From Concurrent to Parallel

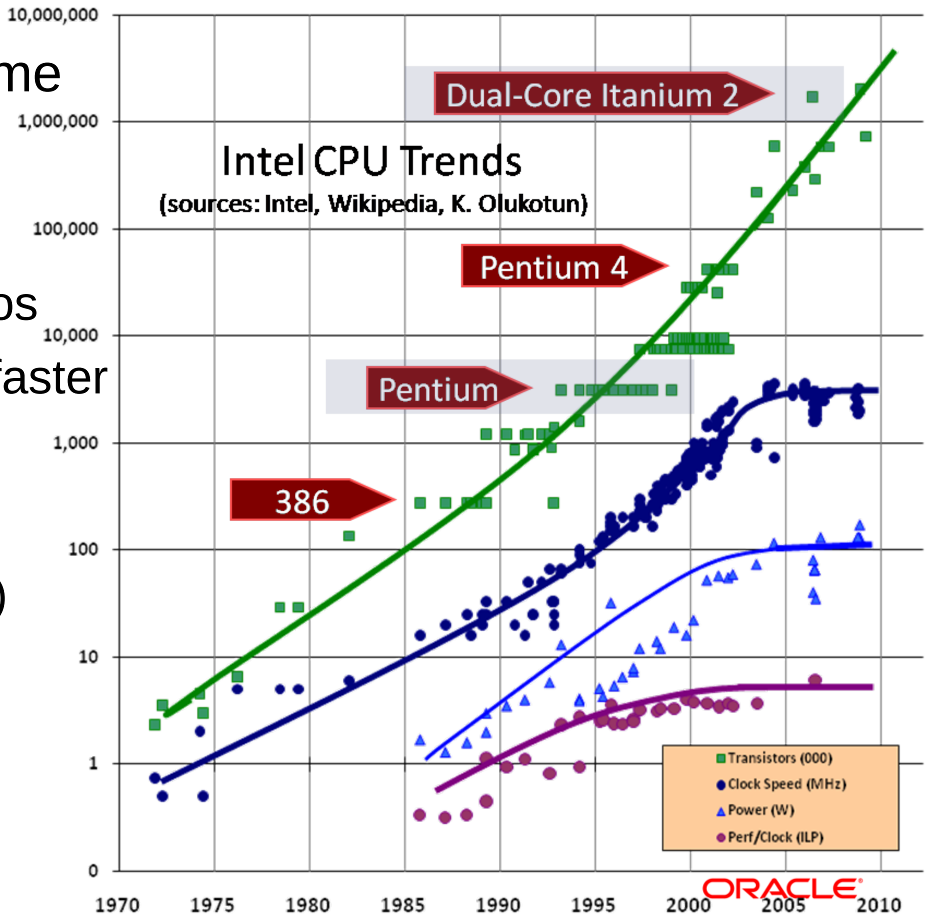
Brian Goetz, Java Language Architect

ORACLE

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.

