Functional Programming in Scala and Java

9 Parallel Streams

9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

Parallel Streams

Parallel Streams support parallel processing

parallelStream(): creates a parallel stream from collections

```
Collection<Long> coll = ...;
coll.parallelStream().reduce(0L, (sum, x) -> (sum + x));
```

parallel(): creates a parallel stream from a sequential stream

```
Arrays.stream(array).parallel().reduce(0L, (sum, x) -> (sum + x));
```

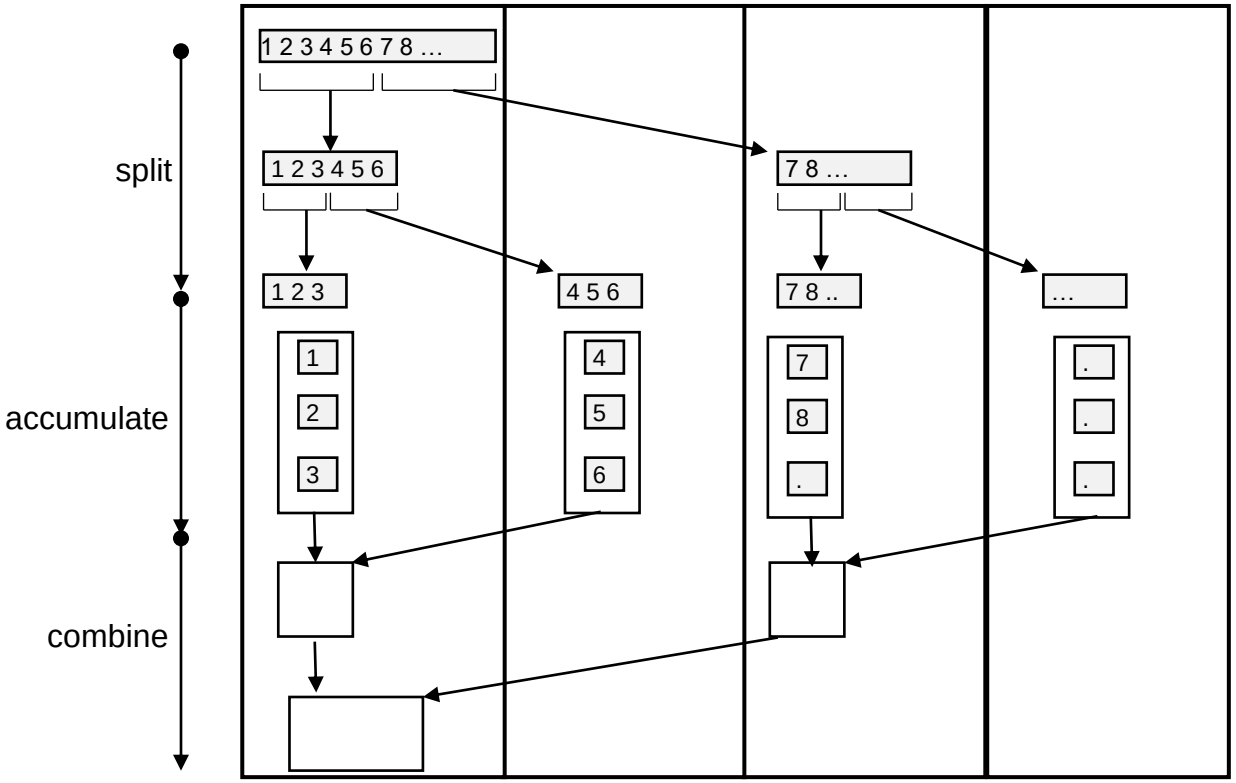
sequential(): change back from parallel to sequential processing

```
coll.parallelStream().map(x -> x * x).sequential().sorted().reduce(0L, (sum, x) -> (sum + x));
```

in case sequential processing is required

Parallel Processing

- Splitting stream in parts and process parts in parallel; then combine partial results
- Parallel processing according to *Divide & Conquer* approach
  - *Split*: divide stream into parts
  - *Accumulate*: Sequential processing for parts
  - *Combine*: Combine results from parts to final result

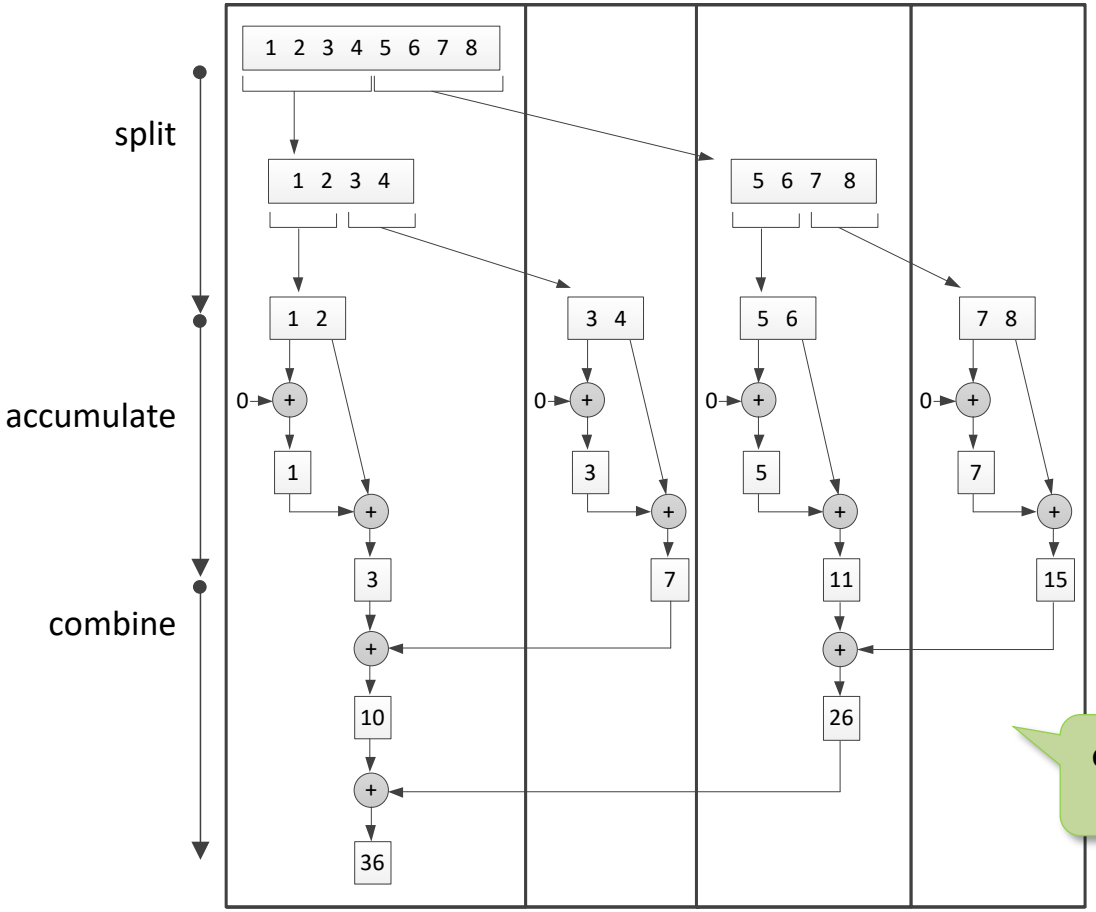


Parallel Processing

Example: reduce

```
<U> U reduce(
    U identity,
    BiFunction<U, ? super T, U> accumulator,
    BinaryOperator<U> combiner
 )
```

```
coll.parallelStream().reduce(
    0L,
    (sum, x) -> (sum + x),
    (sum1, sum2) -> (sum1 + sum2)
);
```



