Optimizing Parallel Java Streams

Matteo Basso, **Filippo Schiavio, Andrea Rosà, Walter Binder**

Università della Svizzera italiana, Switzerland

Università
della
Svizzera
italiana

ICECCS 2022
March 29, 2022

Introduction

- The Java Stream API allows manipulating elements via a pipeline of operations
  - Stream creation
  - Intermediate operations
  - Terminal operations
- Facilitates parallelization
  - Automatically splits computation using `ForkJoinTask`

Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

Sequential ordered version taken from: A. Biboudis et al., "Clash of the Lambdas".

Problem Statement

- The Stream API leads to performance degradation [1], [2], [3]
  - Object allocation
  - Prevents Just-In-Time (JIT) compiler optimizations

[1] A. Biboudis et al., "Clash of the Lambdas".

[2] R. Khatchadourian et al., "An Empirical Study on the Use and Misuse of Java 8 Streams". FASE 2020.

[3] O. Kiselyov et al., "Stream Fusion, to Completeness". POPL 2017.

Related Work

- Transformation of streams into loops [1], [2], [3]
  - Significant performance improvements
  - Optimize only sequential streams
  - Parallel streams are ignored

[1] A. Møller et al., "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization". OOPSLA 2020.

[2] F. Ribeiro et al., "Java Stream Fusion: Adapting FP Mechanisms for an OO Setting". SBLP 2019.

[3] O. Kiselyov et al., "Stream Fusion, to Completeness". POPL 2017.

Goal

- Transforming parallel unordered Java streams to loops executed by `ForkJoinTasks`
  - Significant speedups opportunities
  - The usage of parallel unordered streams is encouraged
  - Related work [1], [2] expand the applicability of our technique

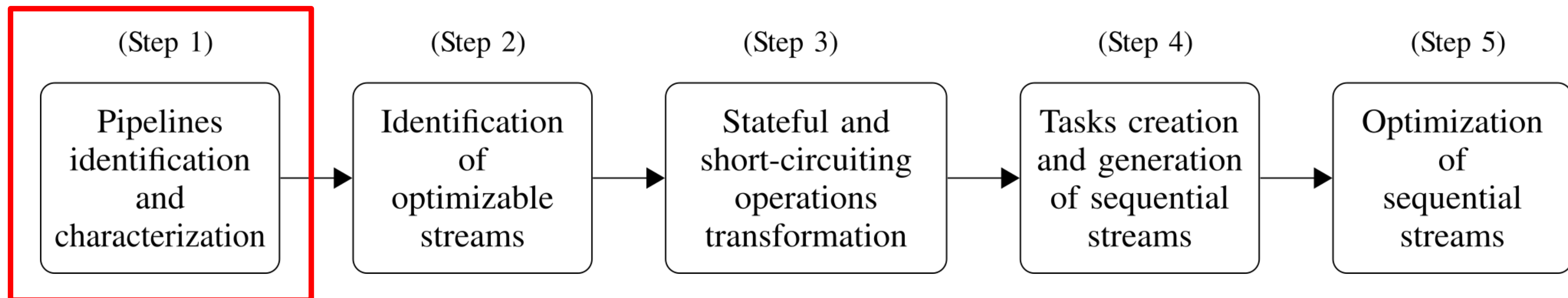
[1] R. Khatchadourian et al., "Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams". Science of Computer Programming 2020.

[2] R. Khatchadourian et al., "A Tool for Optimizing Java 8 Stream Software via Automated Refactoring". SCAM 2018.

Contributions

- Novel technique to translate parallel unordered streams into corresponding imperative code
- Several optimizations for the transformation of stateful and short-circuiting operations to reduce synchronization and contention
- Evaluation of our approach showing significant execution time and memory allocation improvements

Technique



- We use the approach proposed by Møller at al. [1] to identify bytecode instructions that belong to different pipelines
- Pipeline characterization
- Extraction of type information

[1] A. Møller et al., "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization". OOPSLA 2020.