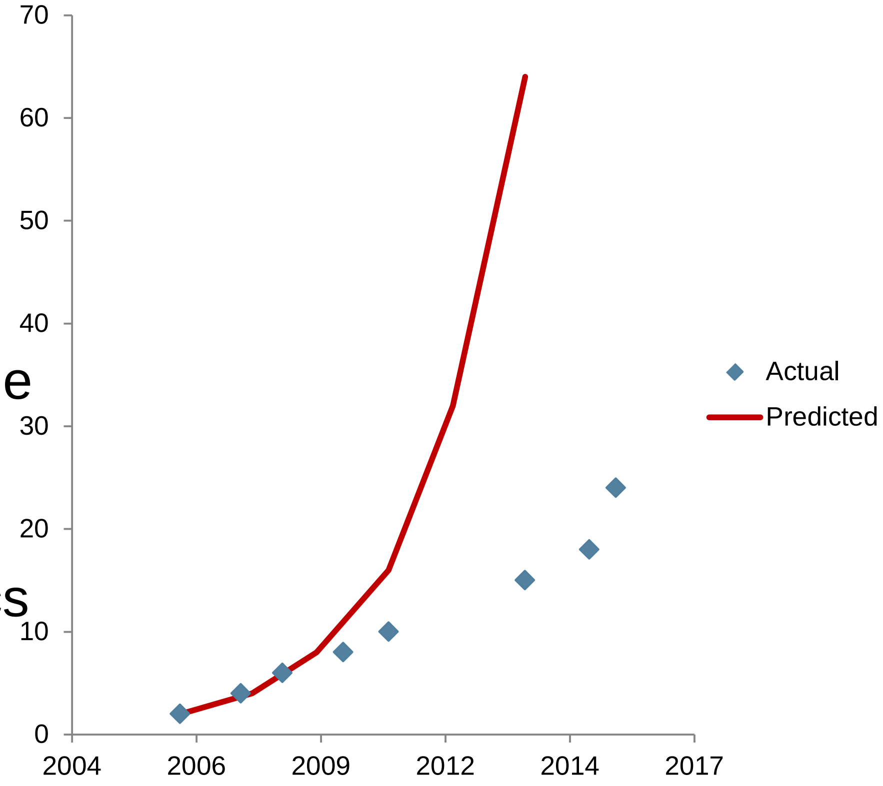
Hardware Trends

- Graph shows Intel CPUs over time
  – (graph courtesy Herb Sutter)
- Moore's Law still in full force
  – Transistor count doubles ~18mos
  – Now giving us *more* cores, not faster
- CPU clock rates stopped going up in 2003
  – (some period of denial followed)



Dude, Where's My Cores?

- If Moore's law is plowed into core count, we'd see red line
- Reality is blue marks
  - For enterprise chips (-EX)
  - Lower for consumer chips
- Chipmakers not delivering all the cores they can
  - Because software isn't ready!
- Except for data-parallel analytics
  - And GPUs



Concurrency, Through The Ages

- Primary goal of using concurrency is maximizing use of CPU(s)
  - Though sometimes used to manage program structure (e.g., CSP, Actors)
- Dominant maximization strategy changes over time
  - Single-core era
    - Alternative to blocking – nonblocking IO, prioritized background tasks
  - Multi-core era
    - Coarse-grained, task-based concurrency
    - Largely about *throughput* – pushing more requests through a server
  - Many-core era
    - Fine-grained data parallelism
    - Largely about *latency* – use more cores to get the answer faster

ORACLE

Hardware Trends Drive Software Trends

- Languages, libraries, and frameworks shape the programs we write
  - We follow the path of least resistance…
- Hardware shapes the languages, library, and frameworks we write
  - Java 1 supported threads, locks, condition queues
  - Java 5 added thread pools, blocking queues, concurrent collections
  - Java 7 added the fork-join library
  - Java 8 added parallel streams

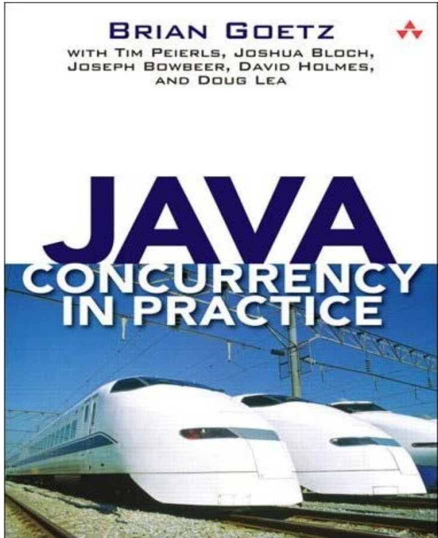
ORACLE

Terminology

- Sadly, definitions for *concurrency* and *parallelism* are not standard
  - Often (mistakenly) used interchangeably
- Historically…
  - *Concurrency* is a property of a program's *structure*
    - Organized as the interaction between multiple cooperating activities
  - Parallelism is a property of a program's *execution*
    - Do things *really* happen simultaneously, or is this just an illusion?
  - Concurrency is the *potential for* parallelism
  - Useful distinction when true concurrent execution was mostly a theoretical concern
  - Less useful distinction today

Terminology

- More commonly today…
  - *Concurrency* is about *correctly* and *efficiently* controlling access to shared resources
    - Example: constructing thread-safe data structures
    - Primitives: Locks, events, semaphores, coroutines, STM
  - *Parallelism* is about using additional resources to produce an answer faster
    - Example: searching a large data set by partitioning
- Why should we care?
  - Concurrency is hard!
    - Reasoning about shared state and locks requires wizardry
    - Not easy even with secret wizard spell book!
  - Parallelism is much easier!
    - Standard trick is *partitioning*
    - And a little bit of discipline



Parallelism

- Parallelism is about using more resources to get the answer faster
  - ***Strictly an optimization!***
  - If additional resources are not available, can still compute sequentially
- Corollary: Only useful if it really does get the answer faster!
- Just because we use more resources …
  - Doesn't mean the computation is always faster than a sequential one
  - Or even as fast…
- Analyze → implement → measure → repeat...
  - Prefer sequential implementation until parallel is proven effective
- Measure of parallel effectiveness is *speedup*
  - How much faster (or slower) compared to sequential?

