17

Parallel Streams

Objectives

After completing this lesson, you should be able to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

Streams Review

- Pipeline
  - Multiple streams passing data along
  - Operations can be Lazy
  - Intermediate, Terminal, and Short-Circuit Terminal Operations
- Stream characteristics
  - Immutable
  - Once elements are consumed they are no longer available from the stream.
  - Can be sequential (default) or **parallel**

Old Style Collection Processing

```
double sum = 0;

for(Employee e:eList){
    if(e.getState().equals("CO") &&
        e.getRole().equals(Role.EXECUTIVE)){
        e.printSummary();
        sum += e.getSalary();
    }
}

System.out.printf("Total CO Executive Pay: $%,9.2f %n", sum);
```

New Style Collection Processing

```
        double result = eList.stream()
            .filter(e -> e.getState().equals("CO"))
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .peek(e -> e.printSummary())
            .mapToDouble(e -> e.getSalary())
            .sum();

        System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
```

- What are the advantages?
  - Code reads like a problem.
  - Acts on the data set
  - Operations can be lazy.
  - Operations can be serial or parallel.

Stream Pipeline: Another Look

```
    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        Stream<Employee> s1 = eList.stream();

        Stream<Employee> s2 = s1.filter(
            e -> e.getState().equals("CO"));

        Stream<Employee> s3 = s2.filter(
            e -> e.getRole().equals(Role.EXECUTIVE));
        Stream<Employee> s4 = s3.peek(e -> e.printSummary());
        DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
        double result = s5.sum();

        System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
    }
```

Styles Compared

Imperative Programming

- Code deals with individual data items.
- Focused on how
- Code does not read like a problem.
- Steps mashed together
- Leaks extraneous details
- Inherently sequential

Streams

- Code deals with data set.
- Focused on what
- Code reads like a problem.
- Well-factored
- No "garbage variables" (Temp variables leaked into scope)
- Code can be sequential or parallel.

Parallel Stream

- May provide better performance
  - Many chips and cores per machine
  - GPUs
- Map/Reduce in the small
- Fork/join is great, but too low level
  - A lot of boilerplate code
  - Stream uses fork/join under the hood
- Many factors affect performance
  - Data size, decomposition, packing, number of cores
- Unfortunately, not a magic bullet
  - Parallel is not always faster

Using Parallel Streams: Collection

- Call from a Collection

```
        double result = eList.parallelStream()
                .filter(e -> e.getState().equals("CO"))
                .filter(e ->
e.getRole().equals(Role.EXECUTIVE))
                .peek(e -> e.printSummary())
                .mapToDouble(e -> e.getSalary())
                .sum();

        System.out.printf("Total CO Executive Pay:
$%,9.2f %n", result);
```

Using Parallel Streams: From a Stream

```
        result = eList.stream()
            .filter(e -> e.getState().equals("CO"))
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .peek(e -> e.printSummary())
            .mapToDouble(e -> e.getSalary())
            .parallel()
            .sum();

        System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

- Specify with `.parallel` or `.sequential` (default is sequential)
- Choice applies to entire pipeline.
  - Last call wins
- Once again, the API doc is your friend.

Pipelines Fine Print

- Stream pipelines are like Builders.
  - Add a bunch of intermediate operations, and then execute
  - Cannot "branch" or "reuse" pipeline
- Do not modify the source during a query.
- Operation parameters must be stateless.
  - Do not access any state that might change.
  - **This enables correct operation sequentially or in parallel.**
- Best to banish side effects completely.

Embrace Statelessness

```
List<Employee> newList02 = new ArrayList<>();
…
  newList02 = eList.parallelStream() // Good Parallel
           .filter(e -> e.getDept().equals("Eng"))
           .collect(Collectors.toList());
```

- Mutate the stateless way
  - The above is preferable.
  - It is designed to parallelize.

Avoid Statefulness

```
List<Employee> eList = Employee.createShortList();
List<Employee> newList01 = new ArrayList<>();
List<Employee> newList02 = new ArrayList<>();

eList.parallelStream() // Not Parallel. Bad.
    .filter(e -> e.getDept().equals("Eng"))
    .forEach(e -> newList01.add(e));
```

- Temptation is to do the above.
  - **Do not do this. It does not parallelize.**

Streams Are Deterministic for Most Part