Step 1 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

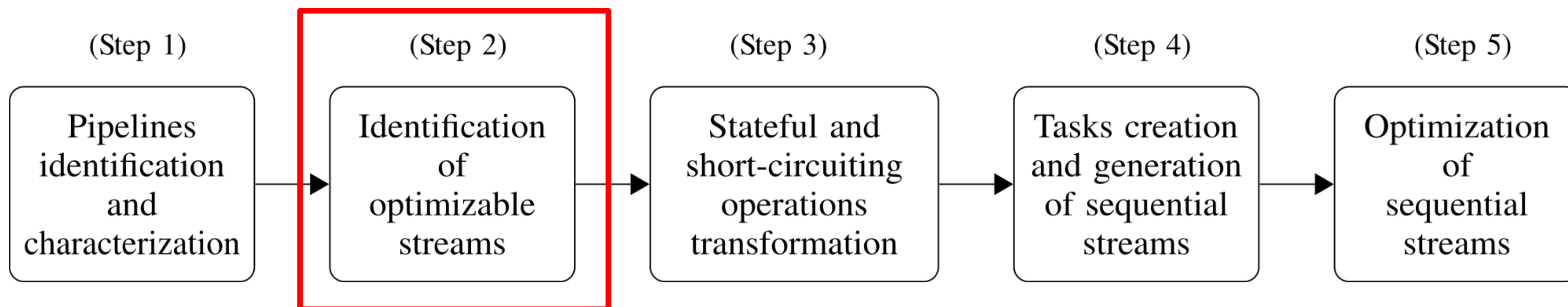
Sequential ordered version taken from: A. Biboudis et al., "Clash of the Lambdas".

Step 1 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

Sequential ordered version taken from: A. Biboudis et al., "Clash of the Lambdas".

Technique



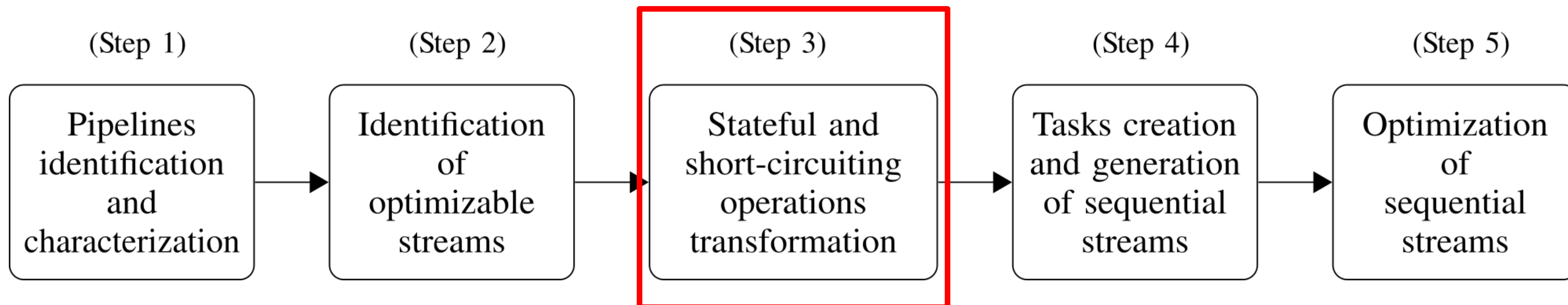
➢ Detection of parallel unordered streams, which can be optimized

Step 2 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

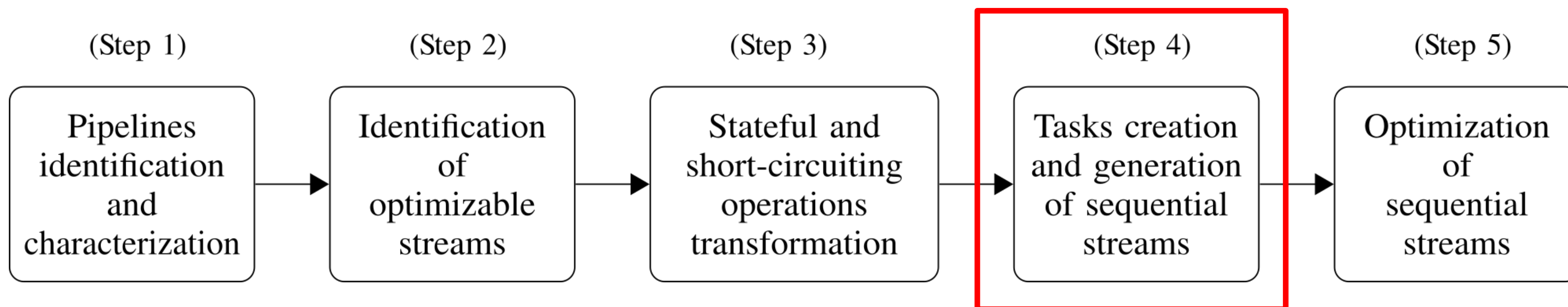
Sequential ordered version taken from: A. Biboudis et al., "Clash of the Lambdas".

