
ORACLE®

JavaOne™

ORACLE®

Thinking in Parallel

Stuart Marks
Brian Goetz
Java Platform Group, Oracle

Java Your Next (Cloud)

Thinking in Parallel

Thinking

Trivial Example

- Convert an array of strings to upper case
  - Conventional (iterative) approach
  - Streams (aggregate) approach

Convert Array of Strings to Upper Case

```
String[] upcase(String[] input) {
    String[] result = new String[input.length];

    for (int i = 0; i < input.length; i++) {
        result[i] = input[i].toUpperCase();
    }

    return result;
}
```

Convert Array of Strings to Upper Case

a b c d e

Convert Array of Strings to Upper Case

a b c d e

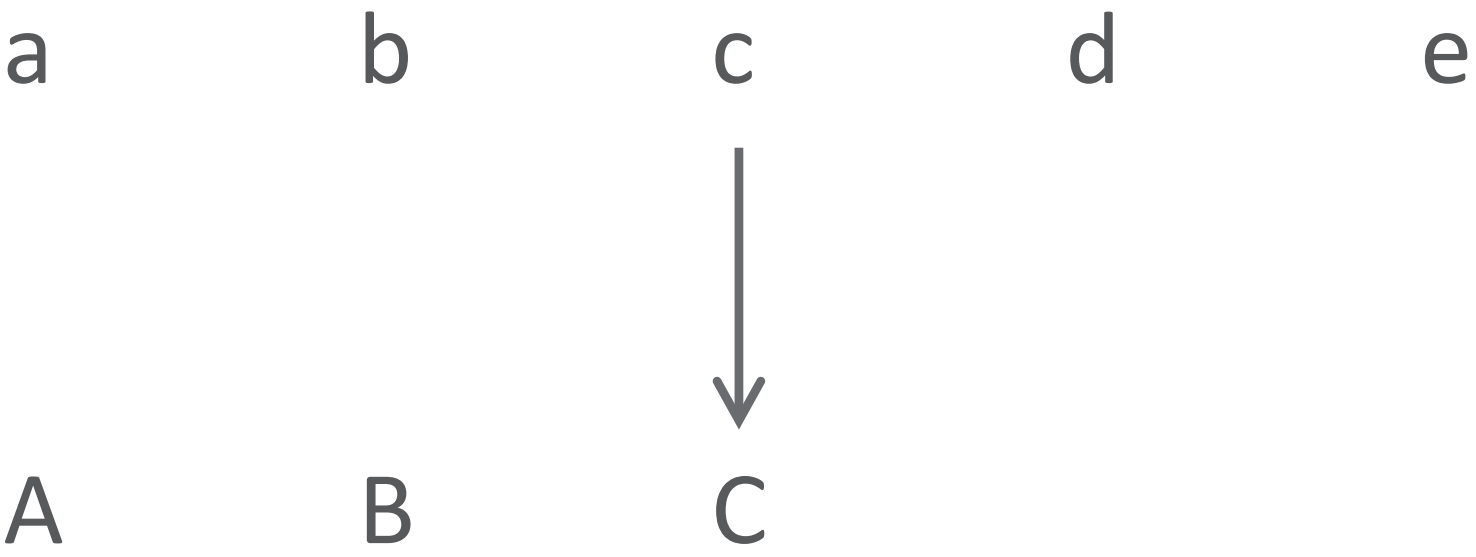
A

Convert Array of Strings to Upper Case

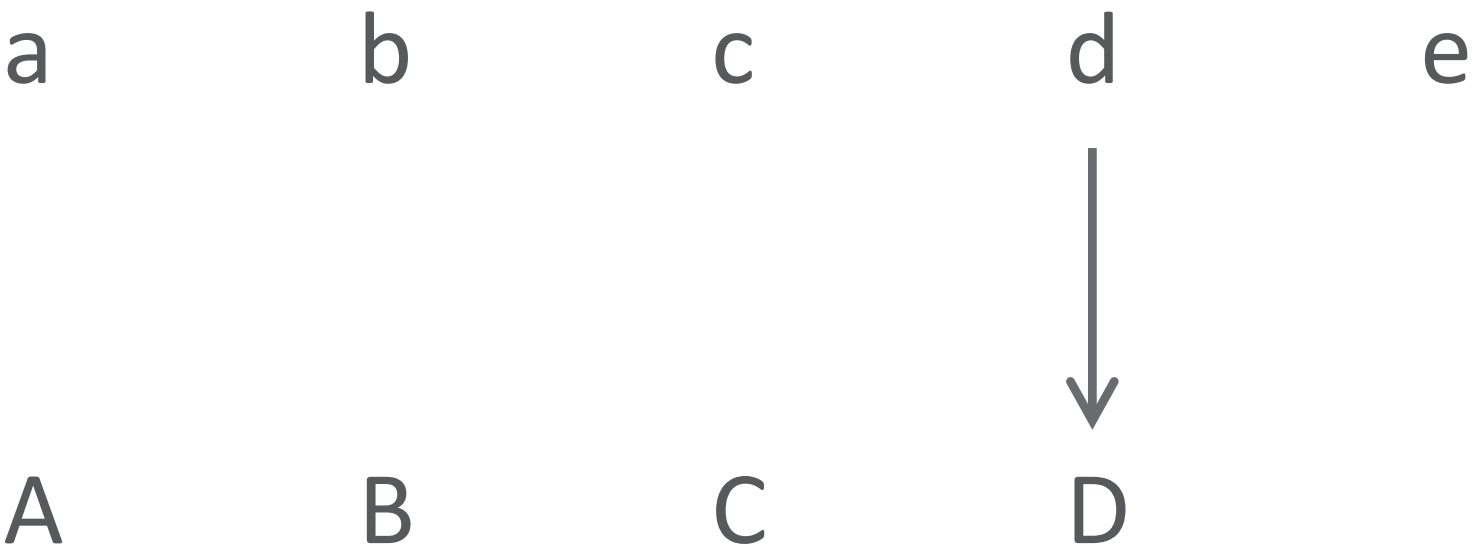
a b c d e

A B

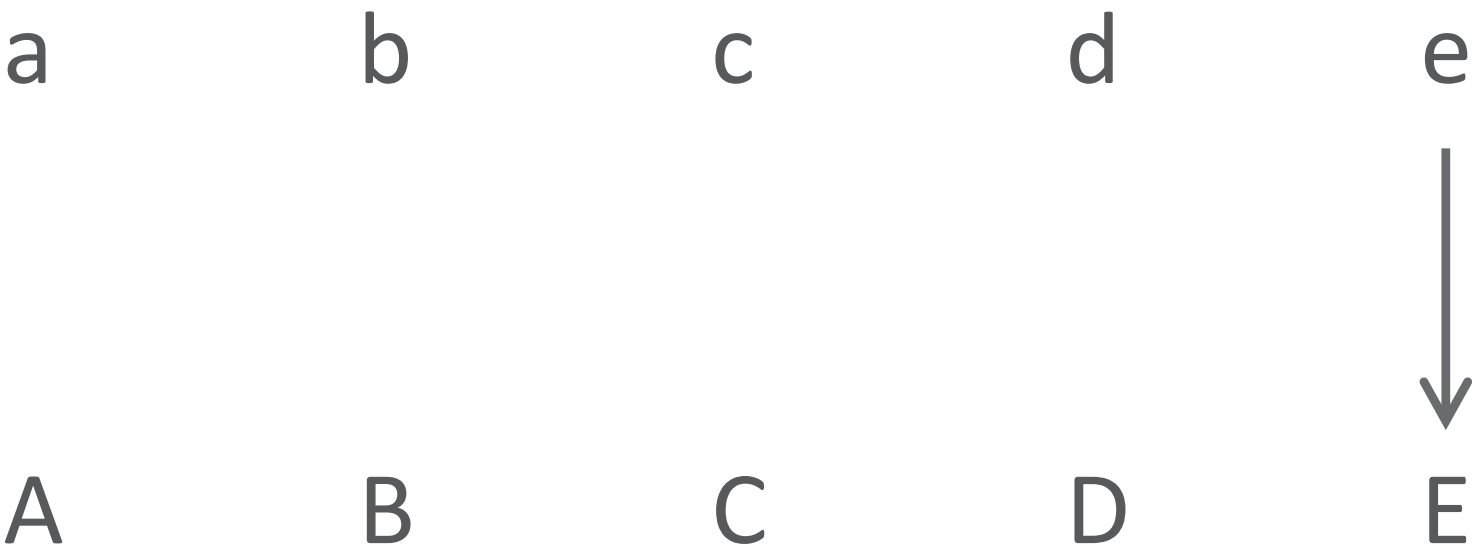
Convert Array of Strings to Upper Case



Convert Array of Strings to Upper Case



Convert Array of Strings to Upper Case

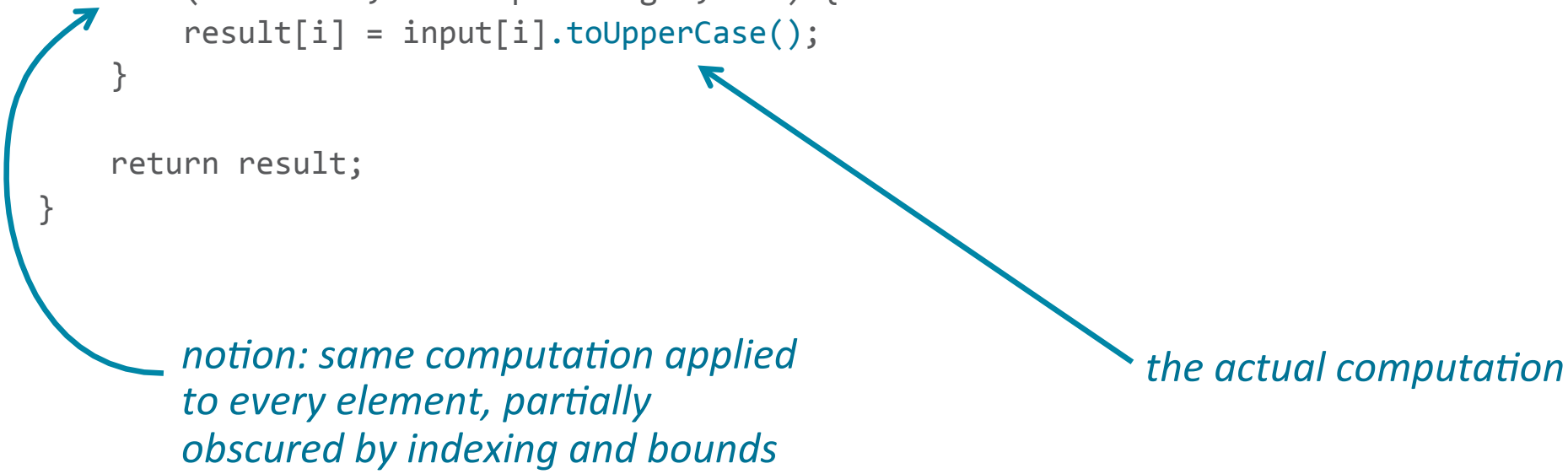


What Parts Are Essential?

```
String[] upcase(String[] input) {
    String[] result = new String[input.length];

    for (int i = 0; i < input.length; i++) {
        result[i] = input[i].toUpperCase();
    }

    return result;
}
```