```
        List<Employee> eList = Employee.createShortList();

        double r1 = eList.stream()
            .filter(e -> e.getState().equals("CO"))
            .mapToDouble(Employee::getSalary)
            .sequential().sum();

        double r2 = eList.stream()
            .filter(e -> e.getState().equals("CO"))
            .mapToDouble(Employee::getSalary)
            .parallel().sum();

   System.out.println("The same: " + (r1 == r2));
```

- Will the result be the same?

Some Are Not Deterministic

```
        List<Employee> eList = Employee.createShortList();

        Optional<Employee> e1 = eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .sequential().findAny();

        Optional<Employee> e2 = eList.stream()
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .parallel().findAny();

        System.out.println("The same: " +
            e1.get().getEmail().equals(e2.get().getEmail()));
```

- Will the result be the same?
  - In this case, maybe not.

Reduction

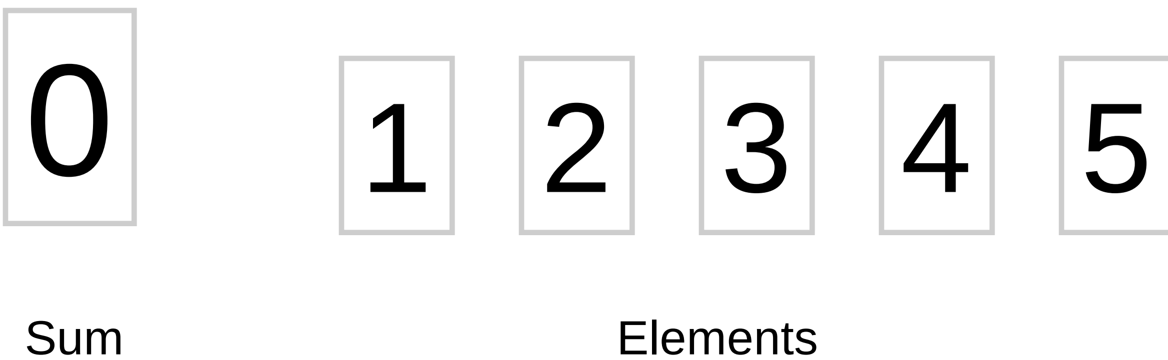
- Reduction
  - An operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
  - Implemented with the `reduce()` method
- Example: `sum` is a reduction with a base value of 0 and a combining function of `+`.
  - $((((0 + a_1) + a_2) + ...) + a_n)$
  - `.sum()` is equivalent to reduce `(0, (a, b) -> a +b )`
  - `(0, (sum, element) -> sum + element)`

Reduction Fine Print

- If the combining function is associative, reduction parallelizes cleanly
  - Associative means the order does not matter.
  - The result is the same irrespective of the order used to combine elements.
- Examples of: sum, min, max, average, count
  - `.count()` is equivalent to `.map(e -> 1).sum()`.
- **Warning:** If you pass a nonassociative function to `reduce`, you will get the wrong answer. The function must be associative.

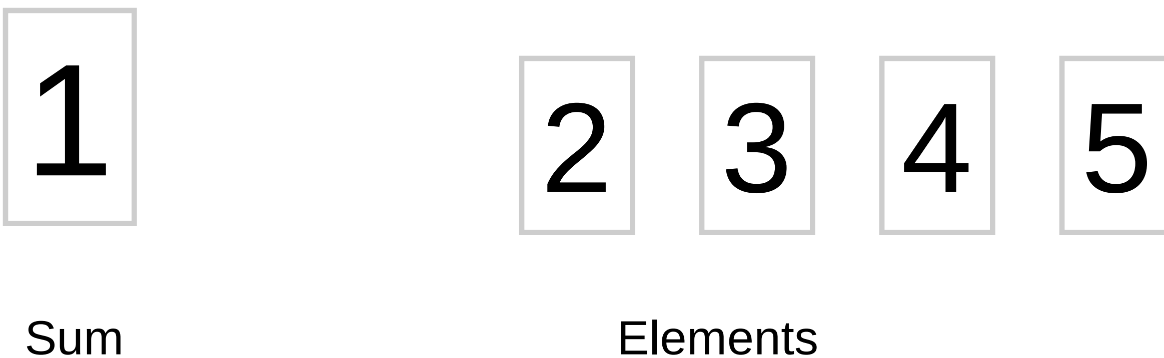
Reduction: Example

```
int r2 = IntStream.rangeClosed(1, 5).parallel()
    .reduce(0, (sum, element) -> sum + element);

