Optimizing Parallel Java Streams

Matteo Basso[*], Filippo Schiavio[†], Andrea Rosà[‡] and Walter Binder[§]
Faculty of Informatics, Università della Svizzera italiana (USI)
Lugano, Switzerland
Email: [*]matteo.basso@usi.ch, [†]filippo.schiavio@usi.ch, [‡]andrea.rosa@usi.ch, [§]walter.binder@usi.ch

***Abstract*—The Java Stream API increases developer productivity and greatly simplifies exploiting parallel computation by providing a high-level abstraction on top of complex data processing, parallelization, and synchronization algorithms. However, the usage of the Java Stream API often incurs significant runtime overhead. Method inlining and the automated translation of code using the Java Stream API into imperative code using loops can reduce such overhead; however, existing approaches and tools are applicable only to sequential stream pipelines, leaving the optimization of parallel streams an open issue. We bridge this gap by presenting a novel method to exploit high-level static analysis to characterize stream pipelines, detect parallel streams, and apply transformations removing the abstraction overhead. We evaluate our method on a set of benchmarks, showing that our approach significantly reduces execution time and memory allocation.**

***Index Terms*—Java, Parallel Streams, Functional Programming, Program Optimization, Bytecode Transformation**

I. Introduction

Nowadays, complex computer systems are gaining increasing interest in the research and industrial communities and are used in an ever-expanding set of applications that includes, e.g., big-data processing, software architectures, software engineering, and safety-critical systems. In such complex systems, performance is crucial, and hence it is essential to develop code that fully exploits modern multicore architectures. However, parallelizing code while achieving good maintainability and scalability is challenging and entails high complexity for the developer. For this reason, several programming languages implement simple declarative programming models that allow one to easily parallelize sequential computations.

This paper focuses on one of such models, the Java Stream API [21] (henceforth called Stream API for short), introduced in Java 8. The API mitigates the aforementioned issues by enabling functional-style operations to process *streams* of elements. In particular, the Stream API allows the developer to declaratively define a pipeline of operations that can be executed either sequentially or in parallel. By invoking a single method, the pipeline execution is automatically parallelized without any intervention or extra specifications required by the developer. Developers can focus on the intended behavior, easily exploiting parallel computations without having to deal with complex error-prone mechanisms related, e.g., to load balancing and synchronization.

However, despite the benefits of this abstraction, several studies show that functional programming in Java using the Stream API suffers from significant performance degradation compared with its imperative counterpart [2], [9], [10]. As the Java bytecode does not support function types, lambda expressions [24] (i.e., anonymous functions often used to specify stream operations) are treated as interfaces with a single abstract method [11], leading to many virtual method calls. These virtual calls may prevent optimizations performed by the Just-In-Time (JIT) compiler. To maintain the benefits of functional programming without sacrificing performance, it is crucial to better optimize programs that use the Stream API.

A prominent technique for removing the abstraction overhead of the Stream API is the automatic translation of pipelines into loops. Such a technique, based on bytecode transformations and method inlining, is particularly suitable for streams and has shown significant performance improvements [10], [14], [23]. However, to the best of our knowledge, existing strategies work only with sequential streams. Parallel streams are either ignored or converted to sequential ones, resulting in missed optimization opportunities. As a result, parallel stream processing may be suboptimal, hence lowering the efficiency of modern complex computer systems using parallel streams.

To address this issue, we propose a novel approach to improve the performance of programs that use parallel streams by automatically transforming them into imperative code, hence removing the abstraction overhead of the Stream API. Our approach relies on static analysis and on bytecode transformations. Contrary to related work, our approach considers the semantics of individual stream operations and characterizes stream pipelines. This allows us to exploit stream- and operation-attributes to determine operations that can be safely parallelized, as well as those that need synchronization.

In particular, we convert parallel streams into fork-join tasks, reducing runtime overhead, and we efficiently manipulate stateful and short-circuiting operations to lower contention and synchronization overheads. Our new transformations also enable the use of existing techniques that remove the abstraction overhead, previously only applicable to sequential streams, to further optimize the generated code. Our technique allows better exploitation of parallelism without any support from the developer and without introducing any change in the API, improving the efficiency of applications using parallel streams in terms of execution time and memory allocation.

Our work makes the following contributions.

1) We propose a novel method that performs static bytecode analysis to detect, characterize, and transform parallel Java streams into imperative code, removing the abstraction overhead (Section IV).

2) We present several optimizations for the transformation of stateful and short-circuiting operations to reduce synchronization and contention (Section V).
3) We evaluate our approach on a set of benchmarks, showing that the resulting code significantly reduces execution time and memory allocation w.r.t. parallel stream execution. In particular, our approach achieves an average execution-time speedup of $6.02\times$ (geometric mean) and a maximum one of $201.03\times$. Moreover, our method yields an average memory reduction of $2.07\times$ (geometric mean), up to a factor of $3.9\times$ (Section VI).

We complement the paper with an overview on the Stream API (Section II), a motivating example (Section III), and a comparison with related work (Section VII). Finally, Section VIII concludes.

II. Background: The Stream API

The Stream API [13] offers a functional interface to manipulate elements via a *pipeline* of operations. Such operations can be divided into two categories: *intermediate* operations either change stream attributes or return a new stream to be further processed by later operations, and *terminal* operations cause the execution of a pipeline, producing a result or side-effects. A stream pipeline starts with a method that creates the stream from a *data source* (such as a Collection, an array, or a generator function), potentially contains intermediate operations, and ends with a terminal operation. Intermediate operations can be further divided into *stateless* if they retain no state, or *stateful* if they incorporate a state that is updated when processing elements. For example, map and filter are stateless operations, while distinct and skip are stateful operations.[1] Intermediate operations that produce finite streams from infinite data sources and terminal operations that may terminate in finite time from infinite input are classified as *short-circuiting*. Two examples of short-circuiting operations are limit and anyMatch.