Technique



- Stateful operations
  - Incorporate a state
- Short-circuiting operations
  - Finite streams from infinite data sources
  - Terminate in finite time from infinite input
- More detail later

Technique



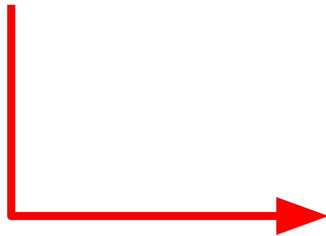
- Conversion of parallel streams into a sequential streams processed by tasks
  - Task creation
  - Pipeline moving
  - Task invocation

Step 4 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

Step 4 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```



```
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

Step 4 - Example

```
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

```
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

Step 4 - Example

```java
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

```java
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

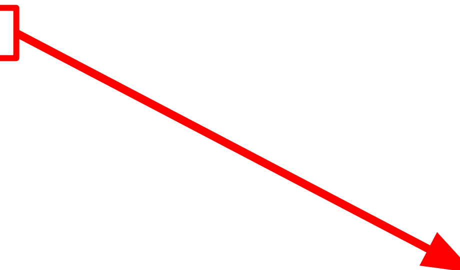
Step 4 - Example

```java
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

```java
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

Step 4 - Example

```java
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```



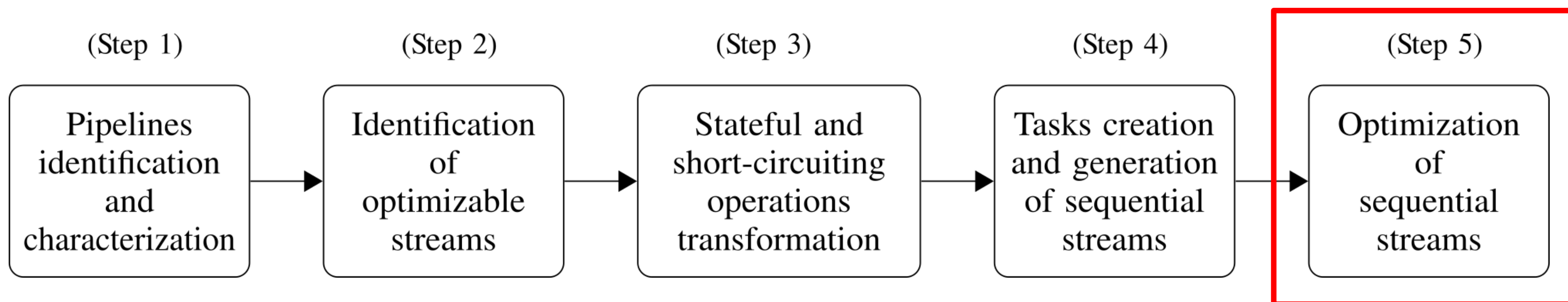
```java
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

Step 4 - Example

```java
int sumOfSquaresEven(int[] source) {
  return IntStream.of(source)
                  .unordered()
                  .parallel()
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * x)
                  .sum();
}
```

```java
int sumOfSquaresEven(int[] source) {
    class SumOfSquaresEvenTask extends ForkJoinTask<Integer> {
        // constructor, compute, and other methods are omitted
        int computePart(int low, int high) {
          return Arrays.stream(source, low, high)
                       .filter(x -> x % 2 == 0)
                       .map(x -> x * x)
                       .sum();
        }

        int merge(int left, int right) {
          return left + right;
        }
    }
    return new SumOfSquaresEvenTask().compute();
}
```

Technique



- Optimization of sequential streams using existing techniques
  - Single loop

Step 5 - Example

```
int computePart(int low, int high) {
    return Arrays.stream(source, low, high)
                .filter(x -> x % 2 == 0)
                .map(x -> x * x)
                .sum();
}
```

Step 5 - Example

```java
int computePart(int low, int high) {
    return Arrays.stream(source, low, high)
                 .filter(x -> x % 2 == 0)
                 .map(x -> x * x)
                 .sum();
}
```

```java
int computePart(int low, int high) {
    int sum = 0;
    for (int i = low; i < high; i++) {
        int x = source[i];
        if (x % 2 == 0) {
            sum += x * x;
        }
    }
    return sum;
}
```

Step 5 - Example

```
int computePart(int low, int high) {
    return Arrays.stream(source, low, high)
                 .filter(x -> x % 2 == 0)
                 .map(x -> x * x)
                 .sum();
}
```

```
int computePart(int low, int high) {
    int sum = 0;
    for (int i = low; i < high; i++) {
        int x = source[i];
        if (x % 2 == 0) {
            sum += x * x;
        }
    }
    return sum;
}
```

Step 5 - Example