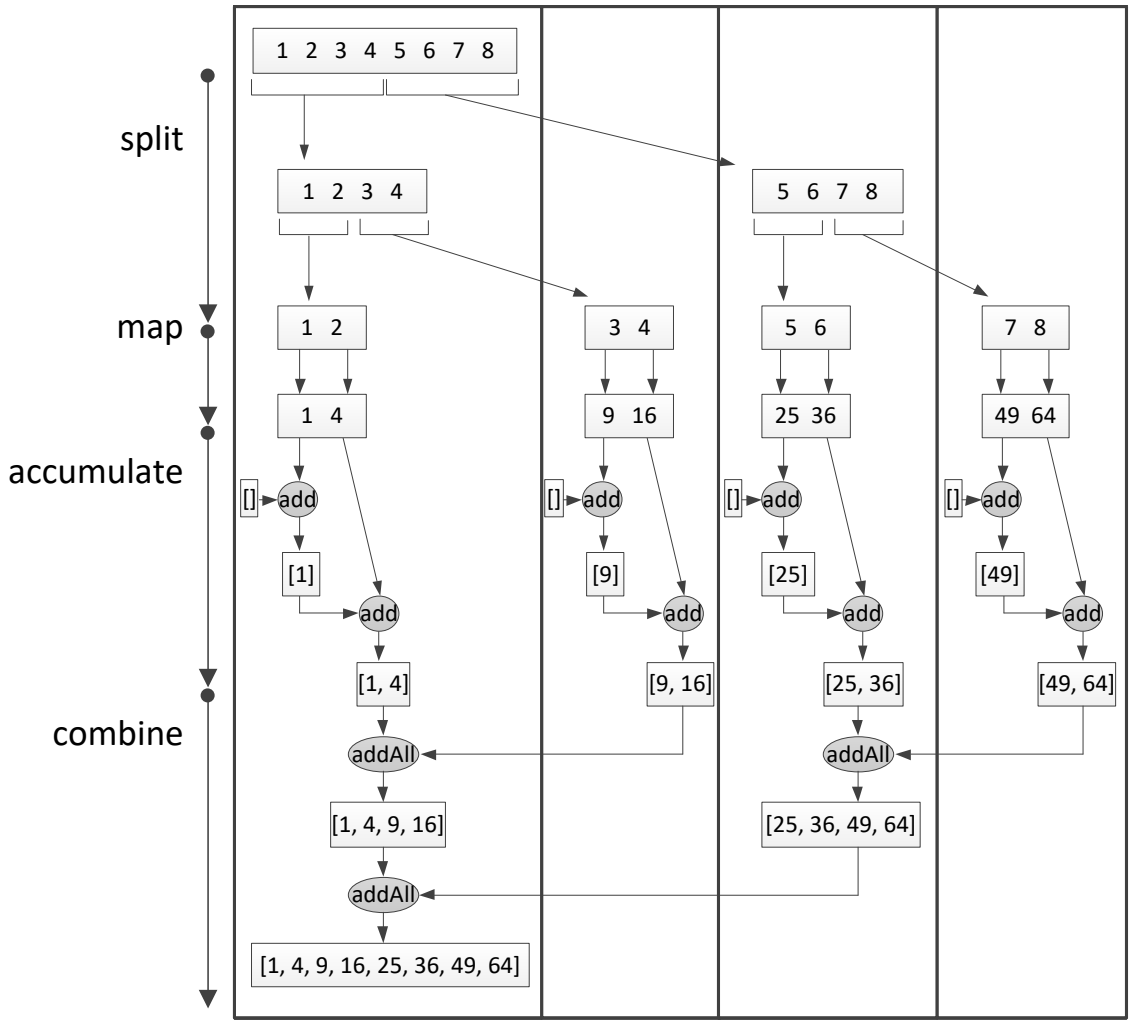
accumulator and **combiner** are pure functions!

Parallel Processing

- Example: collect

```
List<Long> squares = coll.parallelStream()
    .mapToLong(x -> x * x)
    .collect(() -> new ArrayList<Long>(),
             (list, elem) -> list.add(elem),
              (list1, list2) -> list1.addAll(list2));
```

add and **addAll** are not pure, but with controlled side effects!



9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

SPLITERATORS

Splitable iterator

- Interface for parallel processing
  - **Split** for splitting the stream for parallel processing
  - **Iterator** for sequential processing

Tries to do a split; Result is again a Spliterator, which represents next parallel processing.

Estimate number of elements for decision on split

```
public interface Spliterator<T> {

    Spliterator<T> trySplit();                                  Split

    long estimateSize();

    default long getExactSizeIfKnown() {
        return (characteristics() & SIZED) == 0 ? -1L : estimateSize();
    }

    boolean tryAdvance(Consumer<? super T> action);

    default void forEachRemaining(Consumer<? super T> action) {
        do { } while (tryAdvance(action));
    }                                                           iterator

    int characteristics();

    default boolean hasCharacteristics(int characteristics) {
        return (characteristics() & characteristics) == characteristics;
    }
    ...
}
```

For sequential processing. Elements are processed by consumer function action.