Streams can be classified as either *finite* or *infinite*, *ordered* or *unordered* (based on the presence or absence of an *encounter order*), and *sequential* or *parallel*. Given the operations composing the stream pipeline, it is possible to classify the stream and execute the pipeline.

One main advantage of the Stream API is that it greatly facilitates the parallelization of element processing just by adding a single parallel operation that classifies the stream as *parallel*. Internally, through a divide-and-conquer approach, the Stream API splits input data of parallel streams into parts and orchestrates ForkJoinTask [16] instances that process each part in parallel. Such task instances are automatically executed by the common ForkJoinPool instance [15], which is statically constructed by the Java Class Library, requiring no user intervention. The configuration of the API is not under user control.

[1]All operations on streams reported in the paper correspond to methods defined in the Stream, IntStream, LongStream, and DoubleStream interfaces of the java.util.stream package. A complete classification of operations in terms of intermediate/terminal, stateless/stateful, and short-circuiting can be found in the documentation [21]. Using a functional-language notation, we use “operation” as a synonym for “method”.

```
 1  int sumOfSquaresEven(int[] source) {
 2
 3    return IntStream.of(source)
 4                    .unordered()
 5                    .parallel()
 6                    .filter(
 7                         x -> x % 2 == 0
 8                    )
 9                    .map(x -> x * x)
10                    .sum();
11
12  }
```

Fig. 1. Parallel, unordered stream computing the sum of squares of even numbers.

III. Motivating Example

In this section, we report an example of parallel stream and we illustrate the corresponding imperative code produced by our approach.

The example method sumOfSquaresEven (whose sequential ordered version has been originally proposed in a paper by Biboudis et al. [2]) shown in Figure 1 uses a parallel, unordered stream, computing the sum of squares of even numbers stored in the array source provided as a parameter. The IntStream.of method (line 3) creates a new stream that has the source array as its data source. Lines 4 and 5 define the stream as *unordered* and *parallel*, respectively, using the homonymous operations. The stateless filter operation (lines 6–8) selects only even numbers, while the stateless map operation (line 9) computes the squares. The terminal operation sum (line 10) sums the squares, yielding the final result. Since the stream is parallel, the Stream API automatically parallelizes the computation by splitting the source array into several parts assigned to different fork-join tasks that are automatically managed by a fork-join pool.

Our approach aims at transforming the parallel stream in Figure 1 into a corresponding imperative code. Consider the code in Figure 2, showing a partial transformation of the code in Figure 1. The sumOfSquaresEven method in Figure 2 does not contain a parallel stream, but instead the instantiation of an inner class called SumOfSquaresEvenTask (line 3) that extends ForkJoinTask, and an invocation to its compute method (line 24). Such a method, omitted for brevity, splits the data source into parts, creates new tasks, and invokes the computePart method (lines 7–14) on each task. Since the data source of the original stream is an array, splitting is performed by providing the starting and ending indexes of the part (called low and high in the code) as parameters to computePart (line 7). In case of data sources that do not support direct indexing, computePart may take e.g. a Spliterator[2] instance as parameter. The computePart method creates and executes a sequential stream composed of the same operations of the original parallel stream, but considering only the part assigned

[2]A Spliterator [17] is an object that allows traversing (either individually or sequentially in bulk) and partitioning elements of a source. Such as source can be, for example, an array, a Collection, or a generator function.

```
1  int sumOfSquaresEven(int[] source) {
2
3      class SumOfSquaresEvenTask extends ForkJoinTask<Integer>  {
4
5          // constructor, compute, and other methods are omitted
6
7          int computePart(int low, int high) {
8
9            return Arrays.stream(source, low, high)
10                        .filter(x -> x % 2 == 0)
11                        .map(x -> x * x)
12                        .sum();
13
14         }
15
16         int merge(int left, int right) {
17
18           return left + right;
19
20         }
21
22     }
23
24     return new SumOfSquaresEvenTask().compute();
25 }
```

Fig. 2. Code of Figure 1 after transformation, using sequential streams executed in parallel via ForkJoinTask instances.

to the task and providing a partial result (line 9–12). Finally, the merge method (line 16), called by compute, aggregates partial results coming from different parts to obtain the final sum.

Since the computePart method contains a sequential stream, existing techniques can be applied to convert such a stream into a single loop, hence completely removing the Stream API abstraction. Such techniques could not have been applied to the stream in Figure 1, as it is parallel. Our work makes it possible to automatically translate code containing parallel streams (Figure 1) into equivalent code using only sequential streams that are executed in parallel in fork-join tasks (Figure 2), as we will explain in the next section.

IV. Methodology

In this section, we present our approach to transform parallel streams. First, we report the base case of our method in Section IV-A. Then, in Section IV-B we explain how we support stateful and short-circuiting operations.

A. Base Case: Sequence of Stateless Operations

Our approach is based on static analysis and bytecode transformations. Differently from existing techniques, the proposed method operates at a higher abstraction level, considering the semantics of stream operations and performing a characterization of the stream pipelines.

We focus on stream pipelines that are created and executed within the same method, a strategy also used by similar techniques focusing on sequential streams [14]. This is motivated by the fact that Java streams are rarely returned or passed as parameters, as shown by related work [14]; hence, they usually span a single method. As a consequence, our approach cannot fully transform the flatMap operation (limited to the case where the function passed as input returns a parallel stream).

In addition, our method focuses on *parallel unordered* streams, since the most significant speedups can be obtained for such streams. Indeed, without ordering constraints, it is possible to fully exploit parallel out-of-order execution, removing buffering[3] to achieve better performance. As it will be later explained in Section VII, related work [7], [8] proposes approaches to identify sequential streams that can be safely converted to *parallel unordered* streams. Such streams can benefit from our method, hence expanding the applicability of our technique. Furthermore, the Stream API documentation recommends either to use unordered streams or to avoid the usage of parallel ordered ones, if certain operations (such as limit) are part of the pipeline, since synchronization and buffering overheads may be excessive [19], [21].

