EU

About Kirk & Heinz

- Java Champions
- Teach advanced Java courses
  - Performance and Concurrency

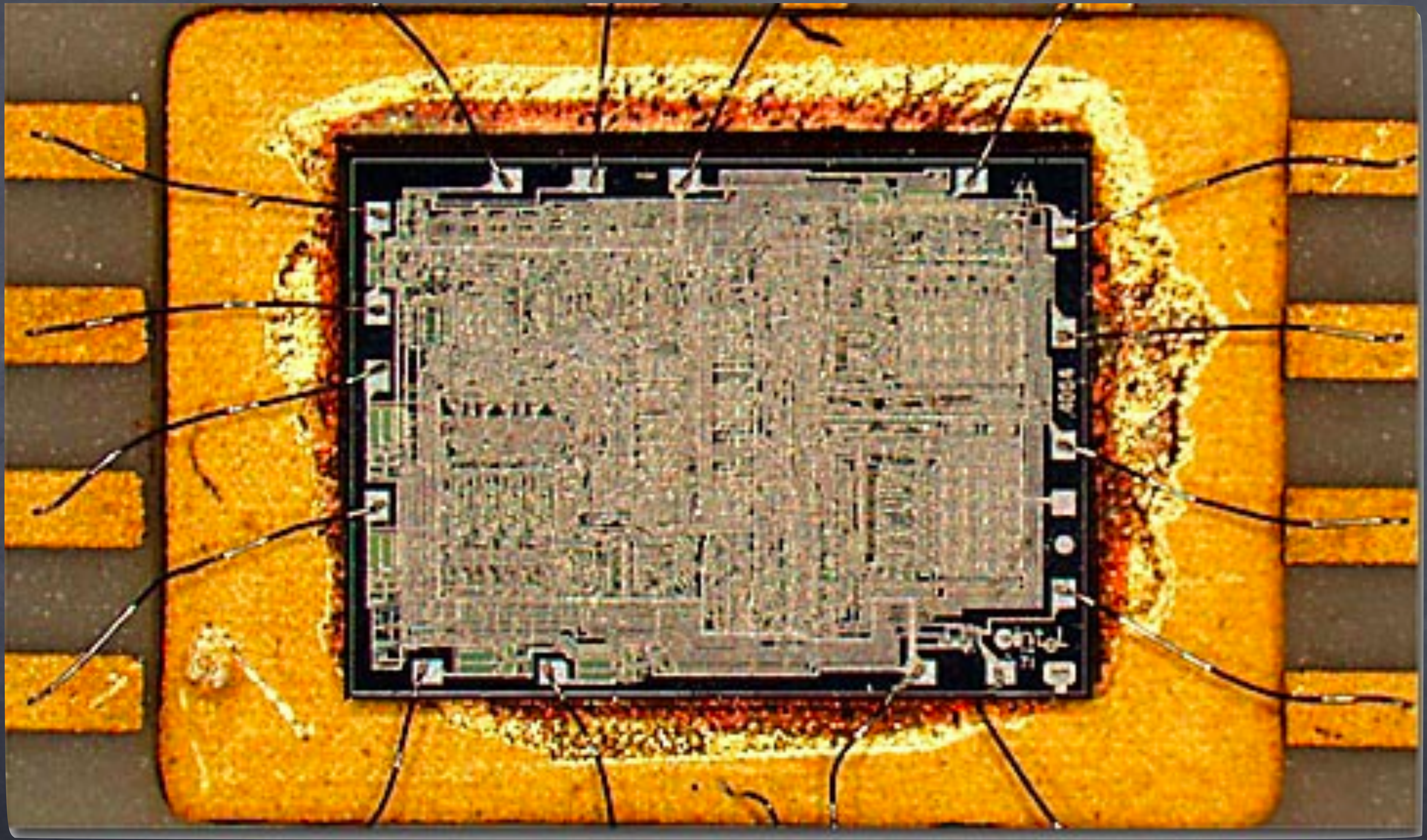
Java™ Champions




SPEAKER

KODEWERK

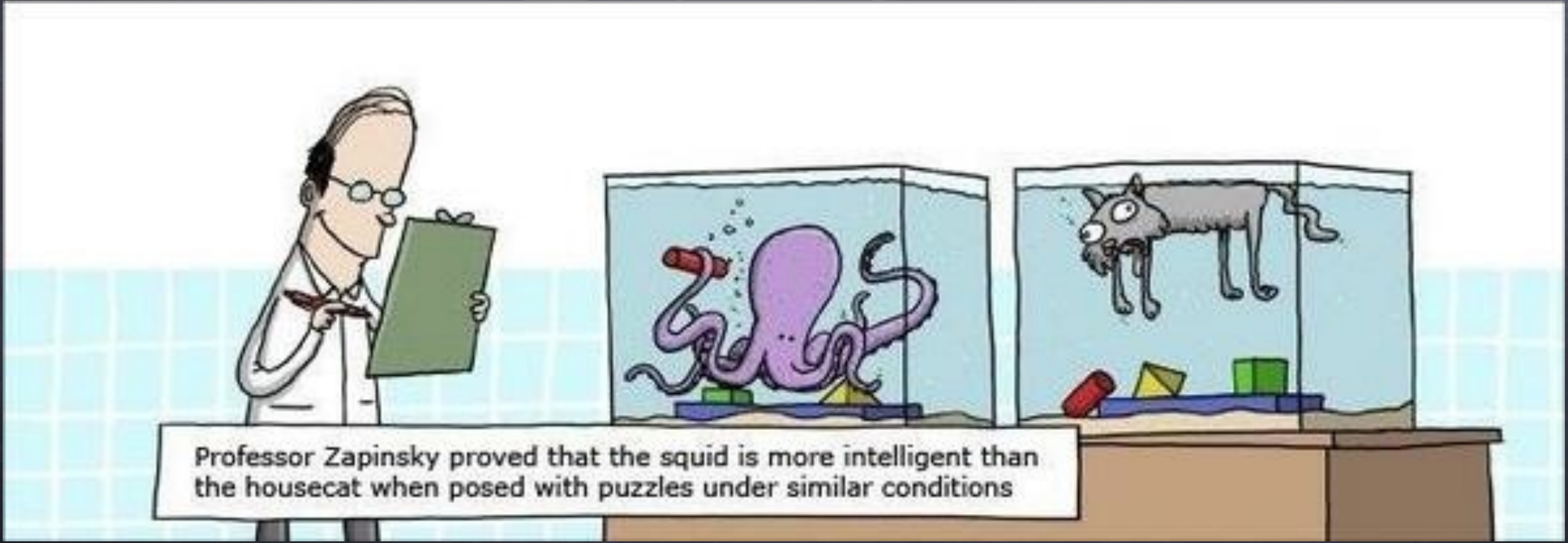
JavaSpecialists.EU



KODEWERK

JavaSpecialists.EU

Benchmark Alert!!!!!!



λ Expressions

LambdaParameters '->' LambdaBody

```
() -> 42
(x,y) -> x * y
(int x, int y) -> x * y
```

Stream

- Created from a data source (Collection??)
- Apply operators on stream of data
- Operations divided into intermediate and terminal operations
  - combine to form a stream pipeline

Stream

- Defined in interface Collection::stream()
  - many classes implement stream()
    - Arrays.stream(Object[])
    - Stream.of(Object[]), .iterate(Object,UnaryOperator)
    - File.lines(), BufferedReader.lines(), Random.ints(), JarFile.stream()

Intermediate Operators

- Lazy evaluation of filter(Predicate)
  - boolean test(T t)
- Transform elements using map(Function)
  - R apply(T t)

Terminal Operators

- Produce values
  - sum(), summaryStatistics(), forEach(Consumer)

KODEWERK

JavaSpecialists.EU

Stream Execution

- Operators combine to form a stream pipeline
  - data source -> intermediate -> intermediate -> termination
- Hitting termination operator start process

Stream Execution

- Easily parallelized
  - supported internally by providing a Spliterator
    - internal iterator that knows how to decompose the stream into sub-streams