Parallelism

- A parallel computation *always involves more work* than the best sequential alternative
  - How could it not? It still has to solve the problem!
  - And also:
    - Decompose the problem
    - Launch tasks, manage tasks, wait for tasks to complete
    - Combine results
- Parallel version always starts out “behind”
  - We hope to make up for this initial deficit by burning more resources
  - To succeed, we need
    - A parallelizable problem
    - A good implementation
    - Good runtime support for execution
    - Enough data

Exploitable Parallelism

- Not all problems parallelize equally!
- Given a function *f*, define

$g(0) = f(0)$

$g(n) = f(\, g(n-1)\,),\ for\ n > 0$

$h(0) = f(0)$

$h(n) = f(n) + h(n-1),\ for\ n > 0$

- How well can we parallelize computing *g* and *h*?
  - Their definitions look very similar
  - Are they equally parallelizable?

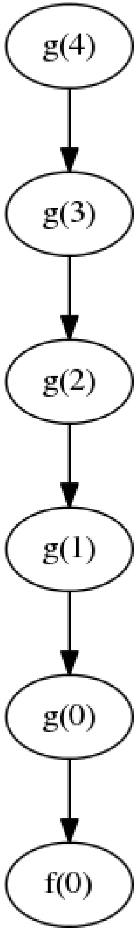
Exploitable Parallelism

$g(0) = f(0)$

$g(n) = f(\ g(n\text{-}1)\ ),\ \text{for}\ n > 0$

- Parallelizing g turns out to be a lost cause
- Can rewrite as

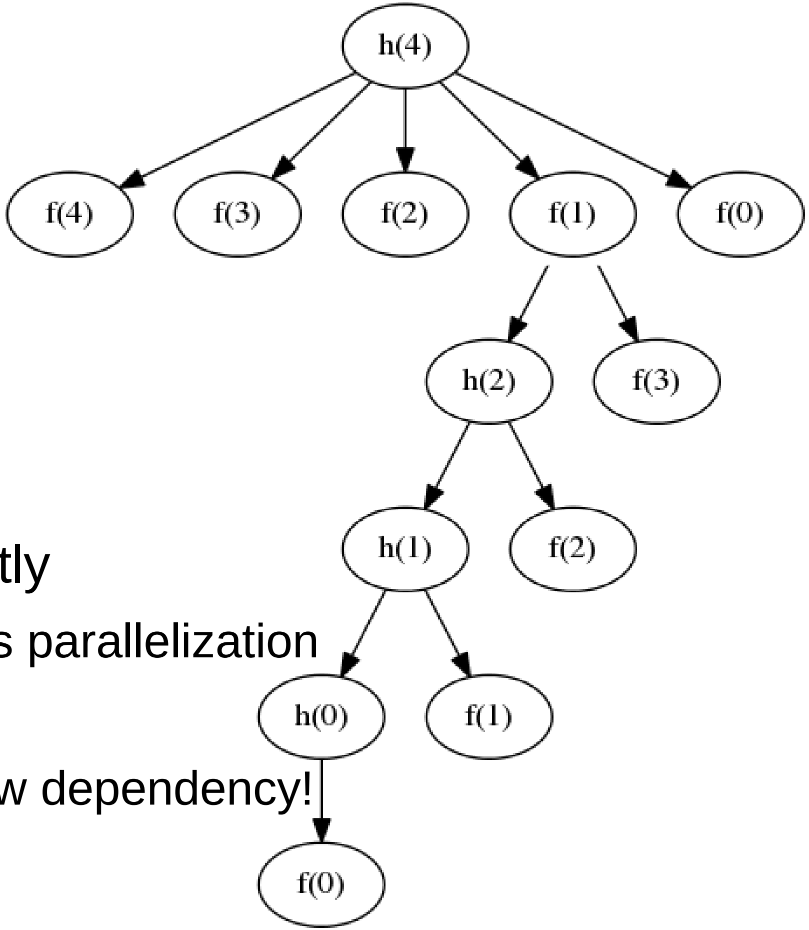
  $g(n) = f(\ f(\ \ldots\ {\scriptstyle n\ \text{times}}\ \ldots\ f(\ 0\ )\ \ldots\ )$
- Can't compute f(f(0)) until we know f(0)
  - Problem is *fundamentally sequential*
  - Parallelism limited by dataflow dependency



Exploitable Parallelism

h(0) = f(0)

h(n) = f(n) + h(n-1), for n > 0



- Parallelizing h turns out to be easy!
- Can rewrite as
  h(n) = f(1) + f(2) + … f(n)
- Can compute each term independently
  - Then add them up, which also admits parallelization
  - Problem is *embarassingly parallel*
  - Though beware of *accidental* dataflow dependency!