For investigating properties of stream

Spliterator: Trace of trySplit

- Trace of **trySplit** for processing 100 elements

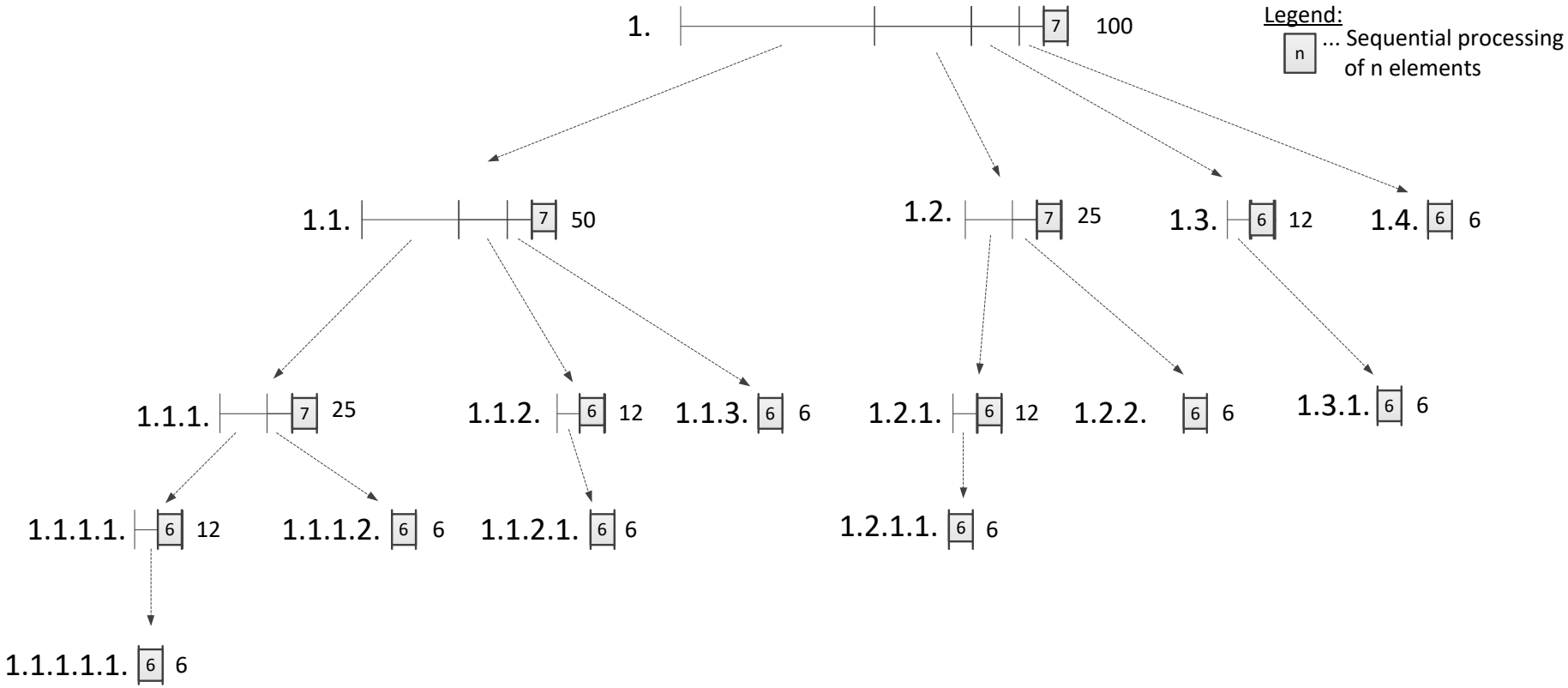
```
list100.parallelStream().reduce("", String::concat);
```

Spliterator	splitted	split size
1.	1.1.	50
1.	1.2.	25
1.1.	1.1.1.	25
1.2.	1.2.1.	12
1.	1.3.	12
1.2.	1.2.2.	6
1.1.1.	1.1.1.1.	12
1.1.	1.1.2.	12
1.1.1.	1.1.1.2.	6
1.2.1.	1.2.1.1.	6
1.	1.4.	6
1.3.	1.3.1.	6
1.1.1.1.	1.1.1.1.1.	6
1.1.2.	1.1.2.1	6
1.1.	1.1.3.	6

Spliterator: Trace of trySplit

- Trace of **trySplit** for processing 100 elements

```
list100.parallelStream().reduce("", String::concat);
```



Parallel Execution by Fork-Join

- Fork-Join works based on Divide&Conquer principle
  - **RecursiveTask**s recursively split subtasks
  - which are executed by Fork-Join thread pool
- Principle of RecursiveTasks

```
public class MyRecursiveTask<T> extends RecursiveTask<T> {

   @Override
   protected T compute() {
      if (problem small ) {
         result = solve problem sequentially
         return result;
      }

      split sub-problem and
      create subtask for sub-problem
      send task to fork-join pool
      ... possibley more splits

      join with subtasks for partial solutions

      result = combine partial solutions
      return result;
   }
```

Is problem small enogh, then solve it sequentially

Split problem into subproblems and send it to Fork-Join thread pool

Wait for solution of subproblems

Combine subproblems and return total solution

Simulating Parallel Reduce (1/2)

- Program simulates parallel execution of **reduce** using Fork-Join pool

Corresponds to: `coll.parallelStream().reduce(0L, (sum, x) -> (sum + x),(sum1, sum2) -> (sum1 + sum2));`

```
class ReduceRecursiveTask<T> extends RecursiveTask<T> {
    private static final int THRESHOLD = 7;
    final Spliterator<T> spliterator;
    final T identity;
    final BinaryOperator<T> accu;
    final BinaryOperator<T> comb;

    public ReduceRecursiveTask(Spliterator<T> spliterator, T identity, BinaryOperator<T> accu,  BinaryOperator<T> comb) {
        super();
        this.spliterator = spliterator;
        this.identity = identity;
        this.accu = accu;
        this.comb = comb;
    }

    T result;
    @Override
    protected T compute() {
        // accumulate sequentially
        long est = spliterator.estimateSize();
        if (est <= THRESHOLD) {
            result = identity;
            spliterator.forEachRemaining((T x) -> result = accu.apply(result, x));
            return result;
        }
        // ... continued on next page ...
```

Is estimated size smaller as threshold, accumulate elements

Simulating Parallel Reduce (2/2)

```
        // continued
        List<ReduceRecursiveTask<T>> subTasks = new ArrayList<ReduceRecursiveTask<T>>();
        while (spliterator.estimateSize() > THRESHOLD) {
            Spliterator<T> split = spliterator.trySplit();
            ReduceRecursiveTask<T> subTask = new ReduceRecursiveTask<T>(split, identity, accu, comb);
            subTasks.add(subTask);
            subTask.fork();

        // accumulate rest of elements of this splitertor
        result = identity;
        spliterator.forEachRemaining((T x) -> result = accu.apply(result, x));

        // join and combine
        for (ReduceRecursiveTask<T> subTask: subTasks) {
            T subResults = subTask.join();
            try {
                result = comb.apply(result, subResult);
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace(System.err);
            }
        }
        return result;
    }
}
```

Split and fork as long as estimated size of this greater than threshold

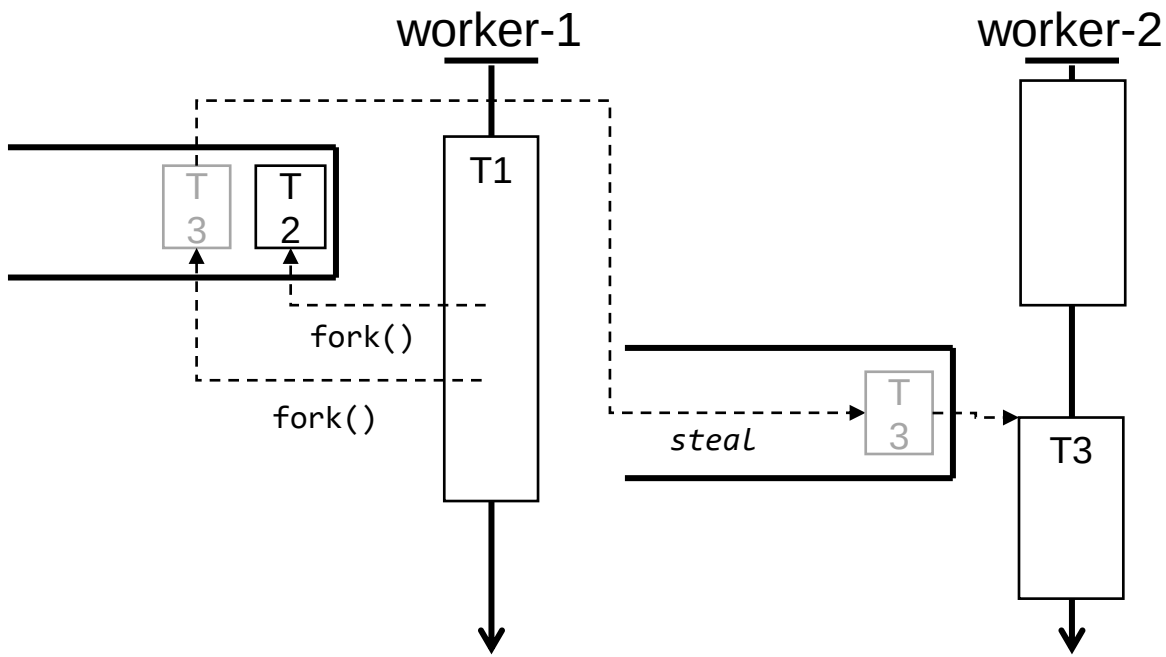
Accumulate rest of elements of this spliterator