```
int computePart(int low, int high) {
    return Arrays.stream(source, low, high)
                .filter(x -> x % 2 == 0)
                .map(x -> x * x)
                .sum();
}
```

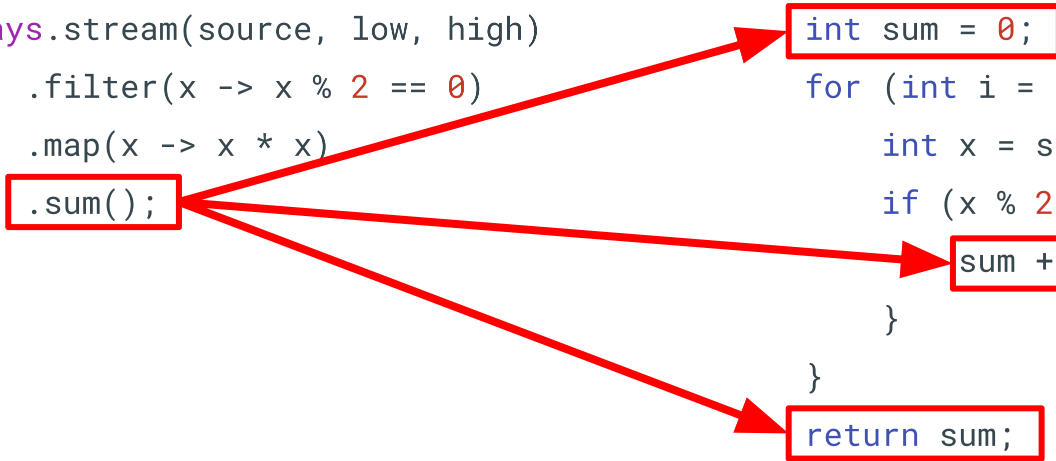
```
int computePart(int low, int high) {
    int sum = 0;
    for (int i = low; i < high; i++) {
        int x = source[i];
        if (x % 2 == 0) {
            sum += x * x;
        }
    }
    return sum;
}
```

Step 5 - Example

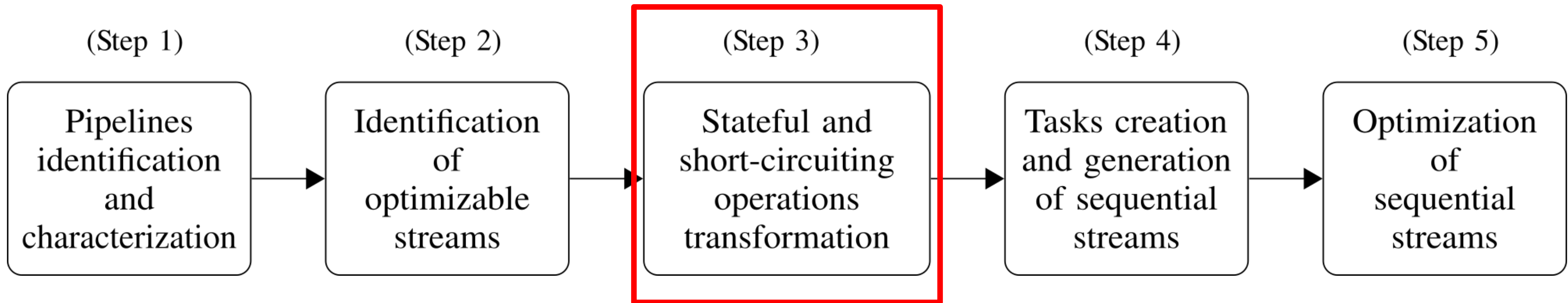
```java
int computePart(int low, int high) {
    return Arrays.stream(source, low, high)
                 .filter(x -> x % 2 == 0)
                 .map(x -> x * x)
                 .sum();
}
```

```java
int computePart(int low, int high) {
    int sum = 0;
    for (int i = low; i < high; i++) {
        int x = source[i];
        if (x % 2 == 0) {
            sum += x * x;
        }
    }
    return sum;
}
```

Step 5 - Example



Step 3 - Stateful Operations



- Stateful operations cannot be moved into tasks while ensuring correctness
  - A different state would be retained for each part of the data source, not for the entire stream

Stateful Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```

Stateful Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```

```
class DistinctSumTask extends ForkJoinTask<Integer> {
    // constructor, compute, and other methods are omitted
    int computePart(int low, int high) {
        return Arrays.stream(source, low, high)
                     .distinct()
                     .sum();
    }

    int merge(int left, int right) {
        return left + right;
    }
}
return new DistinctSumTask().compute();
```

Stateful Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```