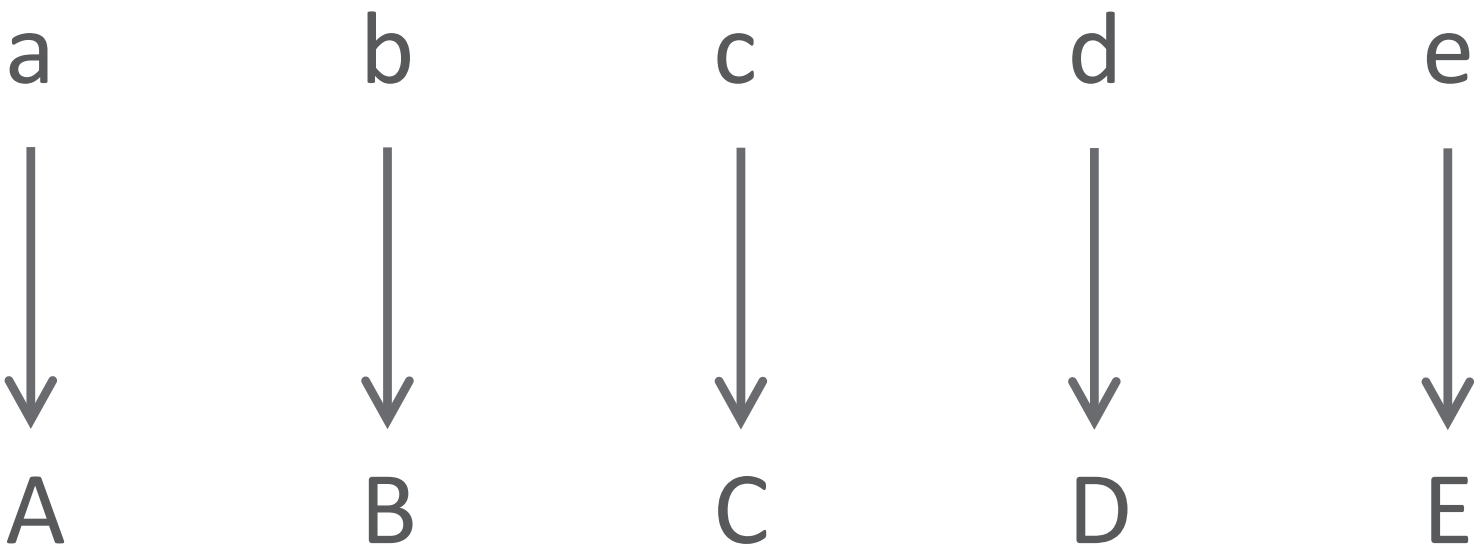
Observations

- For-loop processing
  - Sequential
  - Left-to-right ordering
- Most of this is accidental, not essential
  - Key: each upper-case computation is independent of all others
    - (this is important, stay tuned)
  - But why are they done sequentially, in order?
  - With a for loop, that's all we've got!
    - you have to do extra work to do anything else
    - enhanced-for ("for-each") loop helps, but only a little

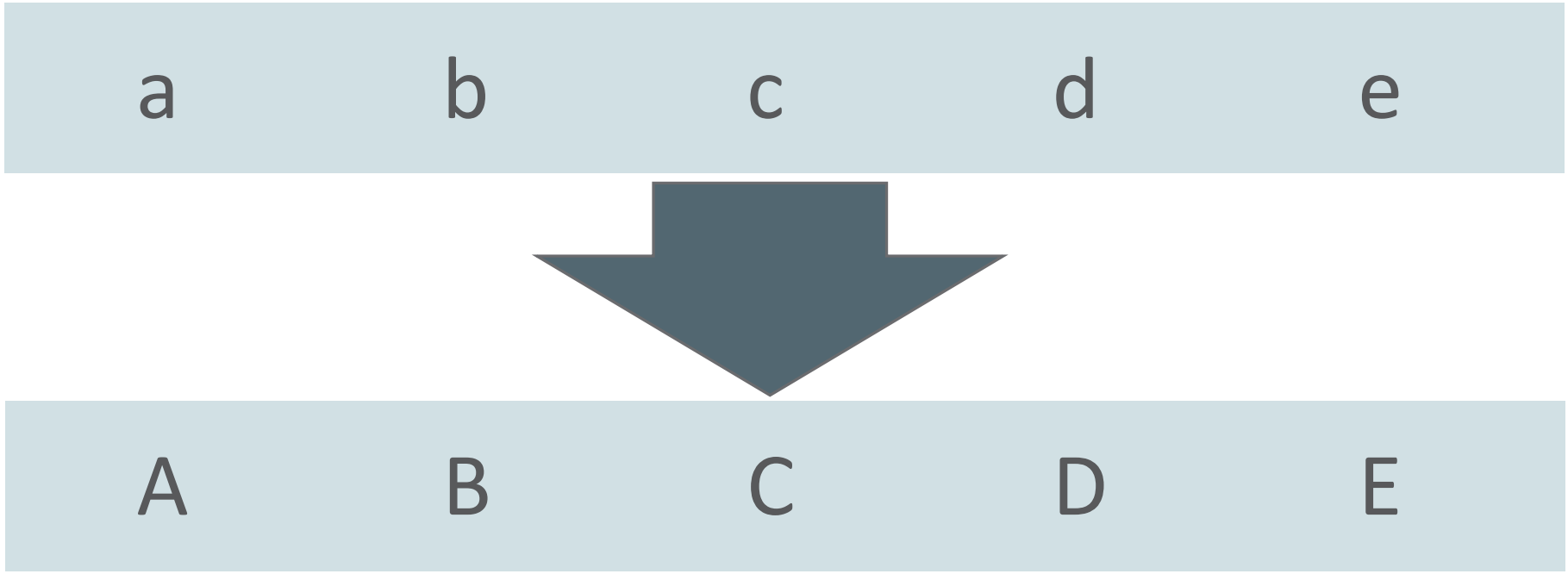
Convert Array of Strings to Upper Case – Streams

```
String[] upcase(String[] input) {
    return Arrays.stream(input)
                 .map(String::toUpperCase)
                 .toArray(String[]::new);
}
```

Can be done in any order, or all at once!



Consider as aggregate operation, not individual operations



Which is Better?

- Streams version is better
  - more compact
  - new and cool
  - more functional
- For-loop is better
  - more efficient
  - more familiar
  - more straightforward

Which is Better?

- Streams version is better
  - more compact
  - new and cool
  - more functional
- For-loop is better
  - more efficient
  - more familiar
  - more straightforward

WRONG!