Join with all subtasks and combining partial results

```
public static void main(String[] args) {
    ReduceRecursiveTask<Long> task =
                new ReduceRecursiveTask<String>(coll.spliterator(), 0L, (sum, x) -> sum + x, sum1, sum2) -> sum1 + sum2);
    ForkJoinPool executor = new ForkJoinPool();
    Long result = executor.invoke(task);
}
```

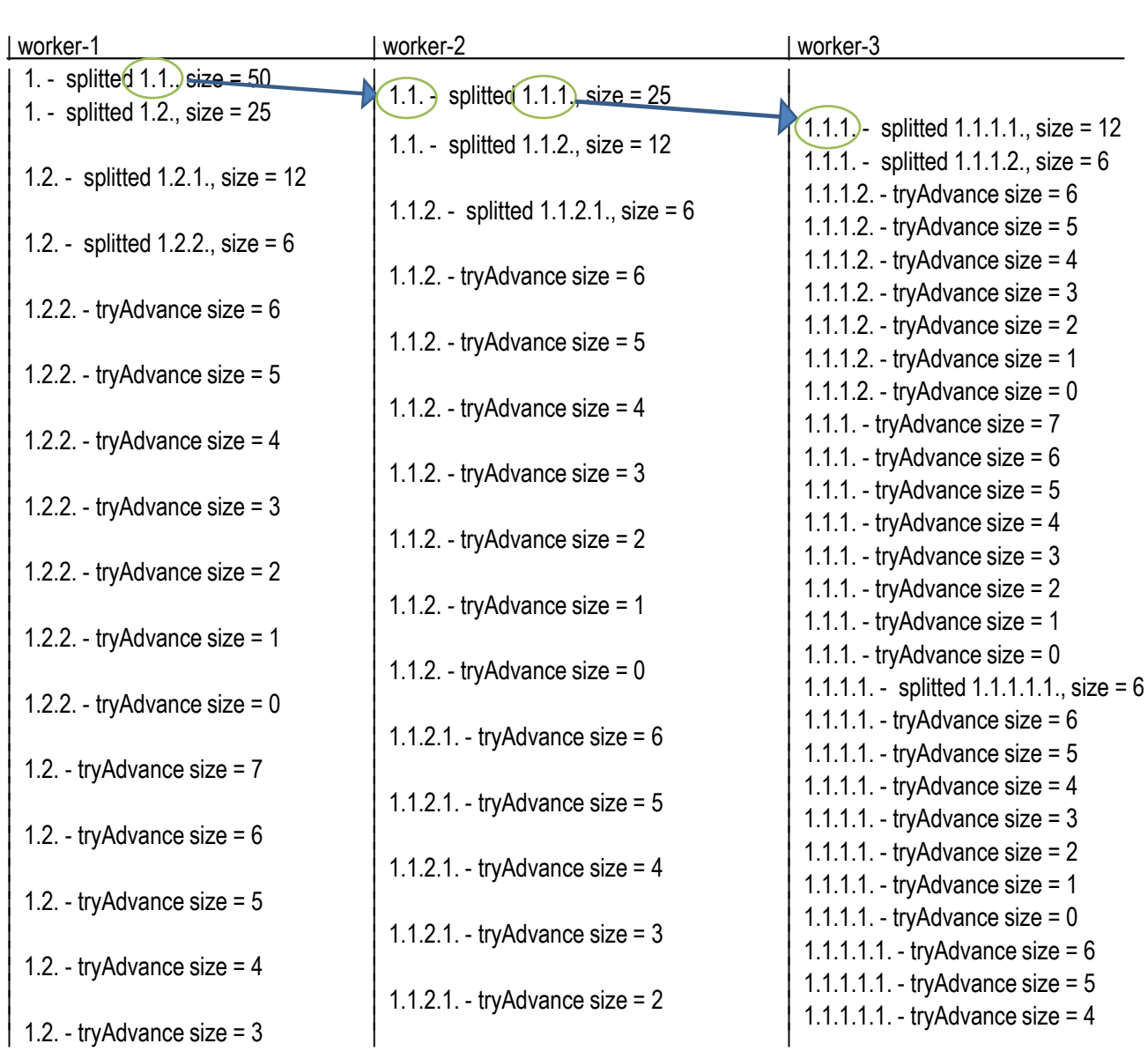
Fork-Join Pool: Work Stealing

- Fork-Join Pool
  - fixed number of *worker* threads (dependent of number of cores, e.g. 3)
  - each thread has queue of tasks
    - **fork()** creates new task and puts it into queue of current threads
  - then fork-join pool works with *work stealing*
    - idle threads „steal“ tasks from queues of other threads



In this way worker threads are kept busy!

Fork-Join Pool: Example Trace



Work stealing:

Controlling Parallel Execution

- Parallel execution by default ForkJoinPool

```
ForkJoinPool.commonPool()
```

Default configuration of number of worker threads based on number of cores

```
int parallelism = ForkJoinPool.commonPool().getCommonPoolParallelism();
```

➡ 3 on my I5-5300U

```
int parallelism = ForkJoinPool.getCommonPoolParallelism();
```

Change of configuration of common pool

- by VM argument

```
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

- using properties

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "5");
```

Remark: Usually standard settings working well!

9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

Parallel Processing and Side Effects

- Pure functions for parallel processing always valid !
- Arbitrary side effects may result in severe problems!

Example: Summing in global variable

```
static long sumSeq = 0;
static long sumPar = 0;

public static void main(String[] args) {

    final Collection<Long> coll = createArrayList(2000);

    sumSeq = 0;                                          sequential
    coll.stream().forEach( x -> sumSeq = sumSeq + x );

    sumPar = 0;                                          parallel
    coll.parallelStream().forEach( x -> sumPar = sumPar + x );

    System.out.format("%n%nSequential sum %8d", sumSeq);
    System.out.format("%nParallel sum   %8d", sumPar);

}
```

Results of several runs

```
Sequential sum   1999000
Parallel sum     1531445

Sequential sum   1999000
Parallel sum     1125720

Sequential sum   1999000
Parallel sum     1615340

Sequential sum   1999000
Parallel sum     1418349

Sequential sum   1999000
Parallel sum     1993167

...
```

Incorrect results for parallel processing due to **data races** !