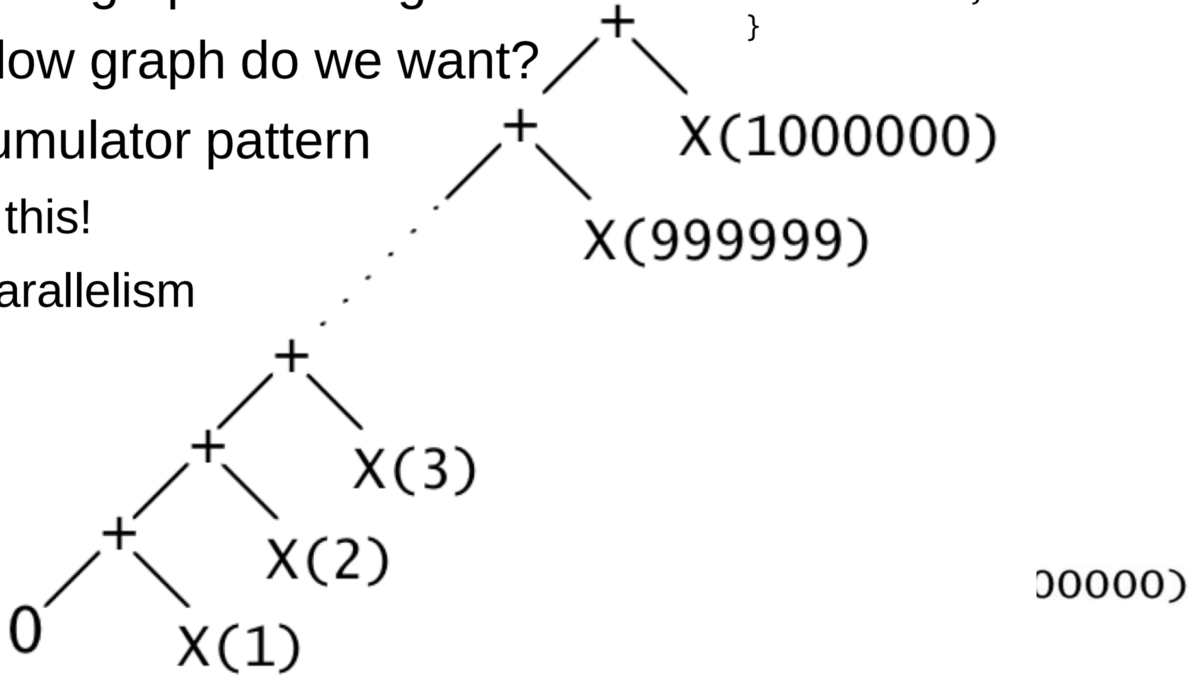
Exploiting Parallelism

- Despite similar-looking definition, *h* was parallelizable but *g* was not
  - But, a naïve implementation of h would have a dataflow graph just like g!
- Its not enough to have exploitable parallelism
  - You have to structure the computation to actually exploit it!
  - Many of the techniques we naturally use in sequential algorithms are impediments in parallel ones
    - Need to unlearn some bad habits

Towards Parallel Computation

```java
int sumSeq(int[] array) {
    int sum = 0;
    for (int i : array)
        sum = sum + i;
    return sum;
}
```

- Simple problem: add numbers from 1..n
- What kind of dataflow graph do we get?
- What kind of dataflow graph do we want?
- Problem #1 – Accumulator pattern
  - Need to unlearn this!
  - Impediment to parallelism



ORACLE

Towards Parallel Computation

- Might try solving the problem with concurrency
- But, the obvious approach is broken!
  - Data race on every access to sum
  - Will get the wrong answer

```
int sumBroken(int[] array) {
    int mid = array.length / 2;
    int sum = 0;
    CONCURRENT {
        {
            for (int i = 0; i < mid; i++)
                sum = sum + array[i];
        }
        {
            for (int i = mid; i < array.length; i++)
                sum = sum + array[i];
        }
    }
    return sum;
}
```

Towards Parallel Computation

- We can fix this, of course…
  - But now it is *much* slower than sequential!
  - Cores are stalled waiting for lock, not doing work
- Problem has exploitable parallelism
  - Failed attempt to exploit it

```
int sumConcurrent(int[] array) {
    int mid = array.length / 2;
    int sum = 0;
    CONCURRENT {
        {
            for (int i = 0; i < mid; i++)
                ATOMIC { sum = sum + array[i]; }
        }
        {
            for (int i = mid; i < array.length; i++)
                ATOMIC { sum = sum + array[i]; }
        }
    }
    return sum;
}
```


Shared State

- There are three ways to safely handle state…
  – Don't share
  – Don't mutate
  – Coordinate access
- The first two are far easier to get right than the third…
  – Let's try "don't share"
  – Partition the array in two chunks, and operate on them separately