Verdict: Streams Version is Better

- Why? Higher level of abstraction
  - Expresses independence of each computation
  - Less accidental complexity, e.g., index computations
  - Implicitly operates on all elements, not individual elements
  - Focus on desired results than mechanics of computing it
- Why hasn't he said anything about parallelism yet?
  - Making it parallelizable isn't what makes it better
    - you might never want to run this code in parallel (see Brian's part)
    - but this code is still better
  - Making the code better makes it parallelizable, as a bonus

A Less Trivial Example

- Splitting a list
  - Split at elements selected by a predicate
  - Result list should be sublists of original list (using List.subList method)
- Example
  - Split the list: `[a, b, #, c, #, d, e]`
  - At elements that equal "#"
  - Expected result: `[[a, b], [c], [d, e]]`
- Based on Stack Overflow question:
  - http://stackoverflow.com/a/29111023/1441122

Thinking Through the Problem

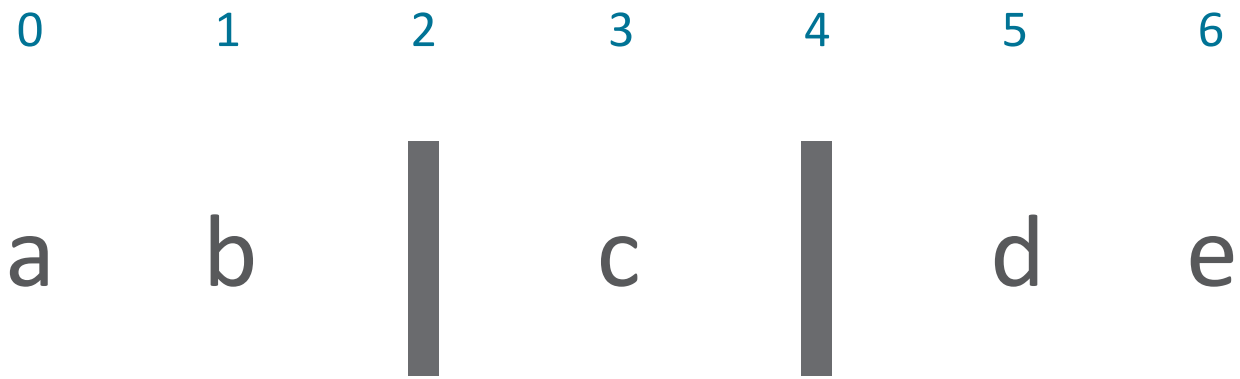
a b # c # d e

Thinking Through the Problem

0	1	2	3	4	5	6
a	b	#	c	#	d	e

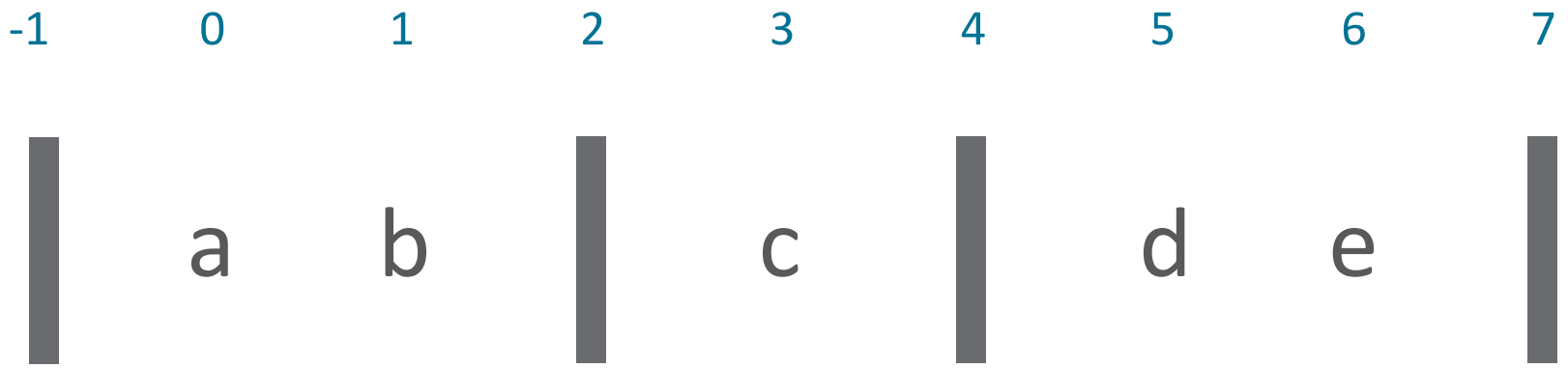
List.subList() wants indexes, so conceptually, number all the elements

Thinking Through the Problem



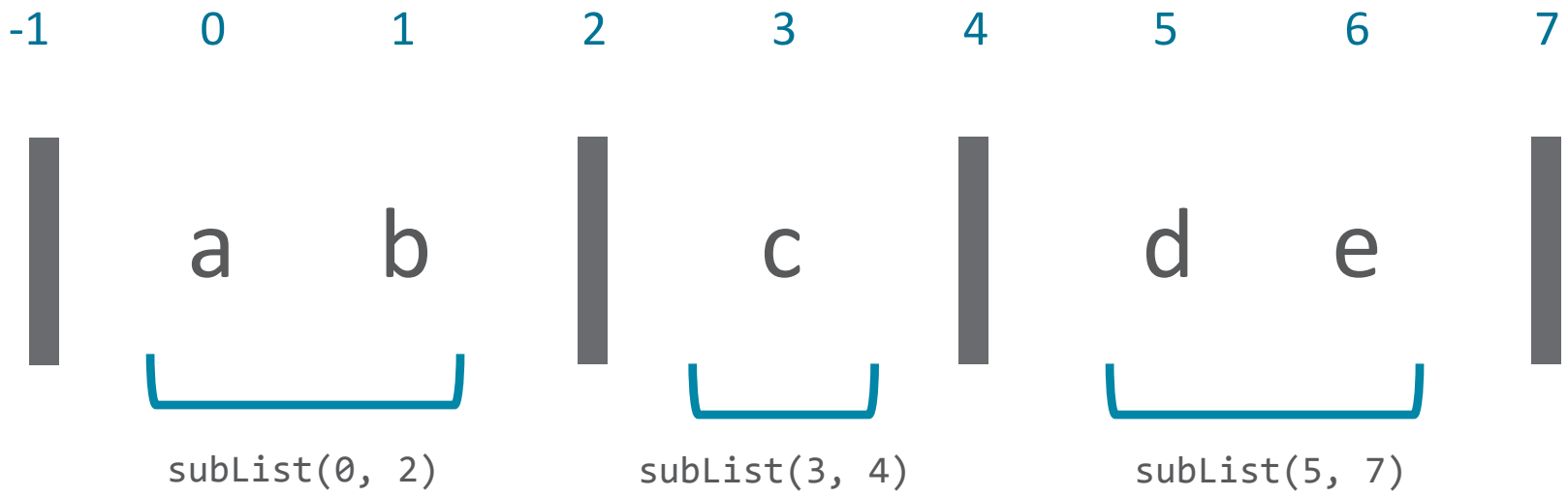
split points are the edges of the sublists