```
class DistinctSumTask extends ForkJoinTask<Integer> {
    // constructor, compute, and other methods are omitted
    int computePart(int low, int high) {
        return Arrays.stream(source, low, high)
                    .distinct()
                    .sum();
    }

    int merge(int left, int right) {
        return left + right;
    }
}
return new DistinctSumTask().compute();
```

Stateful Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```

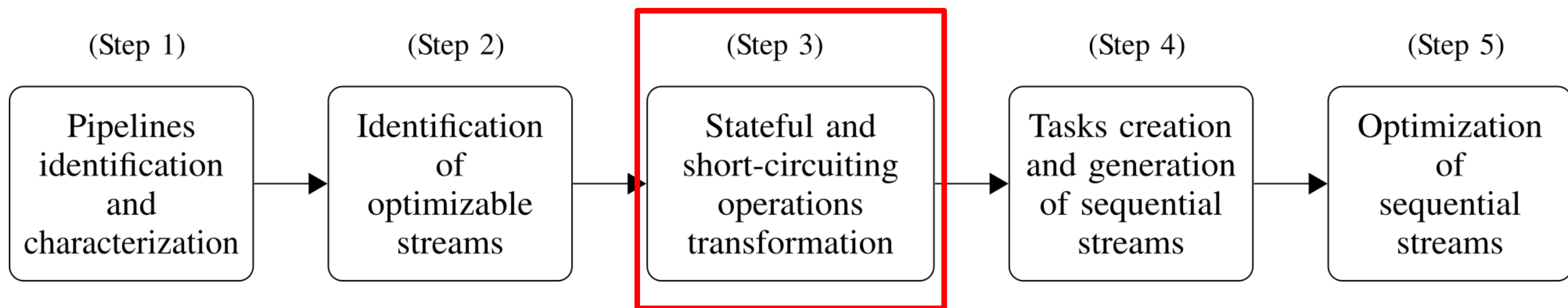
```
class DistinctSumTask extends ForkJoinTask<Integer> {
    // constructor, compute, and other methods are omitted
    int computePart(int low, int high) {
        return Arrays.stream(source, low, high)
                    .distinct()
                    .sum();
    }

    int merge(int left, int right) {
        return left + right;
    }
}
return new DistinctSumTask().compute();
```

Stateful Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```

```
class DistinctSumTask extends ForkJoinTask<Integer> {
    // constructor, compute, and other methods are omitted
    int computePart(int low, int high) {
        return Arrays.stream(source, low, high)
                     .distinct()
                     .sum();
    }

    int merge(int left, int right) {
        return left + right;
    }
}
return new DistinctSumTask().compute();
```

Stateful Operations



- Transform stateful operations to stateless filter operations
  - Predicates manage a shared synchronized state

Stateful Operations - Example

```
IntStream.of(source)
          .unordered()
          .parallel()
          .distinct()
          .sum();
```

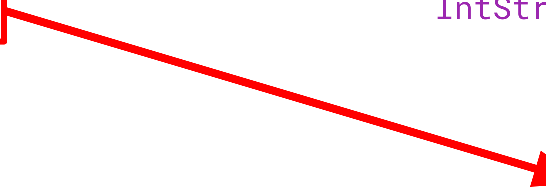
Stateful Operations - Example

```
IntStream.of(source)
          .unordered()
          .parallel()
          .distinct()
          .sum();
```

```
IntPredicate distinctPredicate =
                createDistinctIntPredicate();

IntStream.of(source)
          .unordered()
          .parallel()
          .filter(distinctPredicate)
          .sum();
```

Stateful Operations - Example

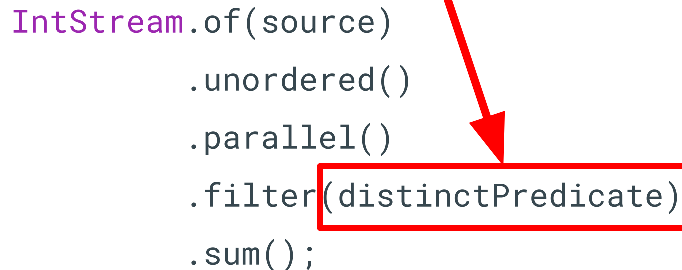
```
IntStream.of(source)
         .unordered()
         .parallel()
         .distinct()
         .sum();
```



```
IntPredicate distinctPredicate =
               createDistinctIntPredicate();

IntStream.of(source)
         .unordered()
         .parallel()
         .filter(distinctPredicate)
         .sum();
```

Stateful Operations - Example

```
IntStream.of(source)
        .unordered()
        .parallel()
        .distinct()
        .sum();
```