Towards Parallel Computation

- First cut at a parallel solution
- Decompose the problem into subproblems
- Solve the subproblems
- Combine the result
- How will this perform?
  - Given enough data ….
    - OK on 1 or 2 cores
    - No further speedup for N > 2 cores
- No shared access to mutable state

```
int sumPartitioned(int[] array) {
    int mid = array.length / 2;
    int leftSum = 0, rightSum = 0;
    CONCURRENT {
        {
            for (int i = 0; i < mid; i++)
                leftSum = leftSum + array[i];
        }
        {
            for (int i = mid; i < array.length; i++)
                rightSum = rightSum + array[i];
        }
    }
    return leftSum + rightSum;
}
```

Divide And Conquer

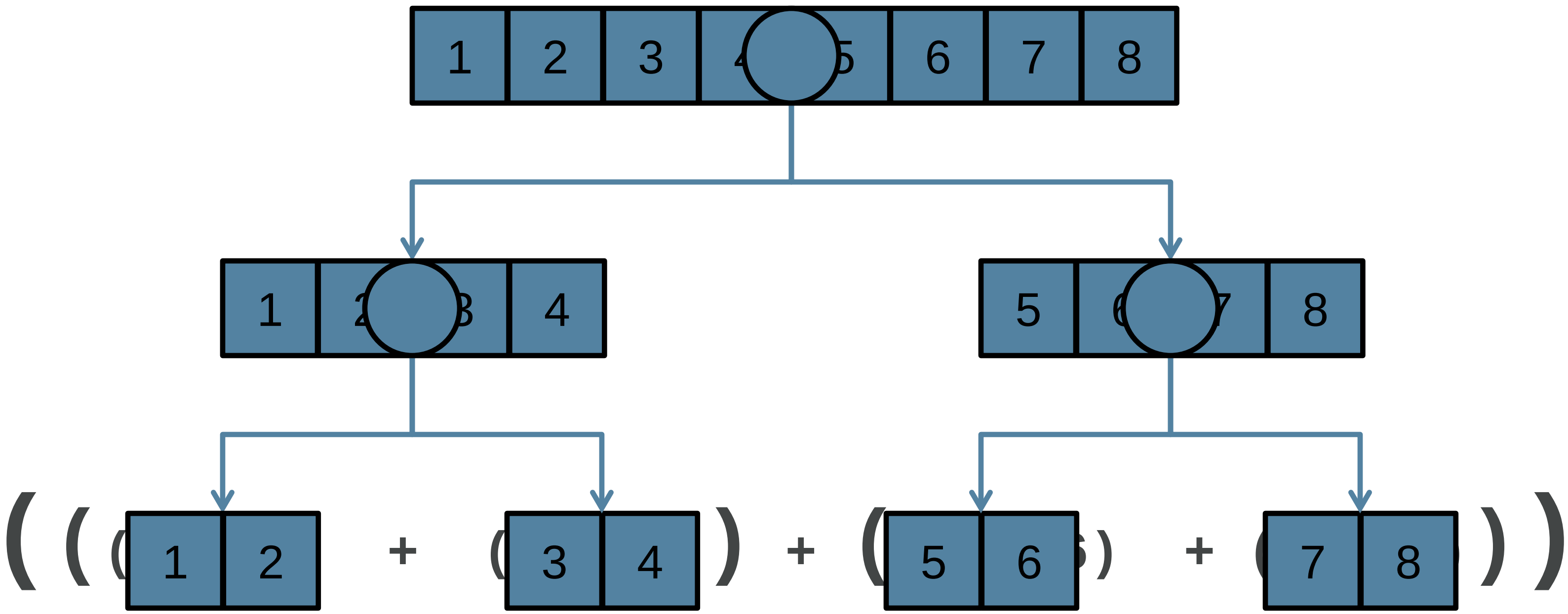
- Standard tool for parallel execution is *divide-and-conquer*
  – Recursively decompose problem until it is small enough for sequential

```
R solve(Problem<R> problem) {
    if (problem.isSmall())
        return problem.solveSequentially();
    R leftResult, rightResult;
    CONCURRENT {
        leftResult = solve(problem.left());
        rightResult = solve(problem.right());
    }
    return problem.combine(leftResult, rightResult);
}
```