In the following text, we detail our approach to identify, characterize, and optimize parallel stream pipelines that only contain stateless, non-short-circuiting operations. Figure 3 illustrates our approach. In the base case considered here, Step 3 (i.e., transformation of stateful and short-circuiting operations) has no effect and will be skipped. We detail Step 3 in Section IV-B, which enables cases where stateful and short-circuiting operations appear in the stream pipeline. Our methodology is applied to the bytecode of Java methods that create and execute at least one stream.

Step 1 identifies and characterizes the pipeline of each stream. We use the approach proposed by Møller at al. [14], i.e., an off-the-shelf pointer analysis [14], [25], to identify bytecode instructions that belong to different pipelines and

[3]In this paper, "buffering" refers to the allocation of temporary intermediate data structures to store elements between computational steps.

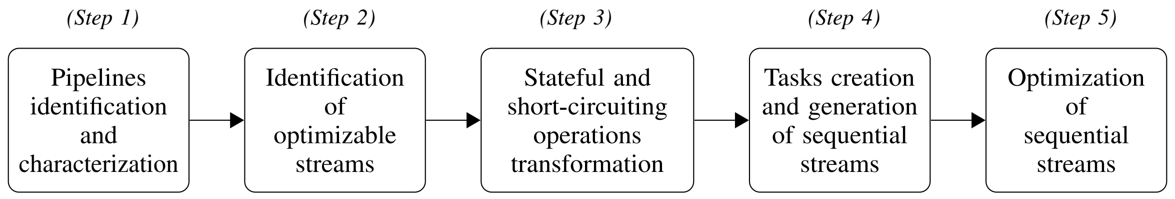

(Step 1)
Pipelines identification and characterization
(Step 2)
Identification of optimizable streams
(Step 3)
Stateful and short-circuiting operations transformation
(Step 4)
Tasks creation and generation of sequential streams
(Step 5)
Optimization of sequential streams

Fig. 3. Overview of the proposed parallel stream optimization method.

need to be processed separately. Then, by further analyzing the bytecode instructions, we characterize each pipeline by analyzing each stage of the pipeline and tracking changes in the attributes of the stream, starting with the method that creates it (e.g., IntStream.of() or Arrays.stream()) and ending with the terminal operation. For example, a pipeline that contains a parallel operation not followed by a sequential operation is marked as parallel. By analyzing the method that creates the stream and the terminal operation, we extract type information about the stream itself, i.e., the input/output types and the data source. Step 2 uses the stream characterization of Step 1 to detect *parallel unordered* streams, which can be optimized by our approach.

Step 4 uses the input/output types of the stream identified in Step 1 to convert a parallel stream into a sequential stream processed by a ForkJoinTask. Considering our motivating example, this step converts the parallel stream of Figure 1 into the code of Figure 2, as detailed in Section III. In particular, we move the bytecode instructions that form the pipeline from the original method to the computePart method of the newly created task class. In the original method, we substitute the parallel stream with an instantiation of a new task and an invocation of its compute method (Figure 2, line 24).

Finally, Step 5 resorts to existing techniques [3], [14] to optimize sequential streams, providing the computePart method of the new task class as input (which contains a sequential stream). Hence, we obtain a single loop, completely removing the abstraction of the Stream API.

While our approach removes the abstraction overhead of using parallel streams, the resulting code uses the same number of fork-join tasks that would be created using the Stream API. Our approach does not alter task granularity, i.e., we use the same logic implemented in the Stream API to determine whether to create new fork-join tasks or to sequentially process a part. In addition, we do not alter the way in which the ForkJoinPool of the Stream API handles parallelism.

B. Stateful and Short-Circuiting Operations

Here, we detail Step 3, i.e., how stateful and short-circuiting operations are manipulated to make them suitable to our technique. Even though we present distinct and limit as examples of stateful and short-circuiting operations, respectively, our method also applies to other operations, e.g., skip. In the following text, we illustrate our approach focusing on IntStreams (and hence int operations and int primitive-type specialization classes, such as IntPredicate) to simplify code and explanation. However, our method can be applied to any primitive-type or object stream with the appropriate simple type modifications.

1) **Stateful operations:** Stateful operations cannot be directly moved into fork-join tasks while ensuring correctness. For example, considering the distinct operation (that removes duplicate stream elements), placing it directly into the computePart method in Figure 2 would lead to different elements only within the given part, not across the whole data source. Hence, the global output may contain the same element more than once, violating the semantics of the distinct operation. Moreover, it is not possible to perform distinct in the merge method, because elements may have been manipulated and aggregated by operations that follow distinct in the stream pipeline. For example, the merge method that corresponds to the sum terminal operation (shown in Figure 2 at lines 16–20) can access only two aggregated results (called left and right) and cannot detect and consider only the distinct elements that compose such aggregated results.

Our solution is to exploit a common synchronized state shared among the different tasks. We observe that every stateful operation defined in the Stream API [18] (except sorted) can be considered as a specialization of a filter operation. Hence, we transform stateful operations to stateless filter operations, providing them dedicated predicates in charge of managing the shared state before transforming the pipeline in Step 4. This manipulation preserves the semantics of the transformed stateful operations and hence guarantees the correctness of the result of the computation [18]. Moreover, our manipulation does not introduce the need for data buffering, hence avoiding unnecessary memory allocations.