Example

```
gcLogEntries.stream()
    .map(applicationStoppedTimePattern::matcher)
    .filter(Matcher::find)
    .map(matcher -> matcher.group(2))
    .mapToDouble(Double::parseDouble)
    .summaryStatistics();
```

data source

start streaming

map to Matcher

```
gcLogEntries.stream()
    .map(applicationStoppedTimePattern::matcher)
    .filter(Matcher::find)
    .map(matcher -> matcher.group(2))
    .mapToDouble(Double::parseDouble)
    .summaryStatistics();
```

filter out uninteresting bits

extract group

map to Double

aggregate values in the stream

KODEWERK

JavaSpecialists.EU

Parallel Streams

```
gcLogEntries.stream()
    .parallel()
    .map(applicationStoppedTimePattern::matcher)
    .filter(Matcher::find)
    .map(matcher -> matcher.group(2))
    .mapToDouble(Double::parseDouble)
    .summaryStatistics();
```

mark stream as parallel

aggregate values in the stream using Fork/Join


MANIAC!!

Fork-Join

- Support for Fork-Join added in Java 7
  - difficult coding idiom to master
- Streams make Fork-Join more reachable
  - how fork-join works and performs is important to your latency

Fork-Join

- Used internally by a parallel stream
  - break the stream up into chunks and submit each chunk as a ForkJoinTask
  - apply filter().map().reduce() to each ForkJoinTask
  - Calls ForkJoinTask invoke() and join() to retrieve results

ForkJoinPool invoke

- ForkJoinPool.invoke(ForkJoinTask) uses the submitting thread as a worker
  - If 100 threads all call invoke(), we would have 100+ForkJoinThreads exhausting the limiting resource, e.g. CPUs, IO, etc.

KODEWERK

JavaSpecialists.EU

ForkJoinPool submit/get

- ForkJoinPool.submit(Callable).get() suspends the submitting thread
  - If 100 threads all call submit(), the work queue can become very long, thus adding latency

What Does This Do?

```
synchronized (System.out) {
  System.out.println("Hello World");
  IntStream.range(0, 4).parallel().
    forEach(System.out::println);
}
```

Deadlock!

- Code output is not consistent, could be:

```
Hello World
2
3
```

- Thread dump doesn't show the deadlock directly

ManagedBlocker

- Concurrency construct that "plays nice" with ForkJoinPool

ManagedBlocker

```
interface ManagedBlocker {
  boolean block()
    throws InterruptedException;
  boolean isReleasable();
}
```

ManagedReentrantLock

- Uses the ManagedBlocker interface
  - Phaser is an example that uses ManagedBlocker
  - One day AbstractQueuedSynchronizer might support ManagedBlockers

ManagedBlocker

- In our code, instead of lock.lock(), we use

`ForkJoinPool.managedBlock(blocker)`

ManagedBlocker

- ForkJoinPool only calls the blocking method when absolutely necessary
  - We should try to achieve goal within isReleasable()

Managed Lock Code

- We will now examine the code more closely in our IDE

KODEWERK

JavaSpecialists.EU

Fork-Join Performance

- Fork Join comes with significant overhead
  - each chunk of work must be large enough to amortize the overhead

C/P/N/Q performance model

- C - number of submitters
- P - number of CPUs
- N - number of elements
- Q - cost of the operation

C/P/N/Q

- Need to offset the overheads of setting up for parallelism
  - NQ needs to be large
    - Q can often only be estimated
    - N often should be > 10,000 elements
    - C may not be your limiting constraint

KODEWERK

JavaSpecialists.EU

Kernel Times

- CPU will not be the limiting factor when
  - CPU is not saturated
  - kernel times exceed 10% of user time
- More threads will decrease performance
  - predicted by Little's Law

Common Thread Pool

- Fork-Join by default uses a common thread pool
  - default number of worker threads == number of logical cores - 1
    - Always contains at least one thread

Common Thread Pool

- Performance is tied to whichever you run out of first
  - availability of the constraining resource
  - number of ForkJoinWorkerThreads

Our own ForkJoinPool

```
ForkJoinPool ourOwnPool = new ForkJoinPool(10);
ourOwnPool.invoke(() ->
    stream.parallel(). ...
        // will be run in ourOwnPool, not commonPool()
        // documented in ForkJoinTask.fork()
```