Thinking Through the Problem



we also need to "synthesize" split points at each end to create bounds for exterior sublists

Thinking Through the Problem



*each subList starts at **this** split point +1*
*and runs until the **next** split point*
(since subList is half-open on the right)
implicitly handles corner cases correctly

Start with this method signature

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {

}
```

Conventional Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    List<List<T>> result = new ArrayList<>();
    int start = 0;

    for (int cur = 0; cur < input.size(); cur++) {
        if (pred.test(input.get(cur))) {
            result.add(input.subList(start, cur));
            start = cur + 1;
        }
    }
    result.add(input.subList(start, input.size()));

    return result;
}
```

Conventional Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    List<List<T>> result = new ArrayList<>();
    int start = 0;

    for (int cur = 0; cur < input.size(); cur++) {
        if (pred.test(input.get(cur))) {
            result.add(input.subList(start, cur));
            start = cur + 1;
        }
    }
    result.add(input.subList(start, input.size()));

    return result;
}
```

apply predicate to each element

sublist creation

Conventional Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    List<List<T>> result = new ArrayList<>();
    int start = 0;

    for (int cur = 0; cur < input.size(); cur++) {
        if (pred.test(input.get(cur))) {
            result.add(input.subList(start, cur));
            start = cur + 1;
        }
    }
    result.add(input.subList(start, input.size()));

    return result;
}
```

initialization of result and state

exposed loop mechanics

why do we have to add one here?

extra addition to result list??

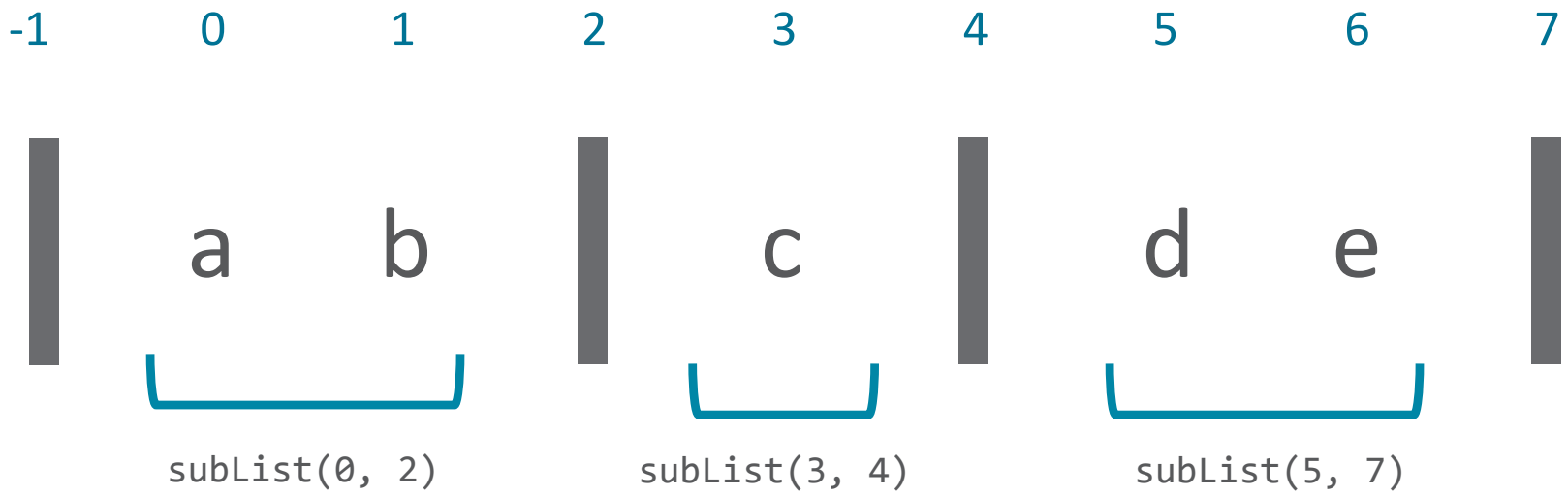
Conventional Approach – Observations

- How do we know this is correct?
  - Does this handle all the corner cases?
  - Handling of trailing sublist is treated non-uniformly
  - Reasoning about loop invariants is subtle
- Hard to see relationship between this code and the model we developed
  - Can you reverse-engineer the diagram from the code?
- Accidental data dependency
  - Each loop iteration depends on state from the previous iteration
  - ***Even though computations of split points are all independent!***

Streams Approach

- Rethink the problem, avoiding iteration
- We are interested in indexes (because of List.subList)
  - Stream over indexes instead, e.g., `IntStream.range(0, last)`
  - Instead of typical stream over values, e.g., `input.stream()`
- Computation of an edge is independent of all other computations
  - Apply the predicate to each element
  - No dependencies on result of predicate on any other result
  - Contrast with the preceding looping approach

Streams Approach



*each subList starts at **this** split point +1*
*and runs until the **next** split point*
(since subList is half-open on the right)
implicitly handles corner cases correctly

Streams Approach – Outline

1. Filter indexes to find interior sublist edges
2. Synthesize exterior edges at each end
3. Compute sublist from ***this*** edge and the ***next*** one

Streams Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    int[] temp  = IntStream.range(0, input.size())
                           .filter(i -> pred.test(input.get(i)))
                           .toArray();
```

compute interior edges by applying the predicate to every element

Streams Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    int[] temp  = IntStream.range(0, input.size())
                           .filter(i -> pred.test(input.get(i)))
                           .toArray();

    int[] edges = new int[temp.length+2];
    System.arraycopy(temp, 0, edges, 1, temp.length);
    edges[0] = -1;
    edges[edges.length-1] = input.size();
```

Synthesize exterior edges.
OK, we admit it, these array operations are ugly.