Figure 4 shows how the distinct operation at line 6 can be translated into a stateless filter (line 17) whose predicate is returned by the createDistinctIntPredicate method at line 12 (implemented in Section V-A) and checks whether the provided element has already been encountered. Figure 5 reports the transformed pipeline shown in Figure 4, after Step 4. The createDistinctIntPredicate method invocation (line 2) is inserted before the definition of the fork-join task (lines 5–18) and hence all the fork-join task instances use the same predicate instance (line 13). Therefore, the same thread-safe Set instance (Figure 7, line 5) is used for checking for duplicates when executing the parallel stream.

2) **Short-circuiting operations:** In a concurrent context, short-circuiting operations need to be handled with care. For

```
// Original pipeline

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

// Transformed pipeline (step 3)

IntPredicate distinctPredicate =
          createDistinctIntPredicate();

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

Fig. 4. Transformation of a stateful distinct operation to a stateless filter operation whose predicate manages a shared state.

```
1  // Original pipeline
2
3  IntStream.of(source)
4           .unordered()
5           .parallel()
6           .limit(K)
7           .sum();
8
9  // Transformed pipeline (step 3)
10
11 IntPredicate limitPredicate =
12           createLimitIntPredicate(K);
13
14 IntStream.of(source)
15          .unordered()
16          .parallel()
17          .filter(limitPredicate)
18          .sum();
```

Fig. 6. Transformation of a stateful short-circuiting limit operation to a stateless filter operation.

```
IntPredicate distinctPredicate =
        createDistinctIntPredicate();

class SumOfDistinctTask
        extends ForkJoinTask<Integer>  {

  // Constructor, compute, merge,
  // and other methods are omitted

  int computePart(int low, int high) {

    return Arrays.stream(source, low, high)
                 .filter(distinctPredicate)
                 .sum();

  }

}

new SumOfDistinctTask().compute();
```

Fig. 5. Transformed pipeline of Figure 4 after Step 4.

example, considering the limit(K) operation (that filters the first K elements), an ideal implementation would process only K elements, avoiding unnecessary computations. In a concurrent setting, we need to prevent the creation and submission of additional tasks and interrupt the concurrent processing of the different parts.

Our approach solves these issues as follows. We replace short-circuiting intermediate operations with a filter, and short-circuiting terminal operations with a mutable reduction, i.e., collect. While filter takes a Predicate as input, collect takes a Collector. The predicate or collector throws a special ShortCircuitingException if element processing should be aborted due to short-circuiting. Indeed, since streams do not provide any way to stop a pipeline operation once it has started, an exception is able to break the normal control flow. Each task is able to catch the ShortCircuitingException and either cancel ongoing computations or send a signal to the other tasks to prevent the creation of additional parts. In case of short-circuiting, stateful intermediate operations, the Predicate provided to filter is used to both handle short-circuiting and manage a shared state.

Figure 6 outlines our approach in the example of limit(K), a short-circuiting, stateful intermediate operation. The limit operation at line 6 is transformed into a filter operation (line 17), which takes a Predicate as input (provided by method createLimitIntPredicate(K) at line 12 and whose implementation exploits the data structure described in Section V-B). Such a Predicate either returns true or short-circuits the computation by throwing a ShortCircuitingException instead of returning false (and filtering only the single element currently being processed).

V. Implementation

This section details our techniques to reduce synchronization and contention. First, Section V-A provides implementation details of the createDistinctIntPredicate method (introduced in Section IV-B for transforming the distinct operation). Then, Section V-B provides a high-level explanation of the data structure internally used by the predicate returned by createLimitIntPredicate (used for transforming the limit operation). Due to lack of space, we do not provide the detailed Java implementation of such a data structure, of the createLimitIntPredicate method, and of the code that handles the ShortCircuitingException.

A. **distinct()**

The implementation of the distinct operation in the Stream API (OpenJDK [20]) uses a ConcurrentHashMap<T, Boolean> (where T is the type of the elements). In particular, the implementation uses the putIfAbsent method defined in ConcurrentHashMap to keep track of the encountered elements, i.e., to determine whether one element must be excluded because a corresponding equal element has already been encountered. However, this approach can lead to performance degradation due to synchronization in putIfAbsent.