ForkJoinPool

```
public void parallel() throws IOException {
    ForkJoinPool forkJoinPool = new ForkJoinPool(10);
    Stream<String> stream = Files.lines(new File(gcLogFileName).toPath());
    forkJoinPool.submit(() ->
        stream.parallel()
            .map(applicationStoppedTimePattern::matcher)
            .filter(Matcher::find)
            .map(matcher -> matcher.group(2))
            .mapToDouble(Double::parseDouble)
            .summaryStatistics().toString());
}
```

KODEWERK

JavaSpecialists.EU

Little's Law

- Fork-Join is a work queue
  - work queue behavior is typically modeled using Little's Law
- Number of tasks in a system equals the arrival rate times the amount of time it takes to clear an item

Little's Law

- Example: System has a requirement of 400 TPS. It takes 300ms to process a request
- Number of tasks in system = 0.300 * 417 = 125

Components of Latency

- Latency is time from stimulus to result
  - internally latency consists of active and dead time

KODEWERK

JavaSpecialists.EU

Components of Latency

- If (thread pool is set to 8 threads) and (task is not CPU bound)
  - task are sitting in queue accumulating dead time
  - make thread pool bigger to reduce dead time

From Previous Example

125 tasks in system - 8 active = 117 collecting dead time

if there is capacity to cope then
    make the pool bigger
else
    add capacity
    or tune to reduce strength of the dependency

KODEWERK

JavaSpecialists.EU

Instrumenting ForkJoinPool

- We can get the statistics needed from ForkJoinPool needed for Little's Law
  - need to instrument ForkJoinTask.invoke()

Instrumenting ForkJoinPool

```
public final V invoke() {
    ForkJoinPool.common.getMonitor().submitTask(this);
    int s;
    if ((s = doInvoke() & DONE_MASK) != NORMAL) reportException(s);
    ForkJoinPool.common.getMonitor().retireTask(this);
    return getRawResult();
}
```

- Collect invocation interval and service time
    - code is in Adopt-OpenJDK github repository

Performance Implications

- In an environment where you have many parallelStream() operations all running concurrently performance maybe limited by the size of the common thread pool

Configuring Common Pool

- Size of common ForkJoinPool is
  - Runtime.getRuntime().availableProcessors() - 1
- Can configure