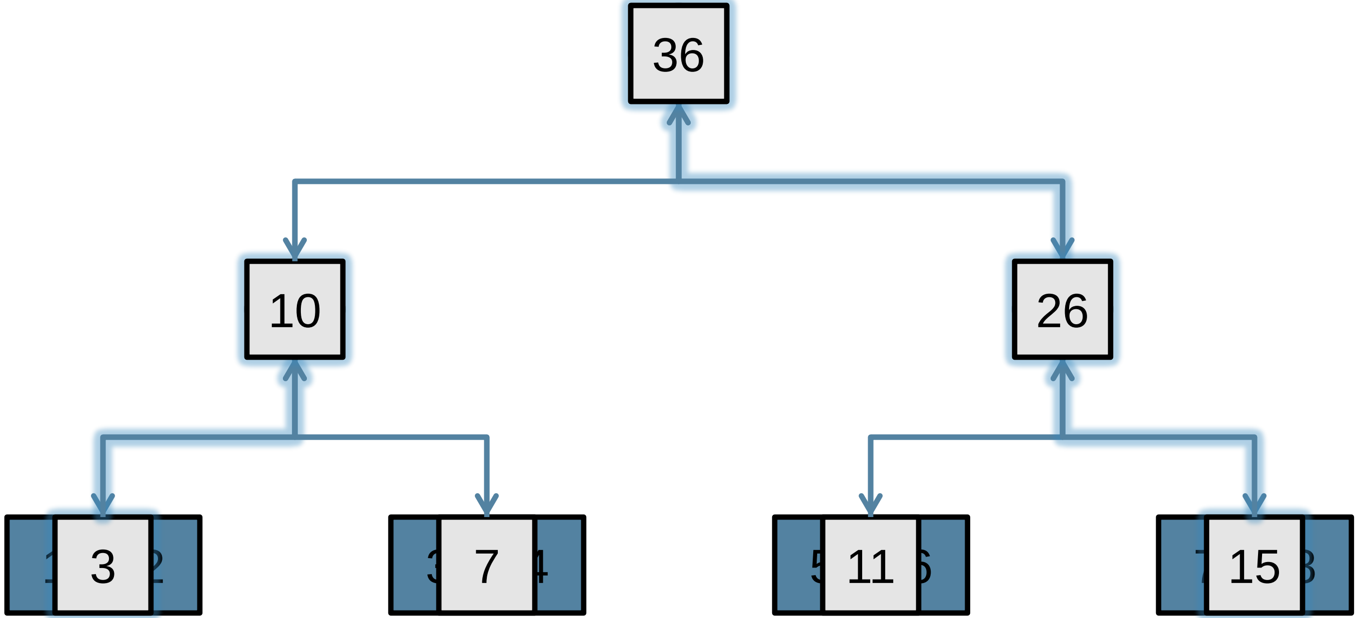

Divide And Conquer

- Recursive decomposition is *simple*
  - Especially with recursively-defined data structures, like trees
  - No shared mutable state – just partitioned reading
  - Intermediate results live on the stack
- Starts forking work early!
  - Beware Amdahl's Law
- Decomposition is dynamic
  - Can incorporate runtime knowledge of core count and load
  - Portable expression of parallel computation

Summing an array in parallel



Summing an array in parallel



Performance Considerations

- Splitting / decomposition costs
  - Sometimes splitting is more expensive than just doing the work!
- Task dispatch / management costs
  - Can do a lot of work in the time it takes to hand work to another thread
- Result combination costs
  - Sometimes combination involves copying lots of data
- Locality
  - The elephant in the room
- Each can steal away potential speedup!
  - In general, need a lot of data to make up for decomposition startup

Fork-Join

- Java SE 7 added the *fork-join* framework to java.util.concurrent
  - Task management framework for *fine-grained*, *CPU-intensive* tasks
    - Scales well from 1 thread to hundreds
  - Specialized and optimized for divide-and-conquer
  - Based on concept of *work stealing*
    - Minimizes contention costs
- Two basic operations
  - Fork a task
  - Wait for (or get callback on) task completion
- Efficient substrate for our CONCURRENT { t1, t2 } construct
  - Relatively low task-management overhead

Streams

- Streams is about *possibly-parallel*, *aggregate operations* on datasets
- Sources can be collections, arrays, generator functions, IO…
  - Support (including parallel) deeply woven into Collections
- Encourages a declarative style – what, not how
  - If done well, more readable and less error-prone
- Pipelines built from basic primitives – filter, map, reduce, sort
  - Exploits laziness – all operations fused into a single pass on the data
- All operations can be executed in parallel
  - But not magic parallelism dust!
- Couldn't get to a library like this without lambdas

ORACLE

Streams

```java
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Seller>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