```
1  static IntPredicate createDistinctIntPredicate() {
2
3      // The thread-safe seenElements Set instance is shared among
4      // all the tasks associated to the same stream pipeline
5      Set<Integer> seenElements = ConcurrentHashMap.newKeySet();
6
7      return element -> {
8
9          if (seenElements.contains(element)) {
10             return false;
11         }
12
13         return seenElements.add(element);
14
15     };
16 }
```

Fig. 7. Factory method to create filter predicates that correspond to the distinct operation.

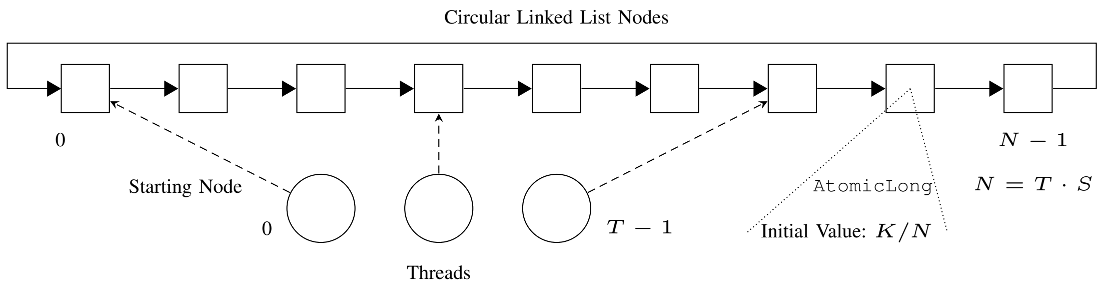


Fig. 8. Counter data structure exploiting N atomic variables to reduce contention. K is the value of the counter, T is the number of threads that access the structure, and S is a parameter that defines the number of nodes per thread. In the figure, $T = S = 3$.

As shown in Figure 7, our implementation of the createDistinctIntPredicate method mitigates this issue by relying on a concurrent Set backed by a ConcurrentHashMap[4] (line 5). To determine whether an element is already in the set, we use the pattern shown in lines 9–13. Since checking whether an element is in the set does not require synchronization (Set.contains at line 9 internally uses ConcurrentHashMap.get, a retrieval operation that does not entail locking), threads do not block, significantly speeding up the execution if several equal elements are provided. If the item is not already in the set, the thread-safe Set.add (line 13) adds it and returns a boolean indicating whether the the element has been added.

B. **limit(K)**

The implementation of the limit(K) operation in the Stream API (OpenJDK) relies on buffering and works as follows. The operations in the pipeline that precede limit are executed on each element; the resulting elements are placed into a fixed-sized buffer. Then, the limit(K) operation processes elements from such buffer, decrementing an atomic counter (initialized with the value K) until it reaches zero. In this way, the counter can be decremented by a number corresponding to the buffer size.

[4]The Java Class Library does not provide a ConcurrentHashSet class. For this reason, we use the keyset of a ConcurrentHashMap.

Our approach aims at exploiting parallel computation as much as possible, avoiding buffering, and creating a single entirely parallelizable pipeline suitable for method inlining. This design choice forces the decrementing operation on the shared atomic counter to be executed for each processed element rather than for each processed buffered sequence, i.e., the counter has to be decremented (by 1) for each processed element. Such an approach could be naively implemented with a single, shared AtomicLong counter. However, although such a solution would work correctly, it may result in poor performance due to high contention on the shared counter.

To mitigate this issue, the predicate returned by the createLimitIntPredicate method relies on a dedicated data structure representing a concurrent counter that starts with an initial value K and that can be only decremented until zero. Internally, such a counter encapsulates multiple AtomicLong variables, thus reducing contention. In particular, our data structure, whose conceptual high-level view is shown in Figure 8, is composed of a circular linked list of $N = T \cdot S$ nodes, where T is the number of threads that access the structure and S is a parameter that defines the number of nodes per thread (in the figure, $T = 3$ and $S = 3$). Each node encapsulates an AtomicLong counter with a initial value of K/N. If K is not multiple of N, the remainder is distributed among the first counters. When the i-th thread ($i \in [0, T-1]$) accesses the structure for the first time, it is assigned a starting node at position $i \cdot S$, such that the starting positions are well distributed among the list.

When a certain thread needs to determine whether the global counter can be decremented, it searches for an AtomicLong whose value is greater than zero, starting from its starting node and traversing the list. If a matching AtomicLong can be found, the thread decrements it and marks it as its new starting node to speed up further list traversals (i.e., skipping counters that are known to be zero). Otherwise, if no positive counter is present, the data structure is marked as exhausted and a ShortCircuitingException is thrown.

Such an approach reduces contention in the early execution stages by evenly distributing threads among the list. As a consequence, different threads access different counters and do not need to block, saving execution time. Contention becomes significant only when there are few local counters that are greater than zero and multiple threads try to access them concurrently. List traversal can be efficiently implemented by checking whether an AtomicLong is positive before atomically decrementing it and comparing it with zero (i.e., performing a first check that does not execute any expensive and possibly repeated compare-and-swap operation).

The value of the parameter S is a compromise between memory allocation and contention (and hence execution time). In particular, memory allocation is proportional to S (since higher values of S lead to the allocation of more nodes) while contention is inversely proportional to S (since more threads try to access the same node and require synchronization). A value of S could be computed as a function of T and K. However, an optimal value of S would consider several other factors, such as the workload, the data source, level of parallelism, and task granularity. We have experimentally determined that a value $S = 6$ leads to a reasonable memory consumption and contention. A more precise estimation of an optimal value of S is left as future work.

VI. Evaluation