Parallel Processing and Side Effects

- Synchronization required

Example: Summing using **AtomicLong**

```
static long sumSeq;
static AtomicLong sumPar;

public static void main(String[] args) {

   final Collection<Long> coll = createArrayList(2000);

   sumSeq = 0;                                    sequentiell
   coll.stream().forEach(x -> seqSum = seqSum + x);

   sumPar = new AtomicLong(0);                    parallel
   coll.parallelStream().forEach(x -> sumPar.addAndGet(x));

   System.out.format("%n%nSequential sum %8d", sumSeq);
   System.out.format("%nParallel sum   %8d", sumPar.get());
}
```

Results of several runs

```
Sequential sum    1999000
Parallel sum      1999000

Sequential sum    1999000
Parallel sum      1999000

Sequential sum    1999000
Parallel sum      1999000

Sequential sum    1999000
Parallel sum      1999000

Sequential sum    1999000
Parallel sum      1999000

...
```

Results are **correct** but **synchronization** required !

Parallel Processing and Side Effects

- Arbitrary side effects may result in severe problems!

Example: **ArrayList** for accumulating results

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

    final Collection<Long> coll = createArrayList(N);

    List<Long> resultsSeq = new ArrayList<Long>();
    coll.stream().forEach(x -> resultsSeq.add(x * x));

    List<Long> resultsPar = new ArrayList<Long>();
    coll.parallelStream().forEach(x -> resultsPar.add(x * x));

}
```

sequential

parallel

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
        at java.lang.reflect.Constructor.newInstance(Unknown Source)
        at java.util.concurrent.ForkJoinTask.getThrowableException(Unknown Source)
        at java.util.concurrent.ForkJoinTask.reportException(Unknown Source)
        at java.util.concurrent.ForkJoinTask.invoke(Unknown Source)
        at java.util.stream.ForEachOps$ForEachOp.evaluateParallel(Unknown Source)
        at java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateParallel(Unknown Source)
        at java.util.stream.AbstractPipeline.evaluate(Unknown Source)
        at java.util.stream.Referen...ipeline.forEach(Unknown Source)
        at java.util.stream.Refere...Head.forEach(Unknown Source)
        at parstr.intro.S...
```

Parallel writes to list result in **exception** !

Stateful computations

- Stateful computations which are okay in sequential computations completely fail in parallel compuations

Example: Filter elements which are smaller than the current maximum (maximum changes)

```
static volatile int max

public static void main(String[] args) {
    final List<Integer> coll = createArrayList(100);
    max = 0;
                                                    sequential
    List<Integer> filteredSeq =
            coll.stream()
                .filter(x -> {
                    if (x > max) {
                        max = x;
                        return true;
                    } else {
                        return false;
                    }
                })
                .collect(ArrayList<Integer>::new,
                         ArrayList<Integer>::add,
                         ArrayList<Integer>::addAll);

    System.out.println(filteredSeq.size());
```

Result:

```
100
```

```
    max = 0;                                                    parallel

    List<Integer> filteredPar =
            coll.parallelStream()
                .filter(x -> {
                    if (x > max) {
                        max = x;
                        return true;
                    } else {
                        return false;
                    }
                })
                .collect(ArrayList<Integer>::new,
                         ArrayList<Integer>::add,
                         ArrayList<Integer>::addAll);

    System.out.println(filteredPar.size());
```

Results wrong and non-deterministic:

```
26
19
...
```

Function parameters: Properties

Computations have to be independent on execution order and if splitted or not splitted

- Special requirements on accumulator and combiner functions
  - stateless, non-interferring and associative

Example **reduce**:

```
<U> U reduce(  U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner )
```

- **identity**: is identity element for reduce and combiner

```
accumulator(identity, a) == a
combiner(identity, a) == a
reduce(emtpy-stream) = identity
combiner(a, identity) == a
```

cf. Monoids

- **associativity** of accumulator and combiner

```
accumulator ( accumulator(a, b), c)   =   (accumulator a, (accumulator(b, c))
combiner (combiner(a, b), c)   =   (combiner a, (combiner(b, c))
```

- **compatibility** of accumulate and combine

```
combiner(u, accumulator(identity, t)) == accumulator(u, t)
```

analogous for collect

Function parameters: Properties

<U> U reduce(U **identity**, BiFunction<U, ? super T, U> **accumulator**, BinaryOperator<U> **combiner**)

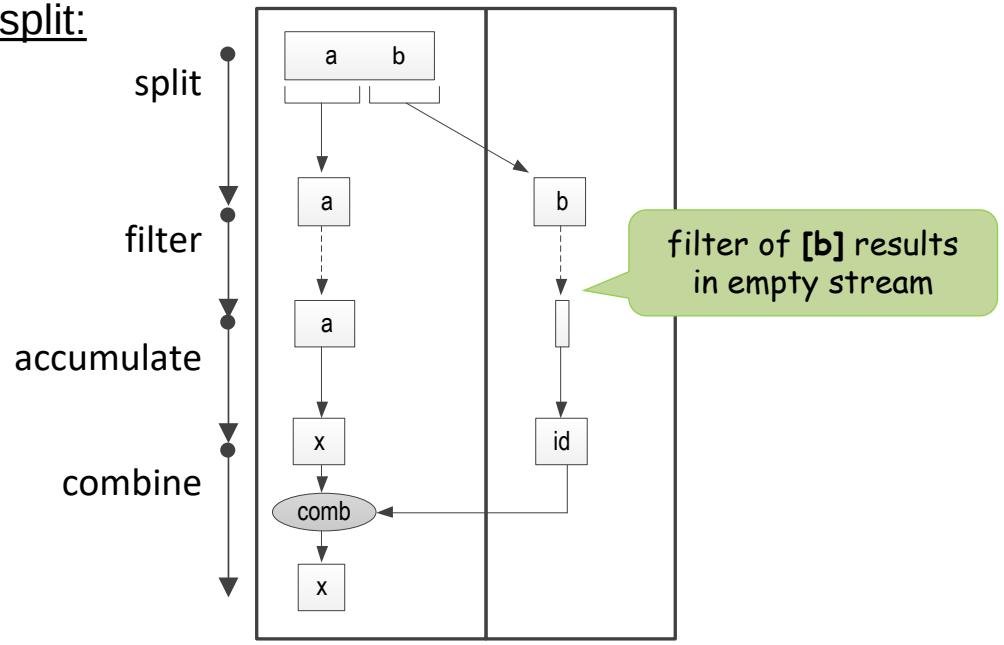
combine (identity, a) = a

combine (a, identity) = a

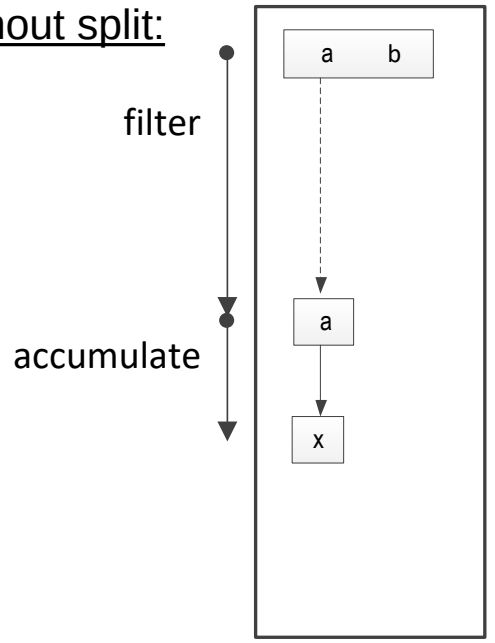
reduce(emtpy-stream) = identity

```
List.of(a, b).parallelStream().filter(..).reduce(id, acc, comb)
```

with split:



without split:



Function parameters: Properties

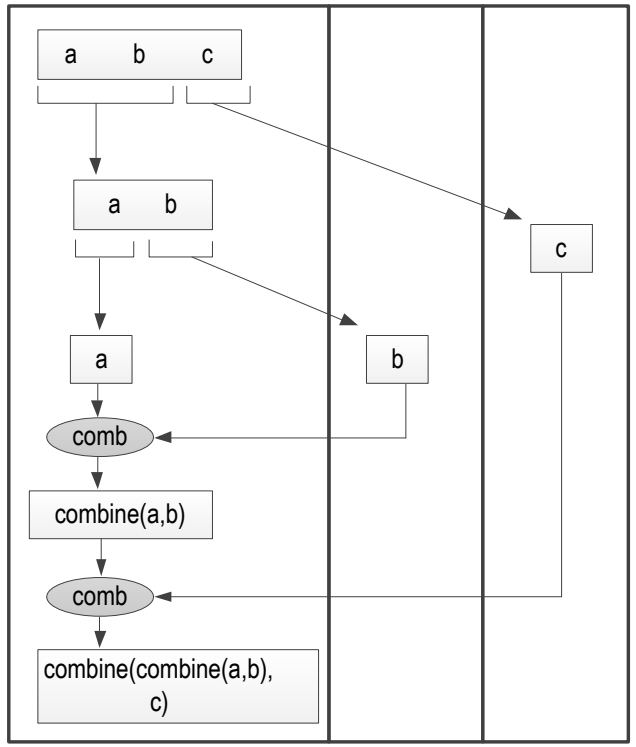
<U> U reduce(U **identity**, BiFunction<U, ? super T, U> **accumulator**, BinaryOperator<U> **combiner**)

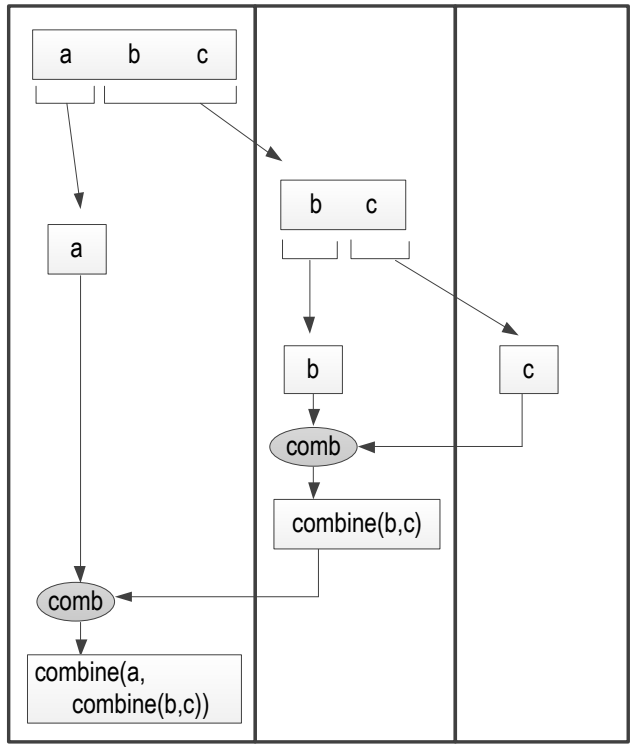
- Associativity of **combine**

combine (combine(a, b), c) = combine (a, combine(b, c))

i.e., order of operations do not matter !

```
List.of(a, b, c).parallelStream().reduce(id, acc, comb)
```





different orders of operations due to different splits

Function parameters: Properties

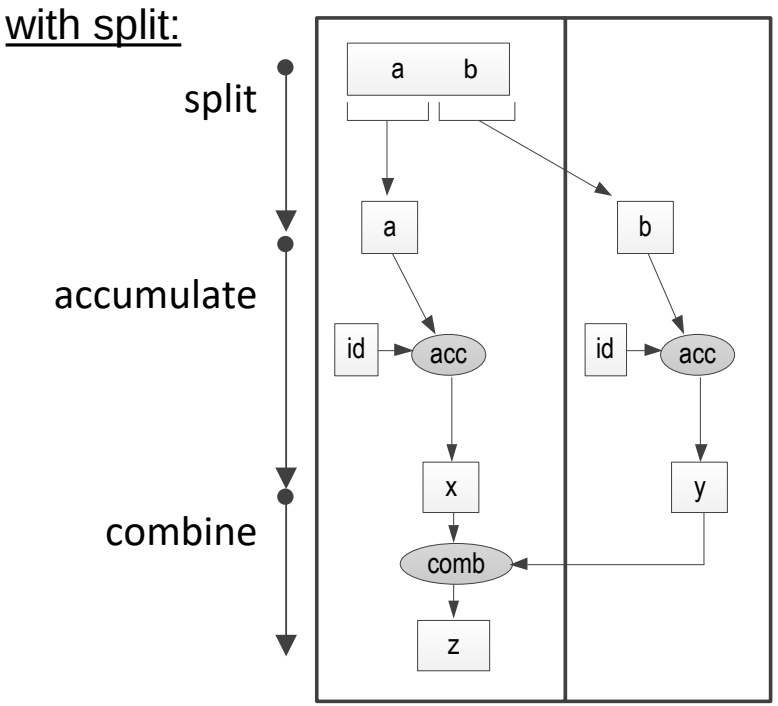
<U> U reduce(U **identity**, BiFunction<U, ? super T, U> **accumulator**, BinaryOperator<U> **combiner**)

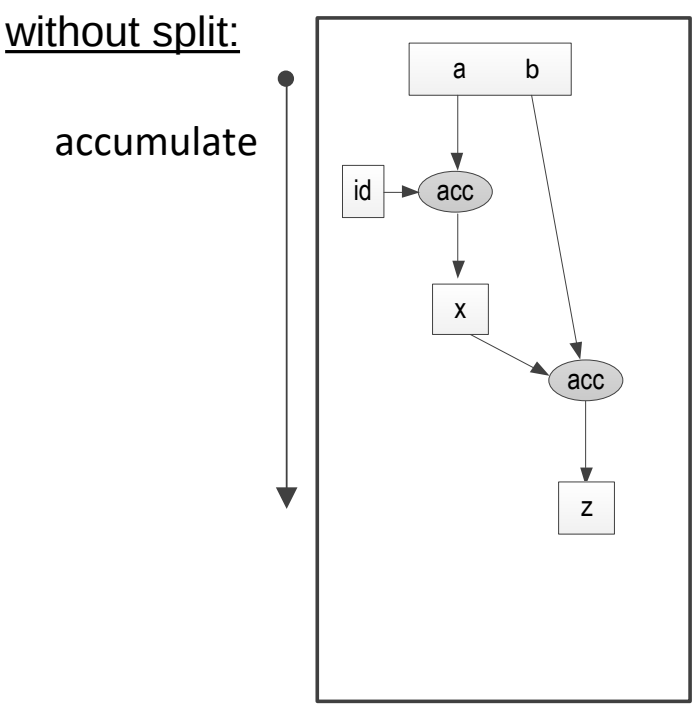
- Compatibility of **accumulate** and **combine**

comb(acc(id, a), acc(id, b)) = acc(acc(id, a), b)