System.out.println("Result: " + r2);
```



Reduction: Example

```
int r2 = IntStream.rangeClosed(1, 5).parallel()
    .reduce(0, (sum, element) -> sum + element);

System.out.println("Result: " + r2);
```



Reduction: Example

```
        int r2 = IntStream.rangeClosed(1, 5).parallel()
            .reduce(0, (sum, element) -> sum + element);

        System.out.println("Result: " + r2);
```

3

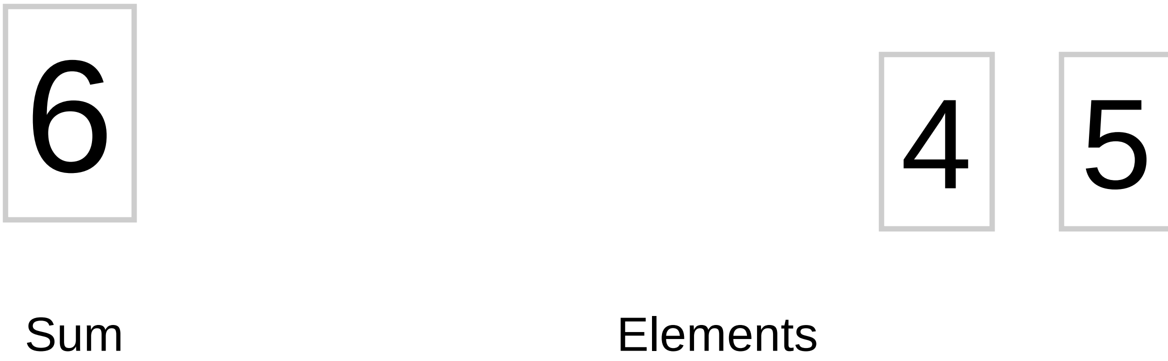
3 4 5

Sum

Elements

Reduction: Example

```
int r2 = IntStream.rangeClosed(1, 5).parallel()
    .reduce(0, (sum, element) -> sum + element);

System.out.println("Result: " + r2);
```



Reduction: Example

```
        int r2 = IntStream.rangeClosed(1, 5).parallel()
            .reduce(0, (sum, element) -> sum + element);

        System.out.println("Result: " + r2);
```

10

5

Sum

Elements

Reduction: Example

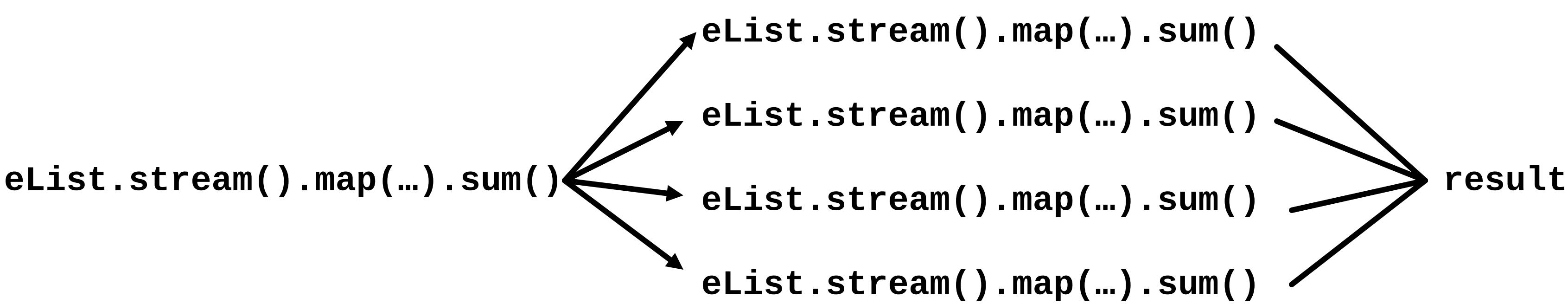
```
int r2 = IntStream.rangeClosed(1, 5).parallel()
    .reduce(0, (sum, element) -> sum + element);