In this section, we evaluate our approach in terms of execution-time speedups and memory savings w.r.t. the Stream API . We first present the evaluation settings (Section VI-A), then we discuss the benefits of our approach in terms of execution time (Section VI-B) and memory savings (Section VI-C).

A. Evaluation Settings

We evaluate an implementation of our approach that employs Streamliner [14] in Step 5 (see Figure 3), a state-of-the-art tool to optimize sequential streams through bytecode transformations. We evaluate our method on benchmarks composed of different stream pipelines, which were first presented by Biboudis et al. [2] and later extended by Møller et al [14]. The benchmark suite has been used to evaluate related approaches, particularly Strymonas and Streamliner.

We evaluate four different versions of each benchmark: (1) using parallel streams and applying our method to remove the abstraction of the Stream API (called StreamParallelOpt); (2) using parallel streams (called StreamParallel); (3) using sequential streams and applying Streamliner to remove the abstraction of the Stream API (called StreamSequentialOpt); (4) using sequential streams (called StreamSequential). Comparing StreamParallelOpt with StreamParallel allows to evaluate our method, i.e., determining the improvement obtained by removing the abstraction of parallel streams. Comparing our approach with StreamSequential and StreamSequentialOpt allow verifying that the parallel optimized version resulting from the proposed method performs better than rewriting the streams as sequential (StreamSequential) and applying Streamliner to optimize the sequential streams (StreamSequentialOpt). Comparing the speedup enabled by Streamliner on sequential streams and the speedup enabled by our approach on parallel streams does not fall in the scope of the evaluation. Instead, our goal is verifying that the code version resulting from our method is the most efficient among the four evaluated ones. As mentioned in Section IV-A, our method uses the same number of fork-join tasks that would be created using the Stream API and does not alter task granularity. Hence, StreamParallel and StreamParallelOpt use the same level of parallelism.

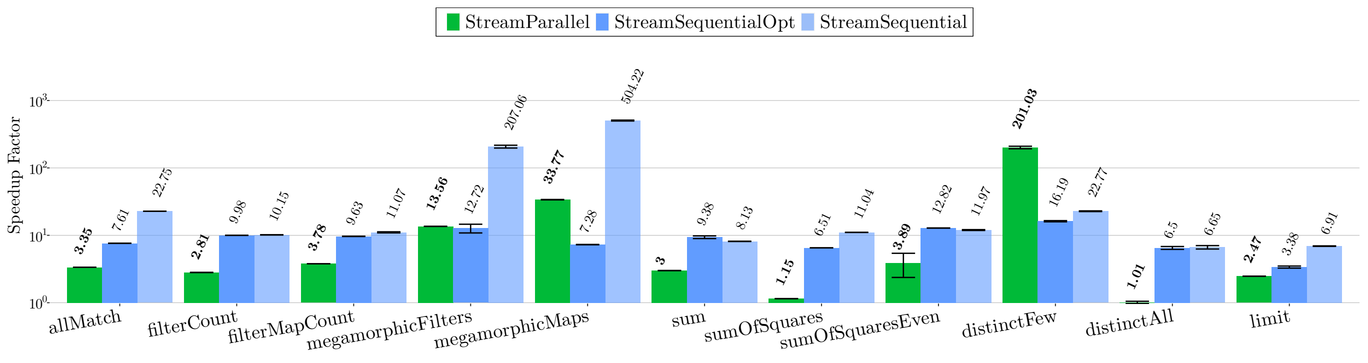
We run all experiments on two machines M_A and M_B. M_A is equipped with an 18-core Intel i9-10980XE (3.00 GHz) with 256 GB of RAM. M_B is equipped with two NUMA nodes, each with an 8-core Intel Xeon E5-2680 (2.7 GHz) and 64 GB of RAM. The operating system is Linux Ubuntu (kernel version 5.4.0-58-generic) and the language runtime is Oracle JDK, build 11.0.10+8-LTS-16 (i.e., the latest long-term-support JDK release at the time of writing).

For each benchmark, we first perform 5 warm-up iterations to let dynamic compilation and garbage-collection ergonomics stabilize; then, we run 10 steady-state iterations. The figures in this section report the arithmetic mean over the steady-state iterations.

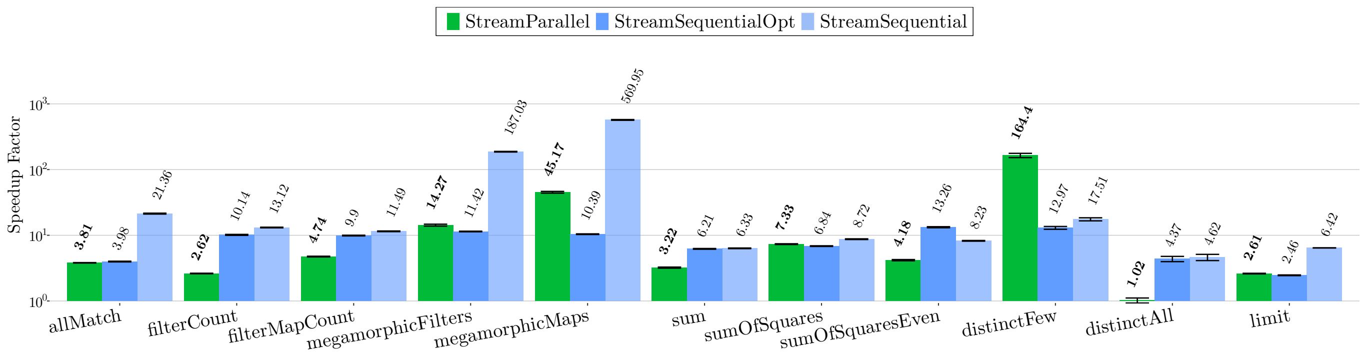
B. Execution Time

Figure 9 reports our results in terms of execution-time speedup factor (i.e., the ratio between StreamParallel, StreamSequential, StreamSequentialOpt, and our approach StreamParallelOpt) on both machines. In particular, the x-axis reports different benchmarks, while the logarithmic y-axis reports the speedup factor. Above each bar, we report the speedup of the code resulting from our approach (StreamParallelOpt) w.r.t. the corresponding code version.

Overall, our approach of transforming parallel streams into iterative loops yields code that is significantly faster than parallel streams. This is remarked by the fact that each green StreamParallel bar is greater than 1. On M_A, the speedups range from a factor of $1.01\times$ (*distinctAll*) to a factor of $201.03\times$ (*distinctFew*), while on M_B, they span values between $1.02\times$ (*distinctAll*) and $164.4\times$ (*distinctFew*). The average speedup factor (geometric mean) is $5.38\times$ and $6.75\times$ for M_A and M_B, respectively, and $6.02\times$ across both machines. In addition, transforming parallel streams into imperative loops using our approach always leads to faster code than both transforming the stream into a sequential one (StreamSequential



(a) Machine A (M_A).



(b) Machine B (M_B).

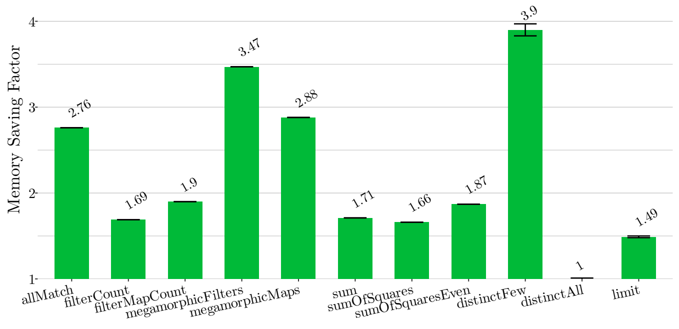
Fig. 9. Speedup factors achieved by the proposed approach (StreamParallelOpt). Error bars indicate standard deviations.

bar) and applying Streamliner to it (to transform a sequential stream into imperative code; StreamSequentialOpt bar). The average speedup factor across both machines w.r.t. StreamSequential and StreamSequentialOpt is $19.51\times$ (geometric mean) and $7.98\times$, respectively. The maximum speedup factor is $569.95\times$ for StreamSequential (*megamorphicMaps* on M_B) and $16.19\times$ for StreamSequentialOpt (*distinctFew* on M_A).

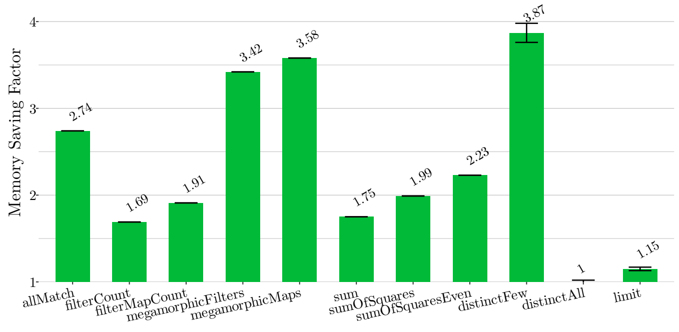
We now provide more details on individual benchmarks. Among the benchmarks that do not use any distinct or limit operation (all except *distinctFew*, *distinctAll*, and *limit*), the highest speedup w.r.t. StreamParallel can be observed in *megamorphicFilters* ($13.56\times$ on M_A, $14.27\times$ on M_B) and *megamorphicMaps* ($33.77\times$ on M_A, $45.17\times$ on M_B). The reason is that in such benchmarks, the streams execute a series of operations that makes it difficult for the JIT compiler to optimize the stream pipeline (i.e., a sequence of filter and map operations, respectively). Our approach is particularly useful in such cases, as it completely removes such operations, resulting in code more easily optimizable by the JIT compiler.