```
IntPredicate distinctPredicate =
                createDistinctIntPredicate();

IntStream.of(source)
        .unordered()
        .parallel()
        .filter(distinctPredicate)
        .sum();
```

Stateful Operations - Example

```
static IntPredicate createDistinctIntPredicate() {
    Set<Integer> seenElements = ConcurrentHashMap.newKeySet();

    return element -> {
        if (seenElements.contains(element)) {
            return false;
        }
        return seenElements.add(element);
    };
}
```

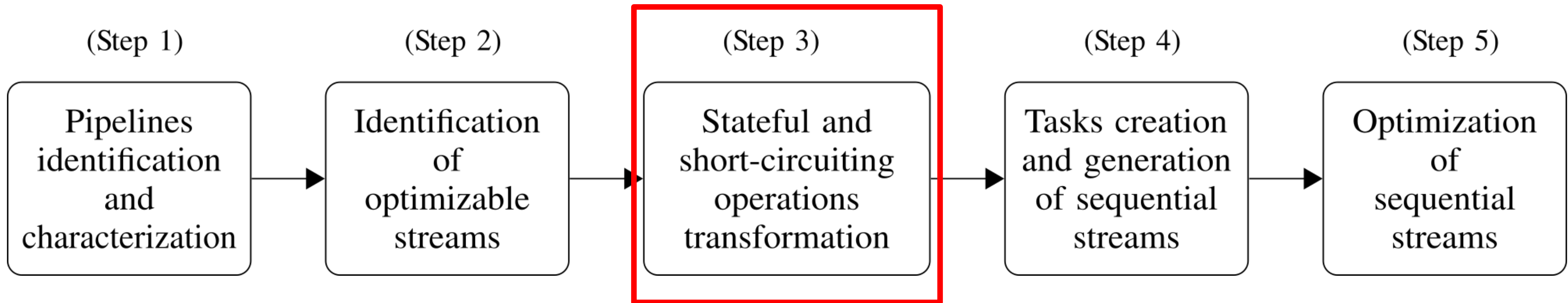
Stateful Operations - Example

```
static IntPredicate createDistinctIntPredicate() {
    Set<Integer> seenElements = ConcurrentHashMap.newKeySet();

    return element -> {
        if (seenElements.contains(element)) {
            return false;
        }
        return seenElements.add(element);
    };
}
```

Stateful Operations - Example

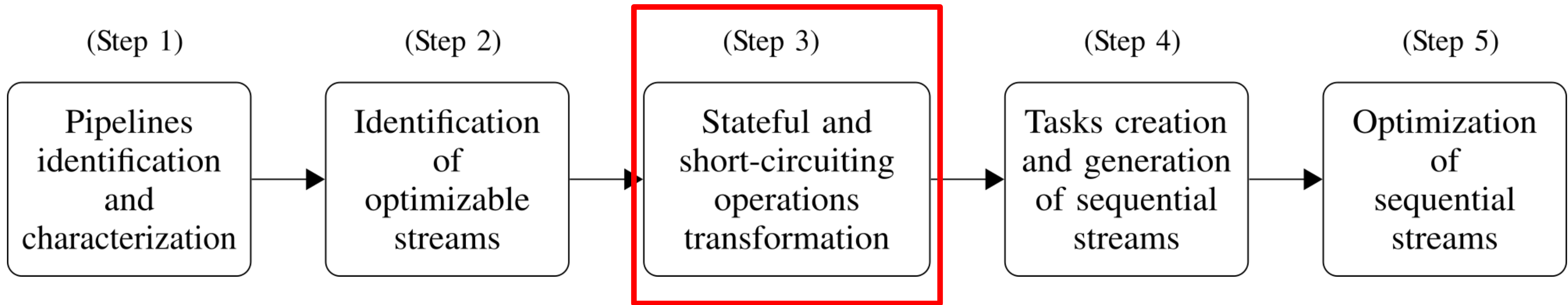
```java
static IntPredicate createDistinctIntPredicate() {
    Set<Integer> seenElements = ConcurrentHashMap.newKeySet();

    return element -> {
        if (seenElements.contains(element)) {
            return false;
        }
        return seenElements.add(element);
    };
}
```

Step 3 - Short-Circuiting Operations



- Short-circuiting operations cannot be efficiently moved into tasks
    - A short-circuiting operation would stop only the computation of its task but not the one of the other tasks
- Prevent the creation and submission of additional tasks
- Interrupt the concurrent processing of the different parts

Step 3 - Short-Circuiting Operations



- We transform:
  - short-circuiting intermediate operations to filters
  - short-circuiting terminal operations to mutable reductions
- A special `ShortCircuitingException` is thrown if element processing should be aborted due to short-circuiting

Short-Circuiting Operations - Example