System.out.println("Result: " + r2);
```

15

Sum Elements

A Look Under the Hood

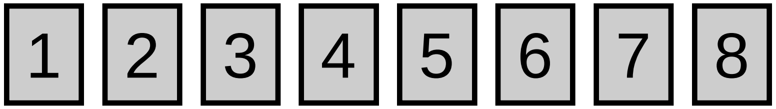
- Pipeline decomposed into subpipelines.
  - Each subpipeline produces a subresult.
  - Subresults combined into final result.



ORACLE

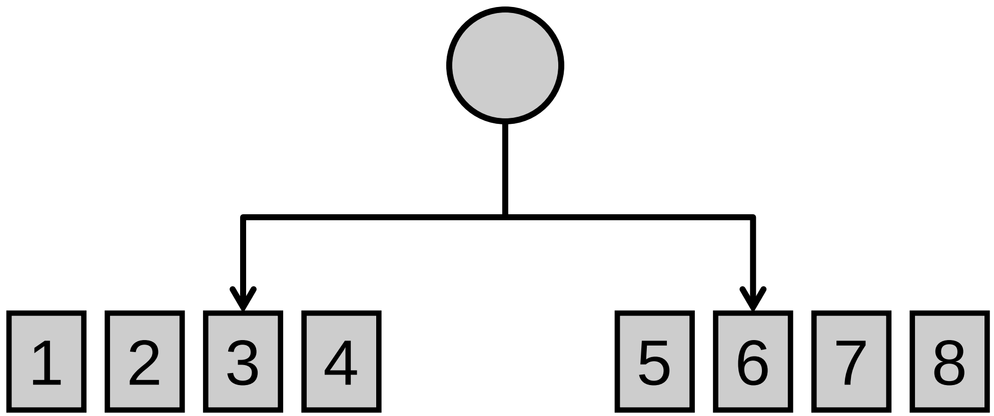
Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



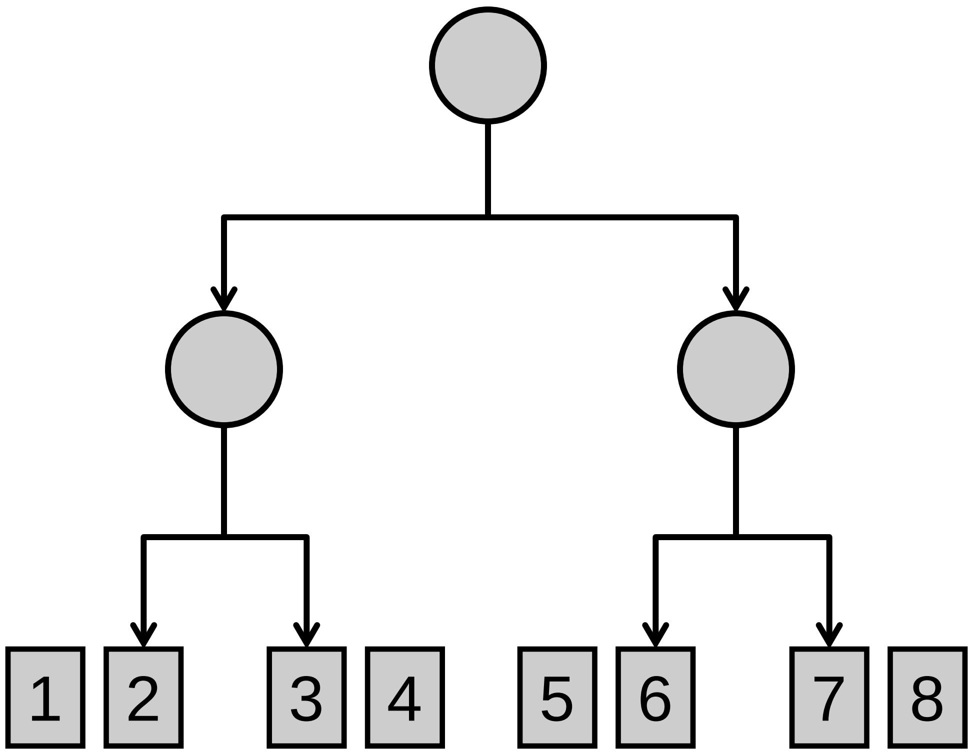
Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



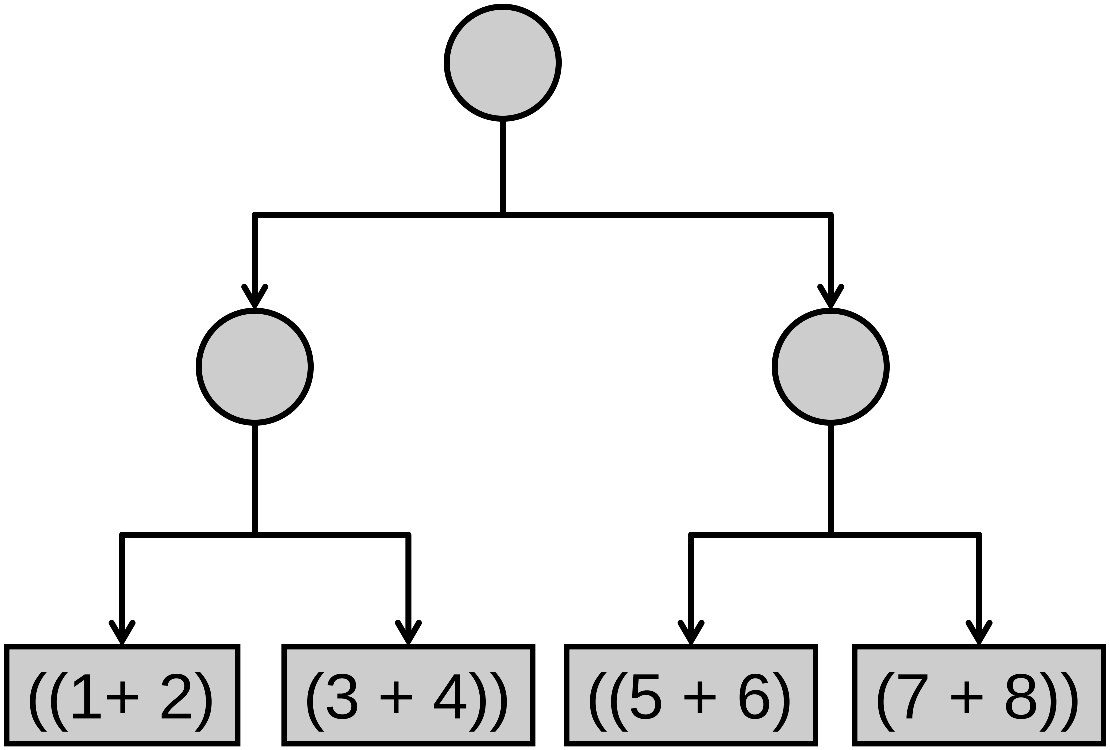
Illustrating Parallel Execution