Streams Approach

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    int[] temp  = IntStream.range(0, input.size())
                           .filter(i -> pred.test(input.get(i)))
                           .toArray();

    int[] edges = new int[temp.length+2];
    System.arraycopy(temp, 0, edges, 1, temp.length);
    edges[0] = -1;
    edges[edges.length-1] = input.size();

    return IntStream.range(0, edges.length-1)
                    .mapToObj(k -> input.subList(edges[k]+1, edges[k+1]))
                    .collect(toList());
}
```

compute sublists from each edge and the one to its right

Quick Aside: Adjust Bounds to Synthesize Exterior Edges

```
<T> List<List<T>> split(List<T> input, Predicate<T> pred) {
    int[] edges = IntStream.range(-1, input.size()+1)
                           .filter(i -> i == -1 || i == input.size() ||
                                   pred.test(input.get(i)))
                           .toArray();

    return IntStream.range(0, edges.length-1)
                    .mapToObj(k -> input.subList(edges[k]+1, edges[k+1]))
                    .collect(toList());
}
```

Streams Approach – Observations

- Which is better?
- Streams code is better
  - Not because it's new, cool, shorter, more functional
  - But because it's at a higher level of abstraction
- Characteristics
  - Independent computations are independent
    - No accidental dependencies are introduced
  - Problem setup treats all cases uniformly
  - Operations on aggregates, not element-at-a-time
  - No loop mechanics to worry about

Streams Approach – Observations

- Useful technique: stream over indexes, not over elements
  - Many, but not all, problems can be solved by streaming over elements
  - If you're fighting the Streams API, try this, it might work
  - Broadly, but not universally applicable
- By the way, you can also run this in parallel!
- But, ***should*** you run it in parallel?

Now, over to Brian...

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?

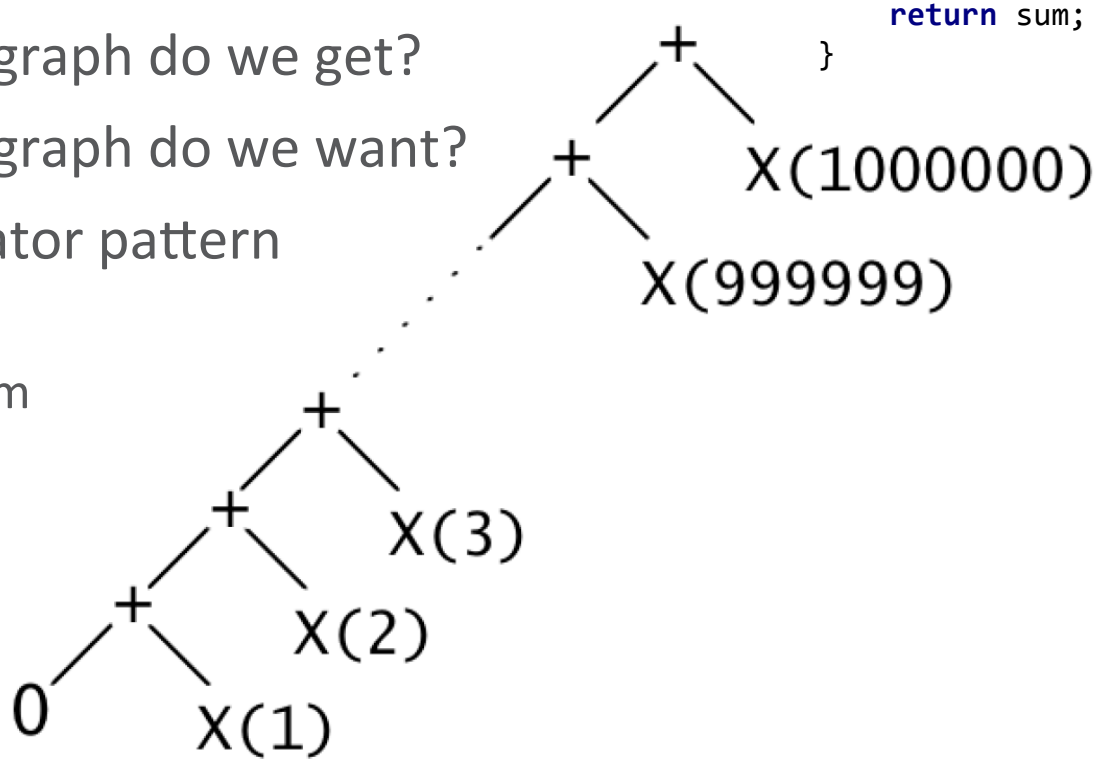
Parallelism

- 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

Towards Parallel Computation

- 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

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



)00000)

Divide And Conquer

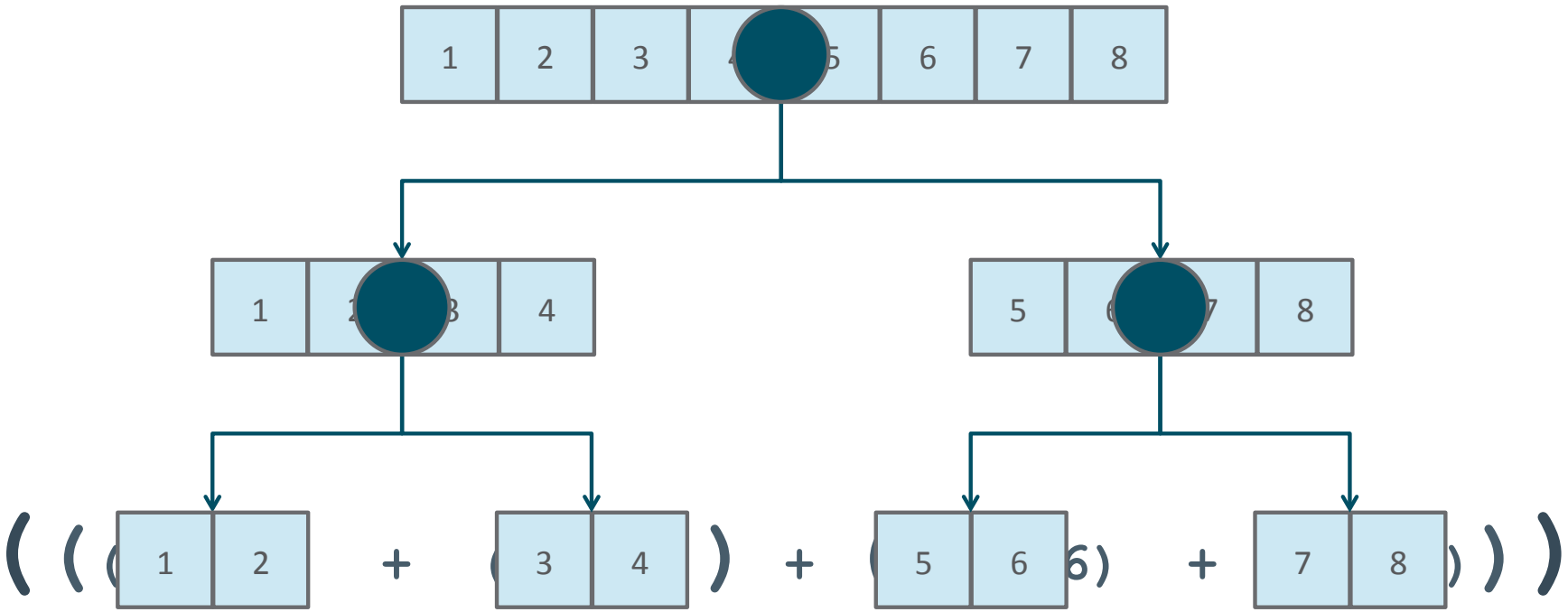
- Standard tool for parallel execution is *divide-and-conquer*
  - Partition the input into chunks that can be independently operated on
  - Recursively decompose problem until it is small enough for sequential