Our approach to transform the distinct operation is evaluated in two benchmarks, i.e., *distinctFew* and *distinctAll*. These two benchmarks are composed of the same pipeline of operations. In *distinctFew* there are only 10 distinct elements in the stream out of a total of 10^8 elements, while in *distinctAll* the pipeline is composed of 10^7 elements that are all different. These two settings allow us to compare a best- (*distinctFew*) and worst-case scenario (*distinctAll*) for our approach. In the best-case scenario, the non-blocking test shown in Figure 7 (seenElements.contains(element) at line 9) returns true in most cases, without incurring synchronization overheads and leading to the execution of line 10. This leads to an impressive speedup of $201.03\times$ on M_A and $164.4\times$ on M_B w.r.t. StreamParallel. On the other hand, in the worst-case scenario, the non-blocking test always returns false, leading to the invocation of the add method at line 13 (which contains sinchronization), and hence to a negligible speedup factor of $1.01\times$ on M_A and $1.02\times$ on M_B.

Finally, regarding the limit operation (evaluated in the *limit* benchmark), our approach outperforms StreamParallel by a factor of $2.47\times$ on M_A and $2.61\times$ on M_B. This speedup mainly stems from the implementation of the counter data structure discussed in Section V-B. To confirm this observation, we evaluate two different variants of our approach: the current one (using the data structure shown in Figure 8) and a "naive" one using a single shared AtomicLong. The former yields code that is $20.55\times$ faster than the latter, confirming the improved efficency of our implementation.



(a) Machine A (M_A).



(b) Machine B (M_B).

Fig. 10. Memory-saving factors enabled by the proposed approach (StreamParallelOpt). Error bars indicate standard deviations.

C. *Memory Allocation*

Figure 10 reports our results in terms of memory-saving factor, i.e., the ratio between the memory allocation of StreamParallel and StreamParallelOpt, where "memory allocation" refers to the allocated bytes per benchmark iteration. We report the memory saving factor on both M_A and M_B.

As the figure shows, our implementation significantly reduces the memory allocation in most of the benchmarks. With the exception of *distinctAll*, on M_A, reductions span from a factor of $1.49\times$ (*limit*) to $3.9\times$ (*distinctFew*), while on M_B they range from $1.15\times$ to $3.87\times$. On average (including *distinctAll*), our technique reduces memory allocation by a factor of $2.05\times$ on M_A, $2.11\times$ on M_B, and $2.07\times$ across both machines. Such reductions stem from the fact that our technique removes the abstraction of the Stream API, which causes the allocations of many objects in each intermediate operation during the processing of a stream pipeline. The only benchmark in which our technique does not reduce the memory allocation is *distinctAll*. This is expected, since in distinctAll the memory pressure is dominated by the large ConcurrentHashMap which keeps track of the 10^7 distinct elements (even without using our approach).

VII. Related Work

Below, we present related work that focuses on the Stream API.

Some related work optimize parallel streams by using hardware accelerators. Ishizaki et al. [6] speed up parallel stream execution by compiling lambda expressions into GPU code and automatically generating runtime calls that handle low-level operations, such as memory management. Hayashi et al [5] develop a JIT compiler exploiting supervised machine learning techniques to construct performance heuristics that allow selecting a preferable hardware device for parallel stream execution. To the best of our knowledge, these approaches do not remove the abstraction of the Stream API and hence are complementary to our technique.

Khatchadourian et al. [7], [8] present an automated refactoring approach to determine when a sequential stream can be safely converted into a parallel and potentially unordered stream. Differently from our work, the execution of parallel streams is not optimized. Hence, the work by Khatchadourian et al. is synergistic with our approach, as it can increase the number of parallel streams where our approach is applicable.

Mei et al. [12] study the usage of the Stream API to develop real-time systems. Their study demonstrates that the Stream API implementation does not allow the usage of real-time worker threads and hence is not suitable to be used in real-time contexts. To overcome this limitation, they propose some implementation-code changes. In contrast to our approach, they do not propose a technique to remove the abstraction overhead of parallel streams.

Differently from parallel streams, several studies focus on the optimization of sequential streams, either via compiler optimizations or by providing more efficient API implementations. Streamliner [14] implements a technique to remove the abstraction overhead of sequential streams. Similarly to our method, it performs bytecode transformations and exploits a pointer analysis to detect stream pipelines. However, since the transformations are low-level and based only on inlining and stack allocation, parallel streams are not supported.

Ribeiro et al. [23] attempt to provide a new augmented Java streaming library whose streams can be optimized via *stream fusion* [4], a technique that allows removing intermediate structures that are allocated during stream processing. This technique is not directly applicable to the existing Stream API, which would need to be properly rewritten to exploit such an optimization. In addition, to the best of our knowledge, this transformation does not work with parallel streams and cannot be automatically applied in Java due to missing meta-programming features such as Haskell's *rewrite rules* [22], hence requiring developer intervention.