```
int r2 = IntStream.rangeClosed(1, 8).parallel()
        .reduce(0, (sum, element) -> sum + element);
```



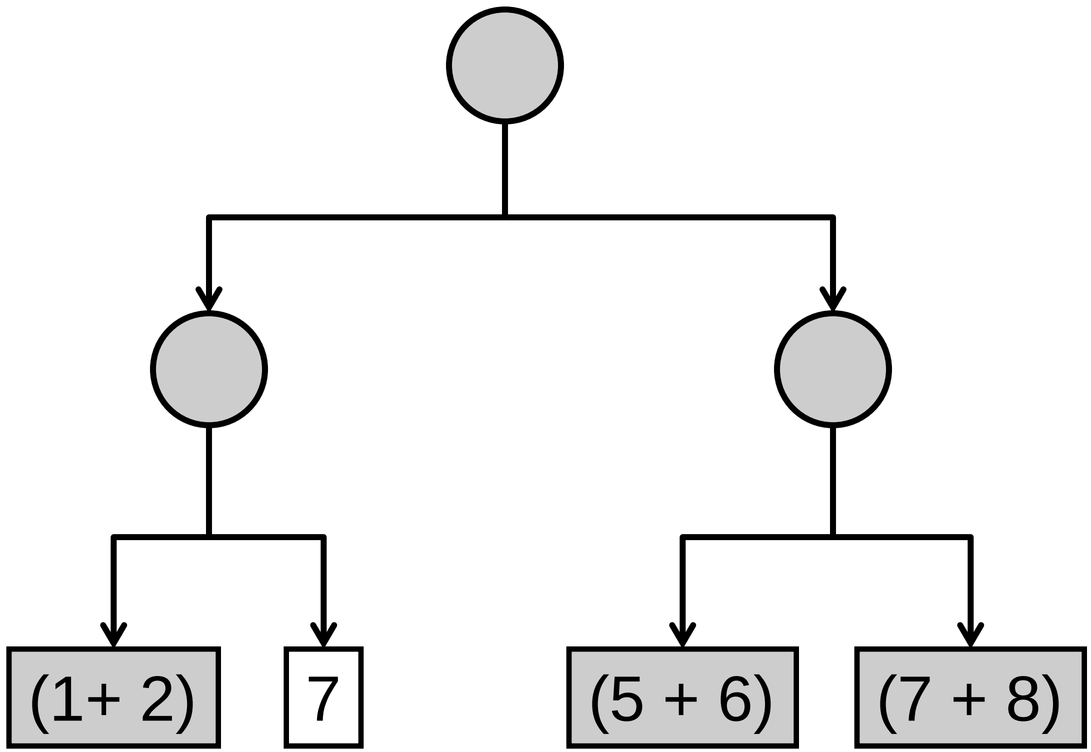
Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



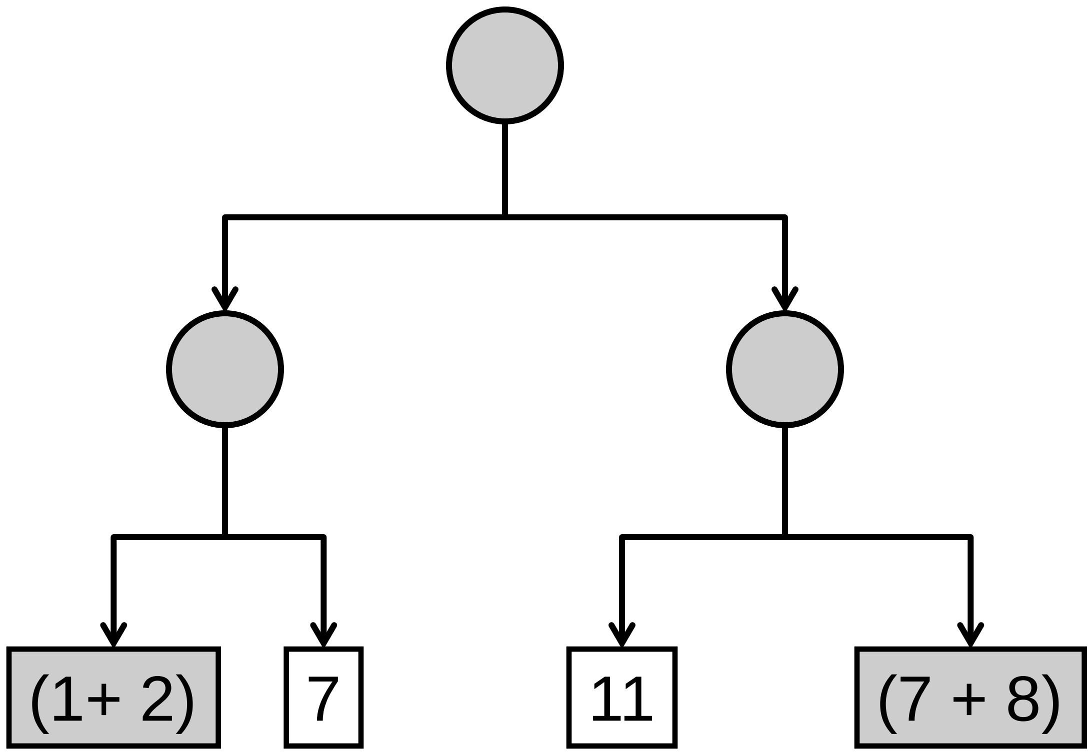
Illustrating Parallel Execution