```java
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sorted(comparing(Seller::getName))
    .forEach(s -> System.out.println(s.getName()));
```

 | |

Parallel Stream Performance

- Splitting / decomposition costs
  - How easily splittable is the source?
- Task dispatch / management costs
  - Handled by FJ framework
- Result combination costs
  - Adding numbers is cheap; merging sets is expensive
- Locality
  - Array-based sources are best

The NQ Model

- Simple model for parallel performance
  - N = number of data items
  - Q = amount of work per item
- Rule of thumb
  - Need NQ > 10,000 to have a chance for parallel speedup

Source Splitting

- Some sources split better than others
  - Cost of computing split
  - Evenness of split
  - Predictability of split
- Arrays split cheaply, evenly, and with perfect knowledge of split sizes
  - Linked lists have none of these properties
  - Iterative generators behave like linked lists, stateless generators behave like arrays
- Compare
  - `IntStream.iterate(0, i -> i+1).limit(n).sum()`
  - vs `IntStream.range(0, n).sum()`

ORACLE

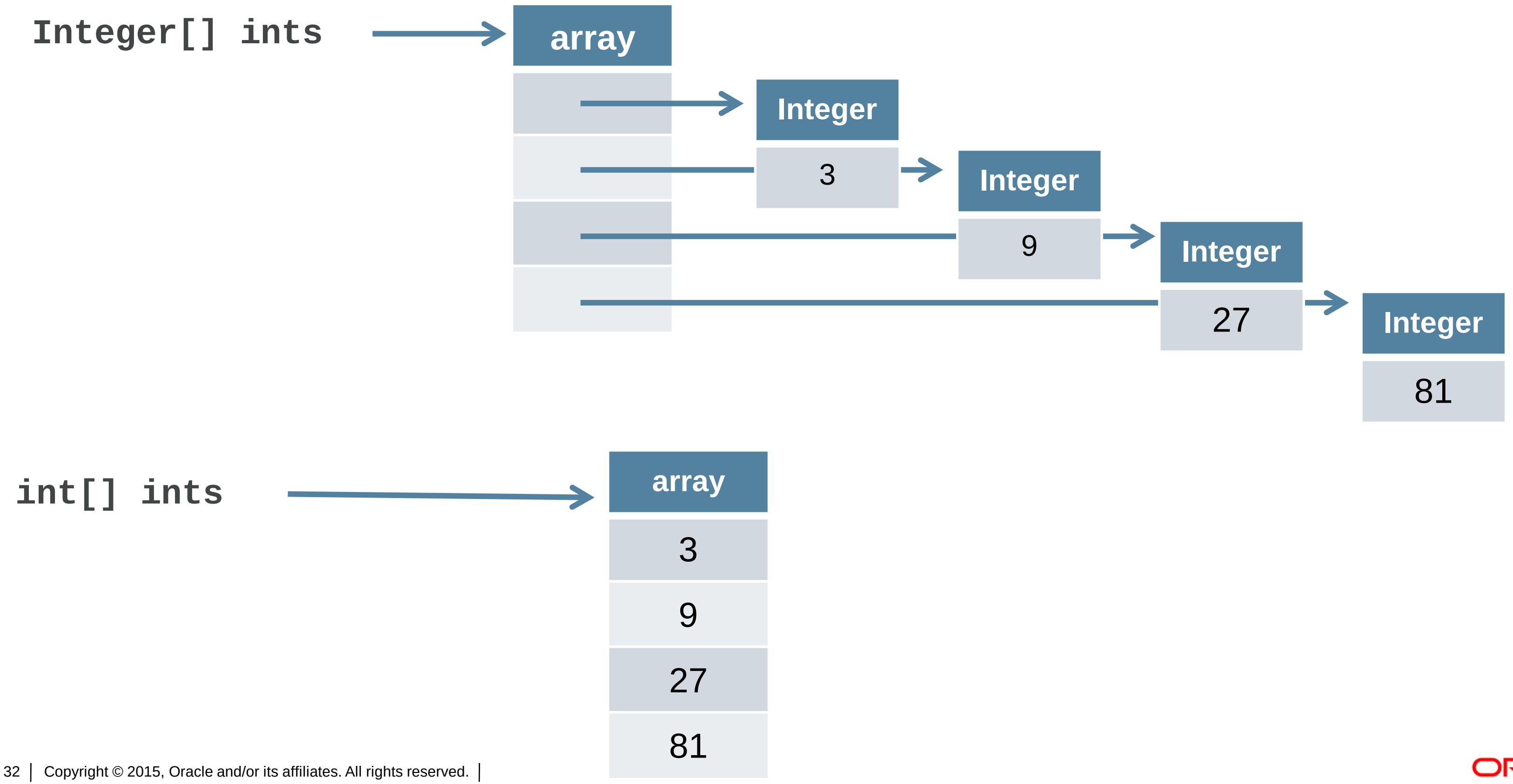
Locality

- Locality is the elephant in the room
- Parallelism wins when we can keep the CPUs busy doing useful work
  - Waiting for cache misses is not useful work
- Memory bandwidth often the limiting factor on many systems
- Array-based, numeric problems parallelize best
- Benchmark: Stream.of(int[]).sum() vs Stream.of(Integer[]).sum()
  - 8-core i7, Java SE 8, Linux

Speedup over Sequential	N=1k	N=10k	N=1M
int	1x	6.2x	7.9x
Integer	(4.9x)	1.5x	3.5x

 | |

Locality



 | | ORACLE