```
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);
}
```

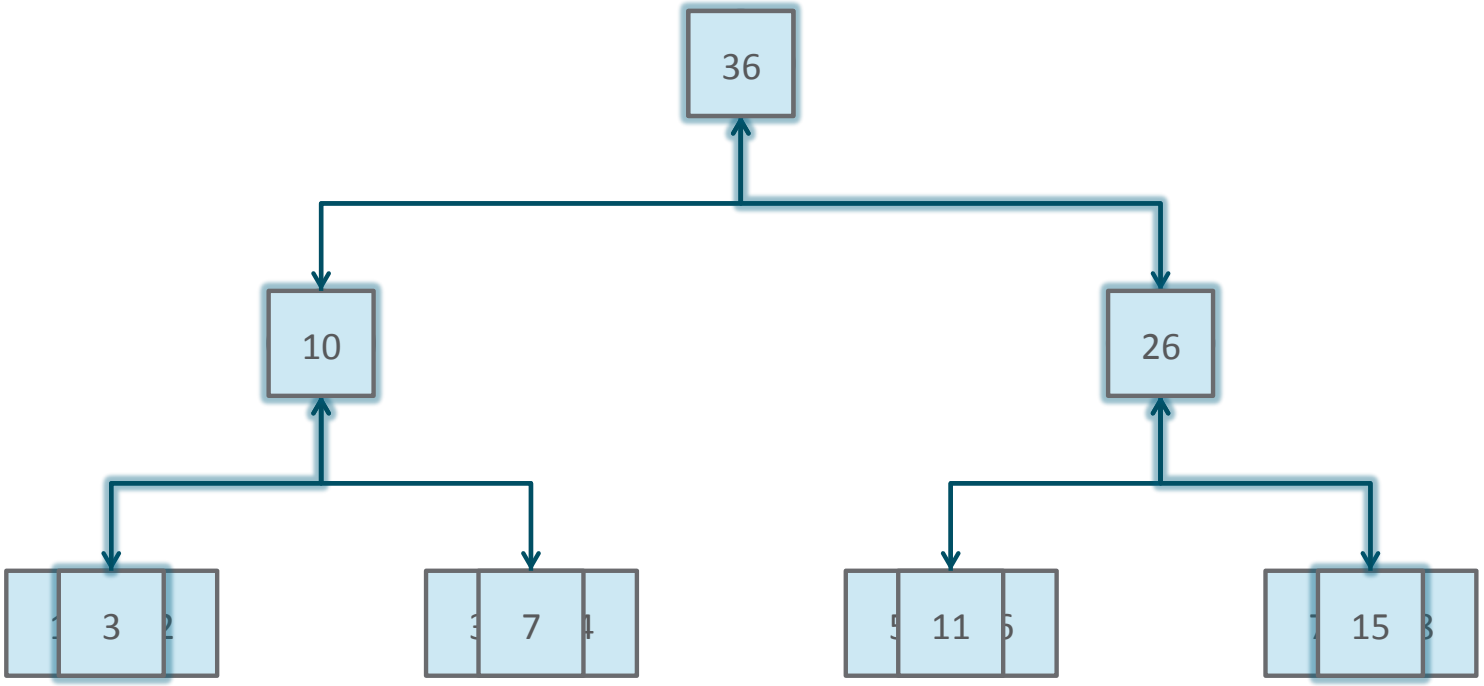
Divide And Conquer

- 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

Parallel Stream Performance

- Streams is about *possibly-parallel*, *aggregate operations* on datasets
  - Streams are efficient, and (usually) merge computation into a single pass
  - But they are NOT magic parallelism dust!
- Still have to ensure that our problem is amenable to parallel solution
  - How easily splittable is the source?
  - How expensive is result combination?
    - Adding numbers is cheap; merging sets is expensive
  - What kind of locality does our computation get?
    - 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
- Most simple stream examples have very low Q