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=N
-Djava.util.concurrent.ForkJoinPool.common.threadFactory
-Djava.util.concurrent.ForkJoinPool.common.exceptionHandler
```

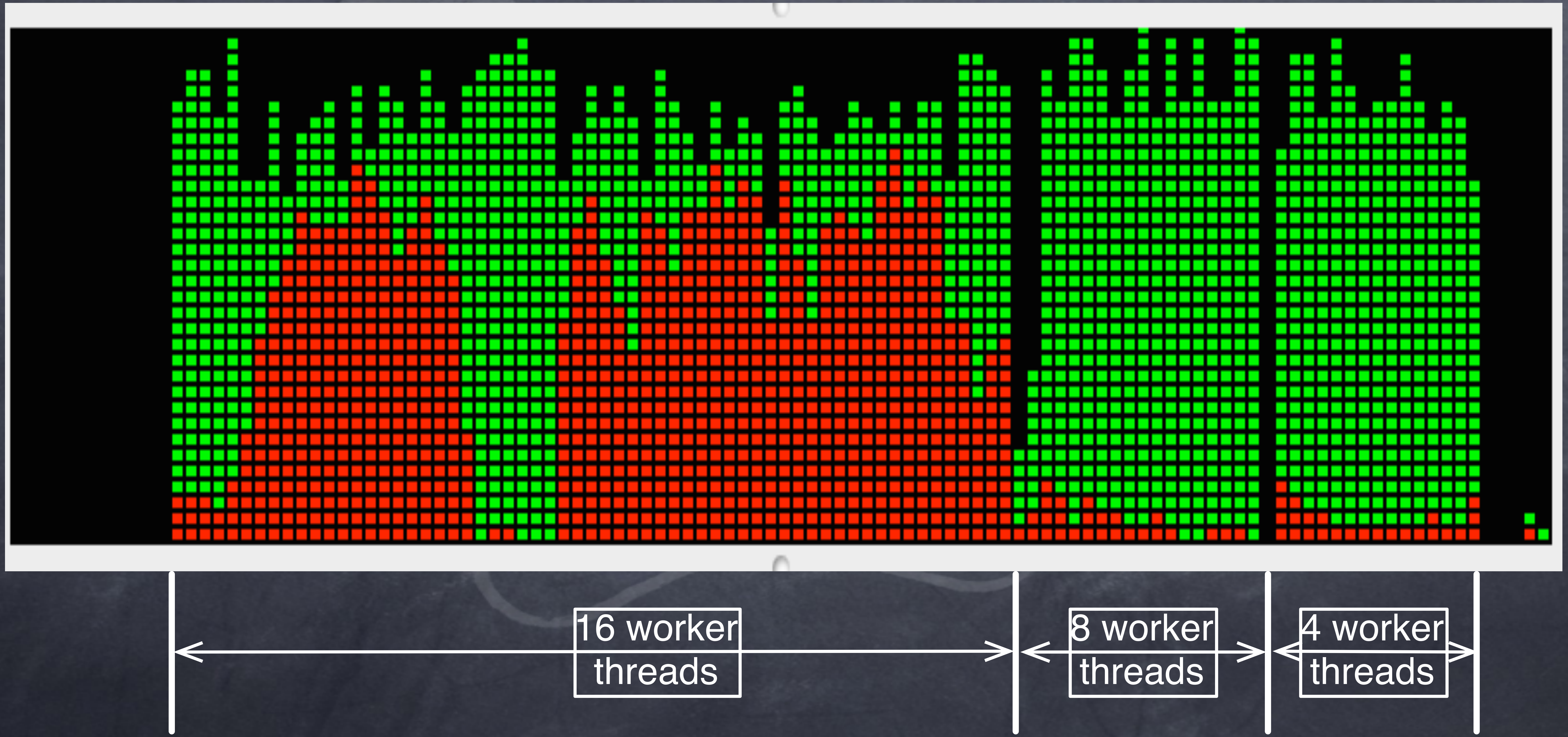
Performance Implications

- Can submit to your own ForkJoinPool
  - must call get() on pool to retrieve results
  - beware: performance will be limited by the constraining resource

```
new ForkJoinPool(16).submit(() -> ……… ).get()
```

Constraining Resource: I/O Logical Cores: 8 ThreadPool: 8	Tasks Submitted	Time in ForkJoinPool (seconds)	Inter-request Interval (seconds)	Expected Number of Tasks in ForkJoinPool	Total Run Time (seconds)
Lambda Parallel	20	2.5	2.5	1	50
Lambda Serial	0	6.1	0	0	123
Sequential Parallel	20	1.9	1.9	1	38
Concurrent Parallel	20	3.2	1.9	1.7	39
Concurrent Flood (FJ)	20	6.0	1.9	3.2	38
Concurrent Flood (stream)	0	2.1	0	0	41

Constraining Resource: CPU Logical Cores: 8 ThreadPool: 8	Tasks Submitted	Time in ForkJoinPool (seconds)	Inter-request Interval (seconds)	Expected Number of Tasks in ForkJoinPool	Total Run Time (seconds)
Lambda Parallel	20	2.8	2.8	1	56
Lambda Serial	0	7.5	0	0	150
Sequential Parallel	20	2.6	2.6	1	52
Concurrent Parallel	20	5.8	3.0	1.9	60
Concurrent Flood (FJ)	20	43	6.5	6.6	130
Concurrent Flood (stream)	0	3.0	0	0	61


16 worker threads
8 worker threads
4 worker threads

Going parallel might not give you the gains you expect

You may not know this until you hit production!

Monitoring internals of JDK is important to understanding where bottlenecks are

JDK is not all that well instrumented

APIs have changed so you need to re-read the javadocs

even for your old familiar classes

Extreme Java Concurrency,
June 10-12, Chania, Greece
javaspecialists.eu
Java Performance Tuning,
May 26-29, Chania Greece
www.kodewerk.com