```
IntStream.of(source)
         .unordered()
         .parallel()
         .limit(K)
         .sum();
```

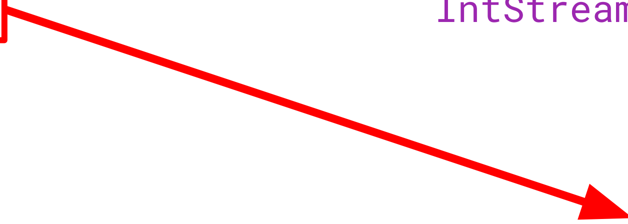
Short-Circuiting Operations - Example

```java
IntStream.of(source)
          .unordered()
          .parallel()
          .limit(K)
          .sum();
```

```java
IntPredicate limitPredicate =
             createLimitIntPredicate(K);

IntStream.of(source)
          .unordered()
          .parallel()
          .filter(limitPredicate)
          .sum();
```

Short-Circuiting Operations - Example

```
IntStream.of(source)
        .unordered()
        .parallel()
        .limit(K)
        .sum();
```

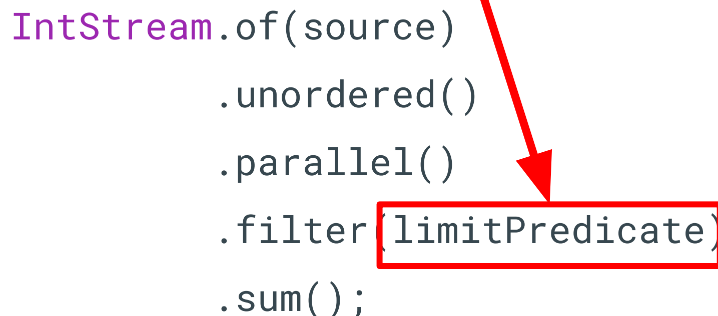


```
IntPredicate limitPredicate =
            createLimitIntPredicate(K);

IntStream.of(source)
        .unordered()
        .parallel()
        .filter(limitPredicate)
        .sum();
```

Short-Circuiting Operations - Example

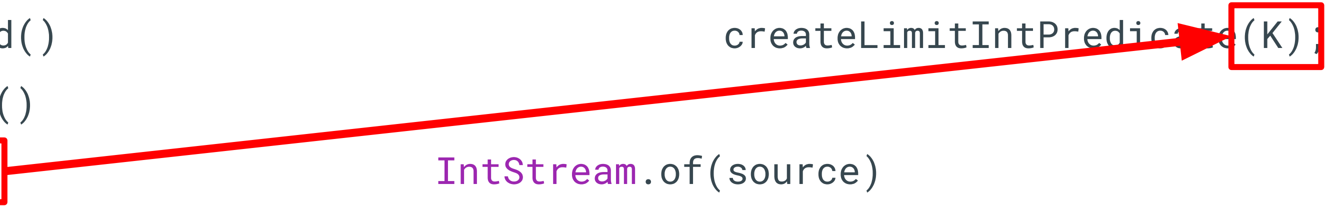
```
IntStream.of(source)
         .unordered()
         .parallel()
         .limit(K)
         .sum();
```

```
IntPredicate limitPredicate =
             createLimitIntPredicate(K);
```



Short-Circuiting Operations - Example

```java
IntStream.of(source)
         .unordered()
         .parallel()
         .limit(K)
         .sum();
```



```java
IntPredicate limitPredicate =
             createLimitIntPredicate(K);