```
int r2 = IntStream.rangeClosed(1, 8).parallel()
    .reduce(0, (sum, element) -> sum + element);
```



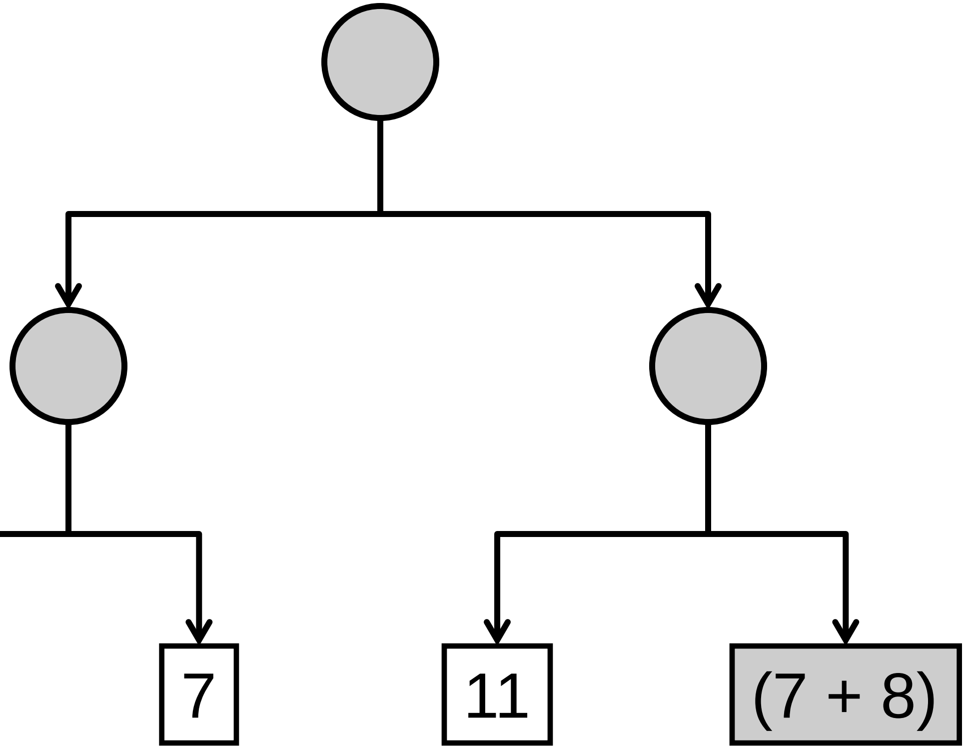
Illustrating Parallel Execution