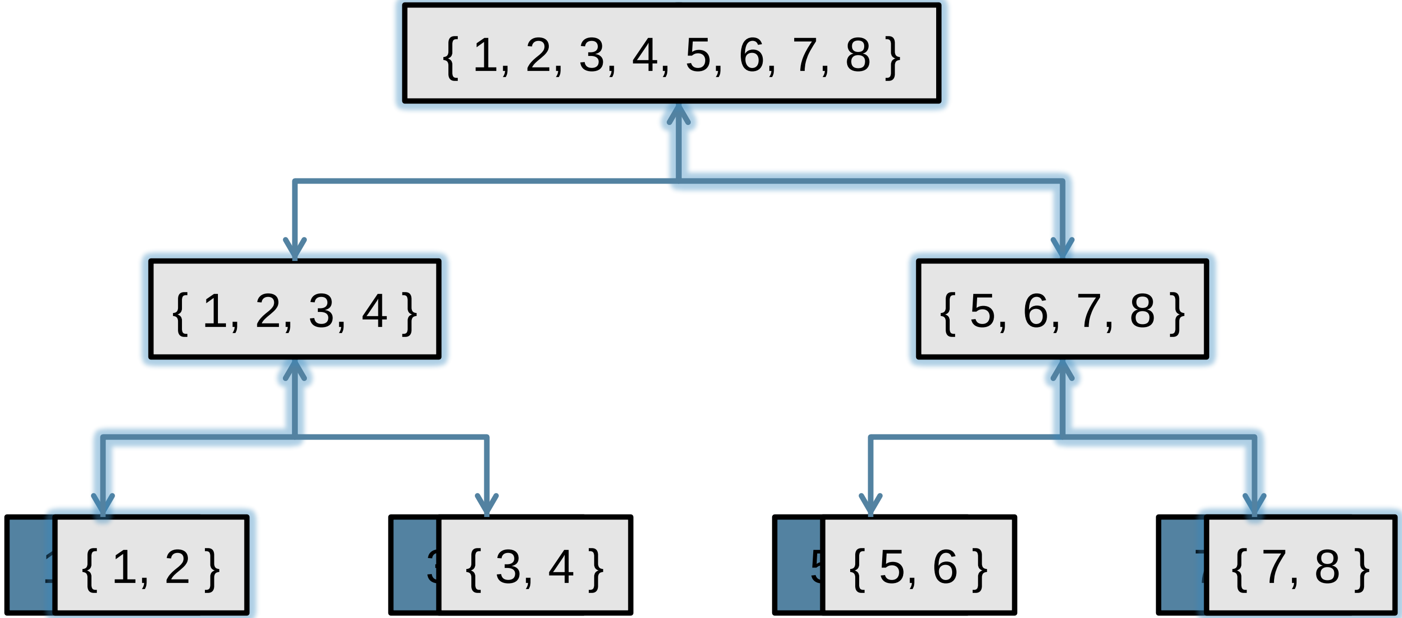
Encounter Order

- Some operations have semantics tied to *encounter order*
  - Encounter order is the order implied by the source
  - Some sources have no defined encounter order (e.g., HashSet)
  - Operations like limit(), skip(), and findFirst() are tied to encounter order
  - Less exploitable parallelism
- Sometimes the encounter order is meaningful, sometimes not
  - Call .unordered() to indicate encounter order is not meaningful to you
  - Ops like limit(), skip(), and findFirst() will optimize in the presence of unordered sources

Merging

- For some operations (sum, max) the merge operation is really cheap
- For others (groupingBy to a HashMap) it is insanely expensive!
  - Involves a lot of copying
  - And repeatedly, up the tree
  - Cost of merging overwhelms the parallelism advantage
- Measuring IntStream.range(0, n).collect(toSet())…
  - For n=10K, approximately 4x *slowdown* going parallel

Merging a set in parallel



Parallel Streams

- Any of the following factors can conspire to undermine speedup
  - NQ is insufficiently high
  - Cache-miss ratio is too high (too many indirections)
  - Source is expensive to split
  - Result combination cost is too high
  - Pipeline uses encounter-order-sensitive operations

Summary

- Streams are cool!
- Parallelism is cool!
- But… parallelism is an optimization
  – And parallel streams are not magic performance dust
- Before optimizing, always …
  – Have actual performance requirements
  – Have reliable performance measurements (not easy!)
  – Ensure that your performance doesn’t meet requirements