StreamAlg [1] offers a new streaming library designed for extensibility and high performance. While the new library outperforms the Stream API, it does not support parallel streams.

VIII. Conclusion

Here, we illustrate the limitations of our method and we give our concluding remarks.

Limitations: Similarly to related work [14], our approach can optimize only streams that are created and executed within the same method. In addition, it may not be possible to optimize streams whose pipeline cannot be statically characterized. For example, a stream execution might be either sequential or parallel depending on the outcome of an if block. In such a case, our static analysis cannot determine whether the stream is parallel or not. Even though a developer can refactor the code such that our method can be applied, we are investigating techniques that require no manual intervention to automatically overcome these limitations.

Concluding Remarks: In complex computer systems, it is crucial to fully exploit modern multicore architectures and obtain high performance while preserving code maintainability and scalability. The Stream API allows one to easily parallelize code by offering a convenient declarative programming model. However, the abstraction overhead of such an API often reduces the effectiveness of the parallelization and increases memory allocation. In this paper, we present a novel method based on bytecode transformations to remove the abstraction overhead of parallel, unordered Java streams by creating specialized fork-join tasks. Our method requires neither developer intervention nor changes in the Stream API. We illustrate several optimizations to reduce the contention and synchronization overhead in the transformation of stateful and short-circuiting operations. Experimental results show that our optimized code significantly reduces execution time and memory allocation. In particular, our approach yields an average execution-time speedup of $6.02\times$ up to a maximum of $201.03\times$, while reducing memory allocation by a factor of $2.07\times$ on average and up to a factor of $3.9\times$.

As part of our future work, we plan to fully support the flatMap operation and to extend our technique to support parallel ordered streams. We also plan to release an open-source tool implementing our approach in the near future.

Acknowledgments

This work has been supported by the Swiss National Science Foundation (project 200020_188688).

References

[1] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. Streams a la Carte: Extensible Pipelines with Object Algebras. In *ECOOP*, pages 591–613, 2015.

[2] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the Lambdas. page 1–11, 2014.

[3] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *OOPSLA*, page 1–19, 1999.

[4] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *ICFP*, page 315–326, 2007.

[5] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. Machine-Learning-Based Performance Heuristics for Runtime CPU/GPU Selection. In *PPPJ*, page 27–36, 2015.

[6] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. In *PACT*, pages 419–431, 2015.

[7] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. *Science of Computer Programming*, 195:1–24, 2020.

[8] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. A Tool for Optimizing Java 8 Stream Software via Automated Refactoring. In *SCAM*, pages 34–39, 2018.

[9] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. An Empirical Study on the Use and Misuse of Java 8 Streams. In *FASE*, pages 97–118, 2020.

[10] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. In *POPL*, page 285–299, 2017.

[11] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, 2017.

[12] Hai Tao Mei, Ian Gray, and Andy Wellings. Integrating Java 8 Streams with The Real-Time Specification for Java. In *JTRES*, page 1–10, 2015.

[13] Michael Duigou. Java Enhancement Proposal 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107, 2011.

[14] Anders Møller and Oskar Haarklou Veileborg. Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization. *Proc. ACM Program. Lang.*, 4(OOPSLA):1–29, 2020.

[15] Oracle and/or its affiliates. Class ForkJoinPool. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinPool.html, 2021.

[16] Oracle and/or its affiliates. Class ForkJoinTask. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinTask.html, 2021.

[17] Oracle and/or its affiliates. Class Spliterator. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Spliterator.html, 2021.

[18] Oracle and/or its affiliates. Interface Stream. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html, 2021.

[19] Oracle and/or its affiliates. Interface Stream, limit operation. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#limit(long), 2021.

[20] Oracle and/or its affiliates. OpenJDK distinct operation source code. http://hg.openjdk.java.net/jdk-updates/jdk11u/file/a30c4bf4e153/src/java.base/share/classes/java/util/stream/DistinctOps.java#l81, 2021.

[21] Oracle and/or its affiliates. Package java.util.stream. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html, 2021.

[22] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. *Haskell 2001*, page 1–233, 2001.

[23] Francisco Ribeiro, João Saraiva, and Alberto Pardo. Java Stream Fusion: Adapting FP Mechanisms for an OO Setting. In *SBLP*, page 30–37, 2019.

[24] Rémi Forax, Vladimir Zakharov, Kevin Bourrillion, Dan Heidinga, Srikanth Sankaran, Andrey Breslav, Doug Lea, Bob Lee, Brian Goetz, Daniel Smith, Samuel Pullara, and David Lloyd. Java Specification Request 335: Lambda Expressions for the Java Programming Language. https://jcp.org/en/jsr/detail?id=335, 2014.

[25] Manu Sridharan and Rastislav Bodík. Refinement-Based Context-Sensitive Points-to Analysis for Java. In *PLDI*, page 387–400, 2006.