```
int r2 = IntStream.rangeClosed(1, 8).parallel()
         .reduce(0, (sum, element) -> sum + element);
```



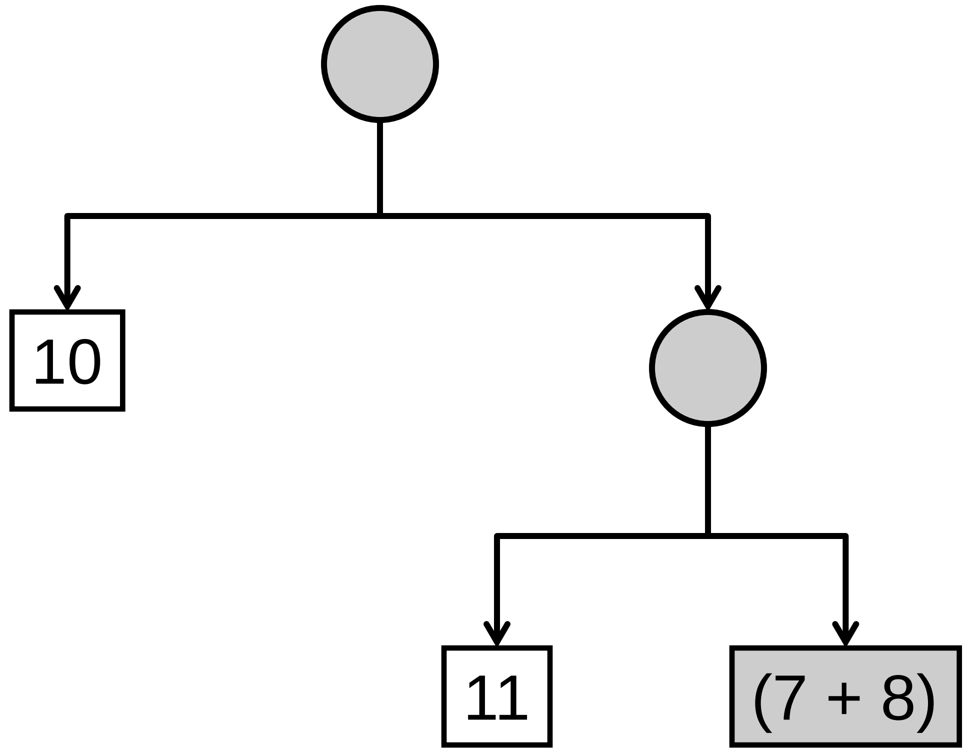
Illustrating Parallel Execution