  - Meaning everything else has to go well to get a speedup

Source Splitting

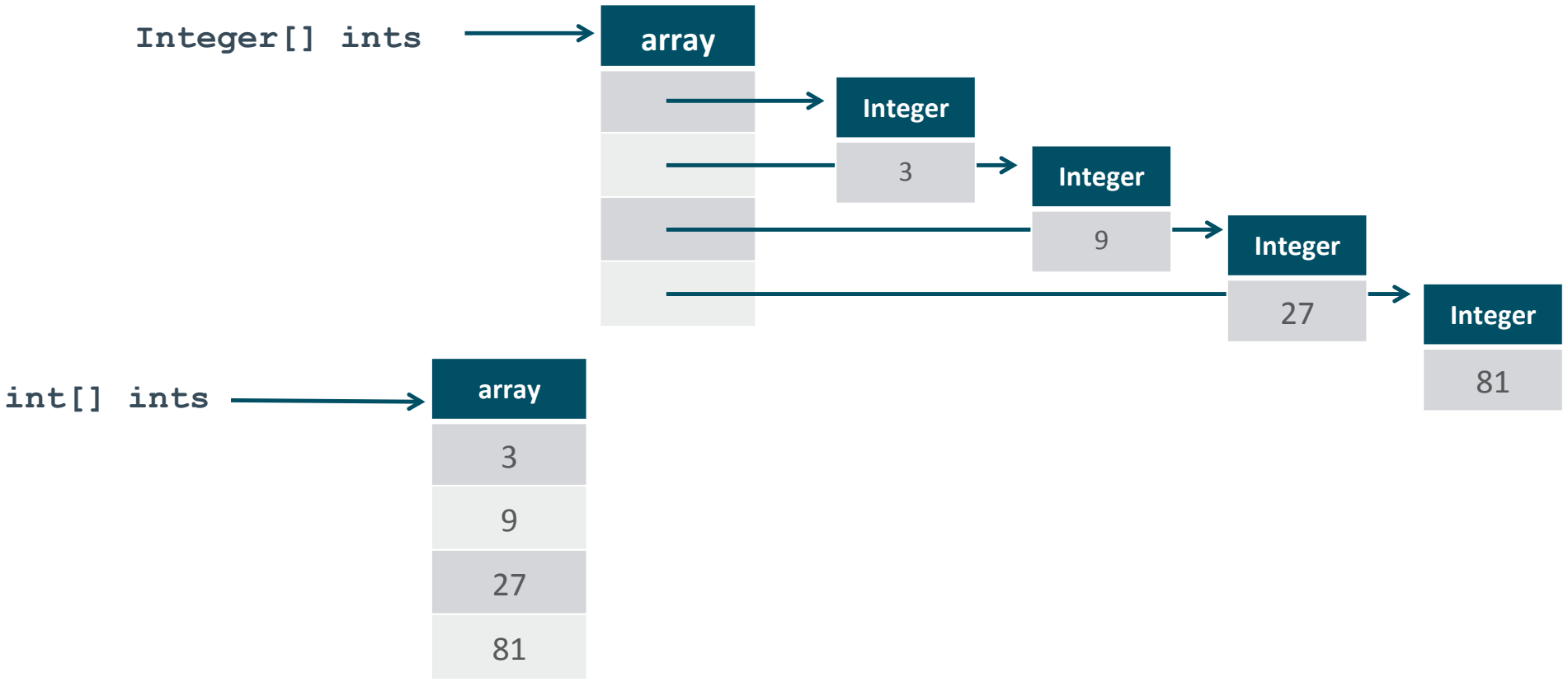
- Some sources split better than others
  - Cost of computing split
  - Evenness of split
  - Predictability of split
- Arrays split cheaply, evenly, and with perfect knowledge of split sizes
  - Linked lists have none of these properties
  - Iterative generators behave like linked lists, stateless generators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

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



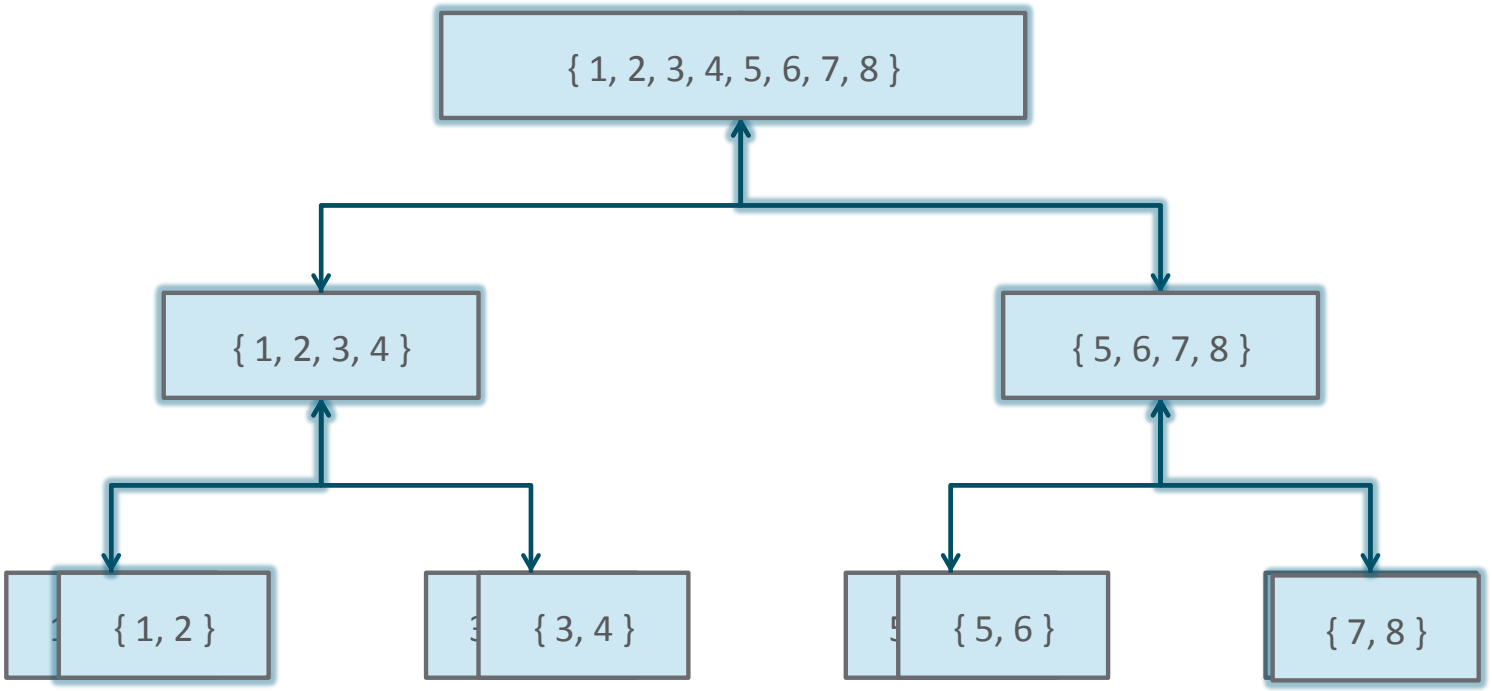
Encounter Order

- 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

Safe Harbor Statement

The preceding 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.

JavaOne™

ORACLE®


ORACLE®