IntStream.of(source)
         .unordered()
         .parallel()
         .filter(limitPredicate)
         .sum();
```

Evaluation

- Implementation of our approach that employs Streamliner [1] in Step 5
- Target workload: stream pipelines [1], [2]

[1] A. Møller et al., "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization". OOPSLA 2020.

[2] A. Biboudis et al., "Clash of the Lambdas".

Evaluation

- We evaluate four different versions of each benchmark
  - **StreamParallelOpt**: parallel streams optimized using our approach
  - **StreamParallel**: parallel streams
  - **StreamSequentialOpt**: sequential streams optimized using Steamliner
  - **StreamSequential**: sequential streams

Evaluation - Execution Time

StreamParallel | StreamSequentialOpt | StreamSequential

Speedup Factor (10^0 – 10^3)

Benchmark	StreamParallel	StreamSequentialOpt	StreamSequential
allMatch	3.35	7.61	22.75
filterCount	2.81	9.98	10.15
filterMapCount	3.78	9.63	11.07
megamorphicFilters	13.56	12.72	207.06
megamorphicMaps	33.77	7.28	504.22
sum	3	9.38	8.13
sumOfSquares	1.15	6.51	11.04
sumOfSquaresEven	3.89	12.82	11.97
distinctFew	201.03	16.19	22.77
distinctAll	1.01	6.5	6.65
limit	2.47	3.38	6.91

- ➢ Significantly faster code w.r.t. parallel streams
- ➢ The average speedup factor is 5.38× (geomean)
- ➢ Our approach always leads to faster code

Evaluation - Execution Time

StreamParallel StreamSequentialOpt StreamSequential

Speedup Factor

10^0 10^1 10^2 10^3

allMatch 3.35 7.61 22.75

filterCount 2.81 9.98 10.15

filterMapCount 3.78 9.63 11.07

megamorphicFilters 13.56 12.72 207.06

megamorphicMaps 33.77 7.28 504.22

sum 3 9.38 8.13

sumOfSquares 1.15 6.51 11.04

sumOfSquaresEven 3.89 12.82 11.97

distinctFew 201.03 16.19 22.77

distinctAll 1.01 6.5 6.65

limit 2.47 3.38 6.91

- Stateful distinct operation evaluated in distinctFew and distinctAll
  - Speedup factor 201.03× and 1.01×

Evaluation - Execution Time

StreamParallel StreamSequentialOpt StreamSequential

Speedup Factor

10^3

10^2

10^1

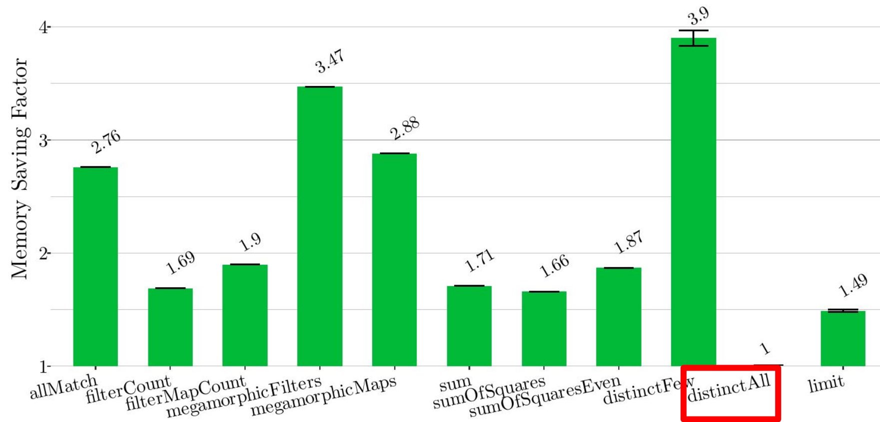
10^0

	StreamParallel	StreamSequentialOpt	StreamSequential
allMatch	3.35	7.61	22.75
filterCount	2.81	9.98	10.15
filterMapCount	3.78	9.63	11.07
megamorphicFilters	13.56	12.72	207.06
megamorphicMaps	33.77	7.28	504.22
sum	3	9.38	8.13
sumOfSquares	1.15	6.51	11.04
sumOfSquaresEven	3.89	12.82	11.97
distinctFew	201.03	16.19	22.77
distinctAll	1.01	6.5	6.65
limit	2.47	3.38	6.91

➢ Short-circuiting stateful limit operation evaluated in the limit benchmark

- Speedup factor 2.47×

Evaluation - Memory Allocation



- Significant memory savings
- On average, our technique reduces memory allocation by a factor of 2.05 × (geomean)
- In distinctAll, the memory pressure is dominated by the large ConcurrentHashMap which keeps track of the distinct elements

Limitations

- ➢ Our approach does not optimize streams whose pipeline is dynamically constructed
  - A stream execution might be either sequential or parallel depending on the runtime outcome of an if block
- ➢ Our approach does not optimize streams that are created and executed in different methods
  - Well-known limitation of related work [1] that optimizes sequential streams

[1] A. Møller et al., "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization". OOPSLA 2020.

Conclusions

- We presented a novel technique to remove the abstraction overhead of parallel unordered Java streams
- Our optimized code significantly reduces execution time and memory allocation
  - Average speedup factor: 5.38× (geomean)
  - Average memory-saving factor: 2.05× (geomean)
- As future work, we plan to optimize parallel ordered streams

Thanks for your attention

- Contacts:

  Matteo Basso

  matteo.basso@usi.ch