```
int r2 = IntStream.rangeClosed(1, 8).parallel()
    .reduce(0, (sum, element) -> sum + element);
```



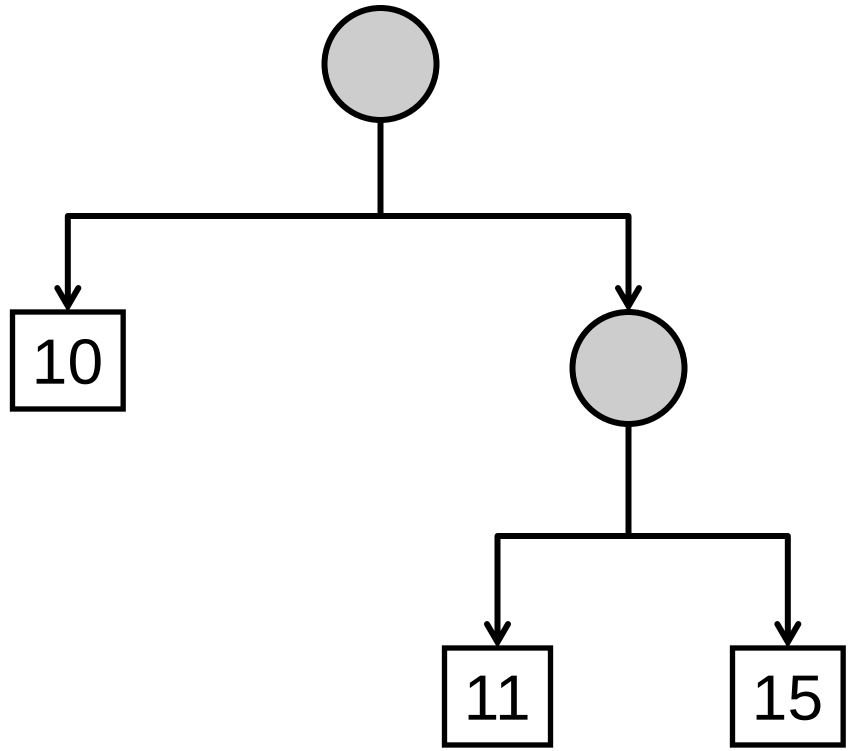
Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



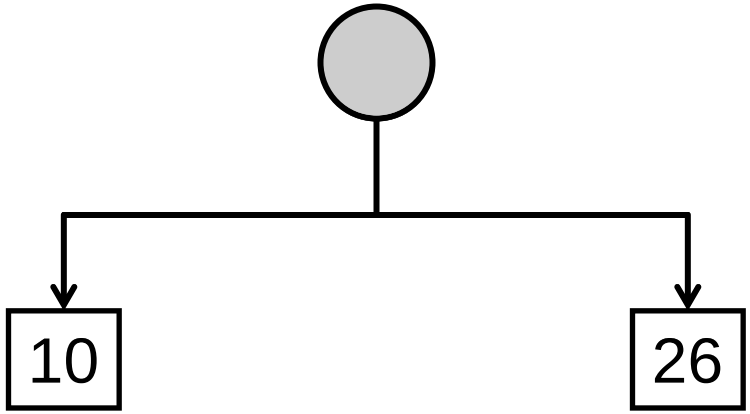
Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
        int r2 = IntStream.rangeClosed(1, 8).parallel()
            .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
int r2 = IntStream.rangeClosed(1, 8).parallel()
    .reduce(0, (sum, element) -> sum + element);
```

36

Performance

- Do not assume parallel is always faster.
  - Parallel not always the right solution.
  - Sometimes parallel is slower than sequential.
- Qualitative considerations
  - Does the stream source decompose well?
  - Do terminal operations have a cheap or expensive merge operation?
  - What are stream characteristics?
    - Filters change size for example.
- Primitive streams provided for performance
  - Boxing/Unboxing negatively impacts performance.

A Simple Performance Model

N = Size of the source data set

Q = Cost per element through the pipeline

N * Q ~= Cost of the pipeline

- Larger N*Q -> Higher change of good parallel performance
- Easier to know N than Q
- You can reason qualitatively about Q
  - Simple pipeline example
    - N > 10K. Q=1
    - Reduction using sum
  - Complex pipelines might
    - Contain filters
    - Contain limit operation
    - Complex reduction using `groupingBy()`

Summary

In this lesson, you should have learned how to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

Practice

- Practice 17-1: Calculate Total Sales Without a Pipeline
- Practice 17-2: Calculate Sales Totals Using Parallel Streams
- Practice 17-3: Calculate Sales Totals Using Parallel Streams and Reduce