Result is same if split and combine or no split and only accumulate

```
List.of(a, b).parallelStream().reduce(id, acc, comb)
```





9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

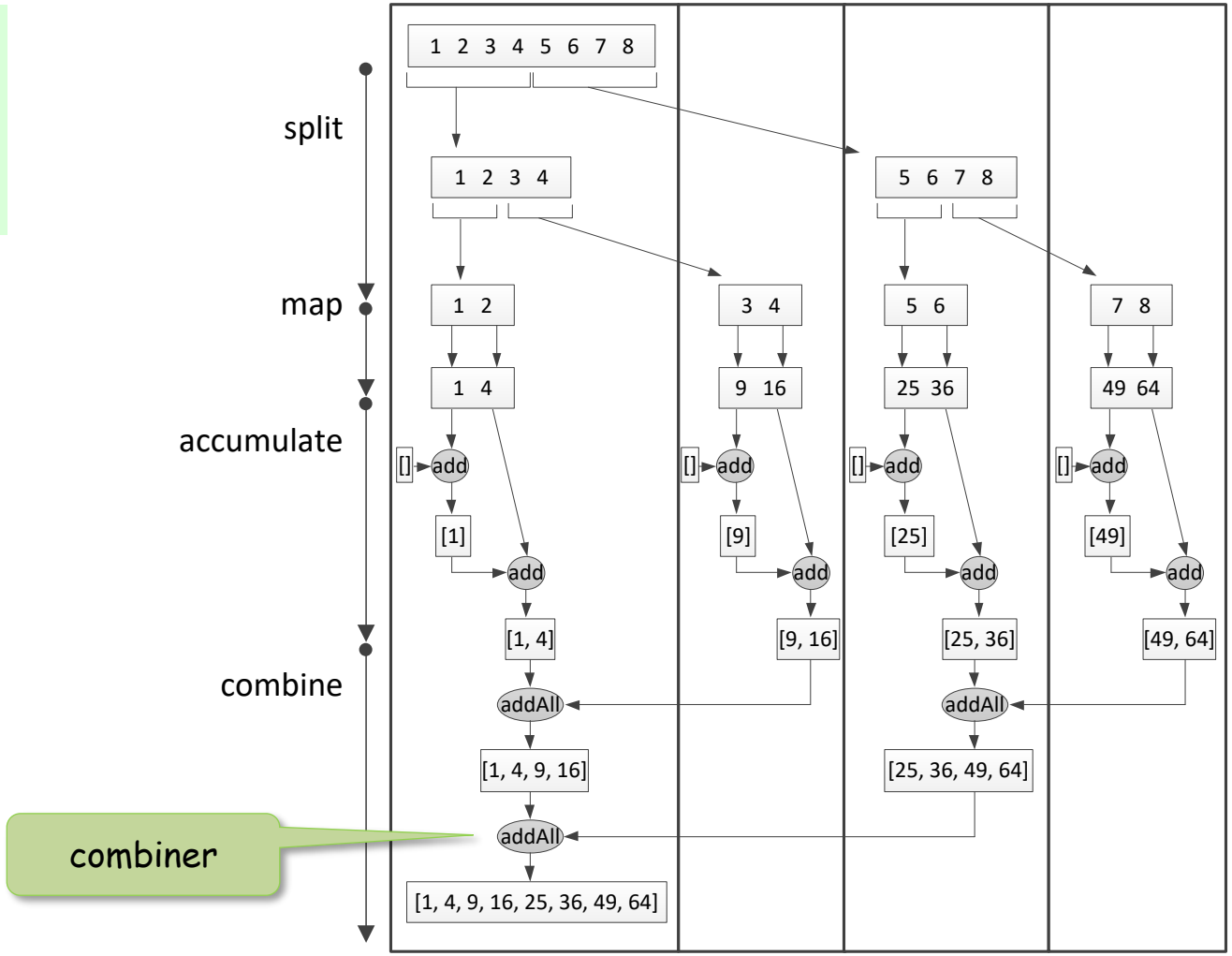
Parallel Collect

■ Parallel collect by combining collections

```
<R> R collect(
   Supplier<R> supplier,
   BiConsumer<R, ? super T> accumulator,
   BiConsumer<R, R> combiner
 )
```

Example:

```
List<String> wordList =
    words.collect(
          () -> new ArrayList<String>(),
          (list, w) -> list.add(w),
          (list1, list2) -> list1.addAll(list2)
    );
```



Parallel Collect with Concurrent Maps

Collectors with **concurrent maps**

- all elements into **single concurrent map**
- work without combiner step

may be more efficient
(see Section on Performance)

Methods of **Collectors** for creating Collector using concurrent maps

■ toConcurrentMap

```
public static <T, K, U> Collector<T, ?, ConcurrentMap<K,U>>
    toConcurrentMap( Function<? super T, ? extends K> keyMapper,
                     Function<? super T, ? extends U> valueMapper )
```

■ groupingByConcurrent

```
public static <T, K> Collector<T, ?, ConcurrentMap<K, List<T>>>
    groupingByConcurrent( Function<? super T, ? extends K> classifier )
```

9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

Run Time

Question: When does using parallel streams pay off?

Answer: Cannot be answered in general as it is dependent on several factors!

But:

- General guidelines
- Specific indicators

General guidelines

1) **Effect of parallelization is greater when**
   - ☐ data set is big
   - ☐ run time for processing single elements is high

Commonplace!

N	... number elements
Q	... time for processing of single data element
*N * Q*	... total processing, potential for parallelization
P	... parallel processing units

minimal runtime possible: **N * Q / P**

2) **Processing should be without I/O and without blocking calls**

Experiments

- Experimental settings
  - Microbenchmark Harness (JHM)
  - Dual Core Intel i5-8350U 1,7GHz and 16GB memory
  - Java-System Version 8

- Experiments will show effects of
  - size of data source
  - type of data source
  - intermediate operations
  - terminal operations
  - types of collections in collect
  - boxing
  - collectors with map and combiner vs. collectors with concurrent map

Experiment 1: Size of Data Source

- IntStream from int-Array

```
int[] intArr
```

- Building sum of elements
  - ☐ ➔ Overhead for parallelization in comparison to operation is high

simple operation

sequential:

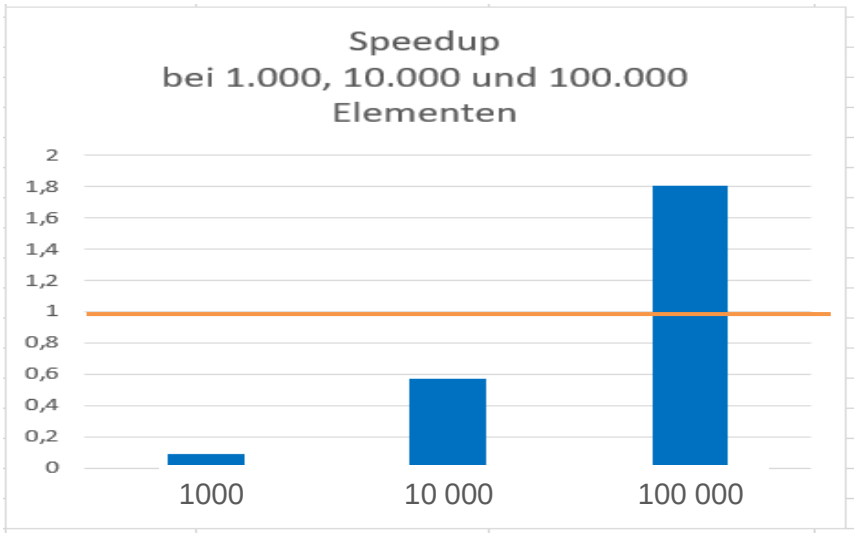
```
Arrays.stream(intArr).reduce((x, y) -> x + y);
```

parallel:

```
Arrays.stream(intArr).parallel().reduce((x, y) -> x + y);
```

- Comparison with array sizes: 1000, 10 000, 100 000
- Measuring run time and achieved speedup

Speedup
(> 1 is better)



Needs 100.000 elements for gaining speedup!

Experiment 2: Type of Data Source

- Comparing Arrays, ArrayList, LinkedList

```
Integer[] integerArr
```

```
ArrayList<Integer> integerArrayList
```

```
LinkedList<Integer> IntegerLinkedList
```

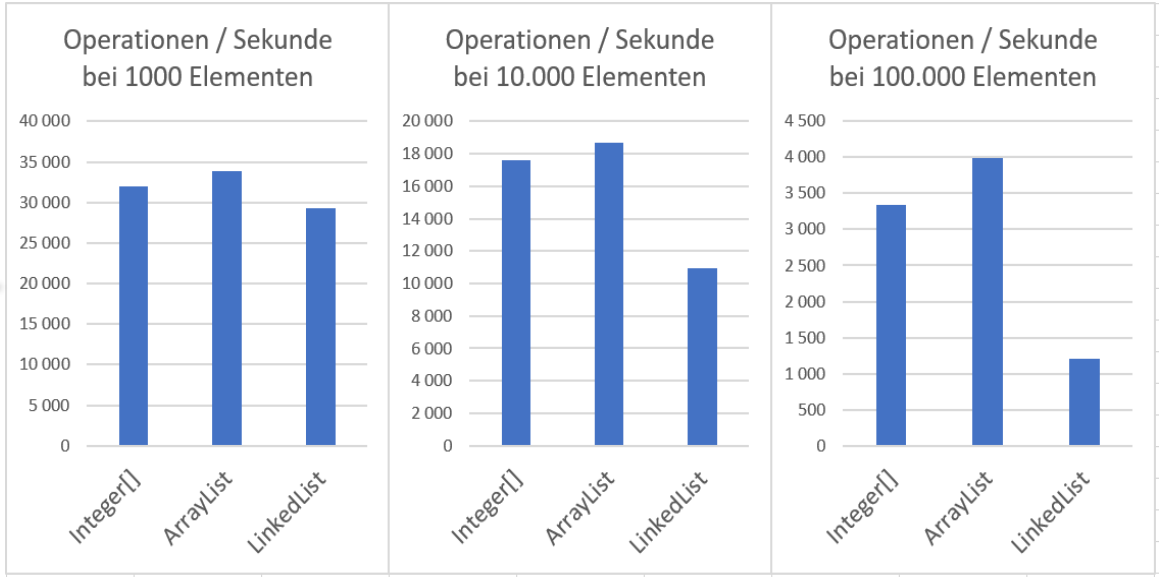
- Building sum of elements

```
Arrays.stream(integerArr).parallel().reduce((x, y) -> x + y);
integerArrayList.parallelStream(intArr).reduce((x, y) -> x + y);
integerLinkedList.parallelStream(intArr).reduce((x, y) -> x + y);
```

- Comparison of run time for different types of data sources

Operations / seconds
(higher is better)

Arrays good
ArrayList good
LinkedList bad



Experiment 3: Different Collections

- Comparing ArrayList, LinkedList, HashSet, TreeSet

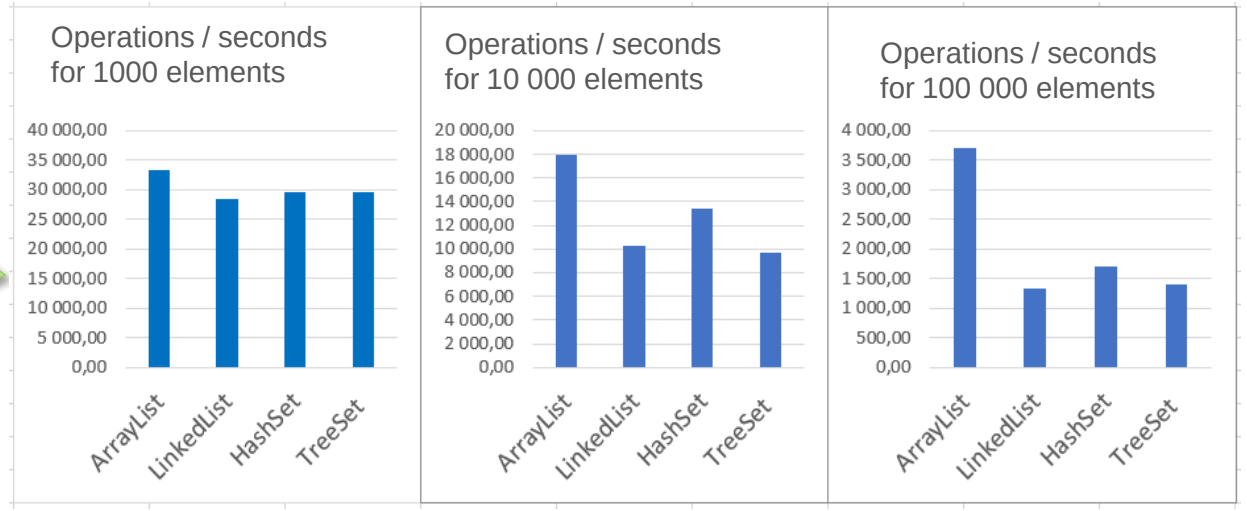
```
ArrayList<Integer> integerArrayList
LinkedList<Integer> IntegerLinkedList
HashSet<Integer> IntegerHashSet
TreeSet<Integer> IntegerTeeSet
```

- Building sum of elements

- Comparison of run time for parallel execution for different collections

Operations / seconds
(higher is better)

ArrayList good
LinkedList bad
HashSet bad
TreeSet bad



Experiment 4: range versus iterate

- Comparing int-Array, range und iterate

```
int[] intArr;
Arrays.stream(intArr)
```

```
IntStream.range(0, size)
```

```
IntStream.iterate(0, x -> x + 1).limit(size)
```

- Building sum of elements
- Comparison of run time for parallel execution for int-Array, range und iterate

Operations / seconds
(higher is better)

int[] and range very good

iterate extremly bad (because is inherently sequential)

Experiment 5: Processing Elements

- **map** of elements

```
Arrays.stream(intArr).map(x -> proc(x, n)).reduce((x, y) -> x + y);
```

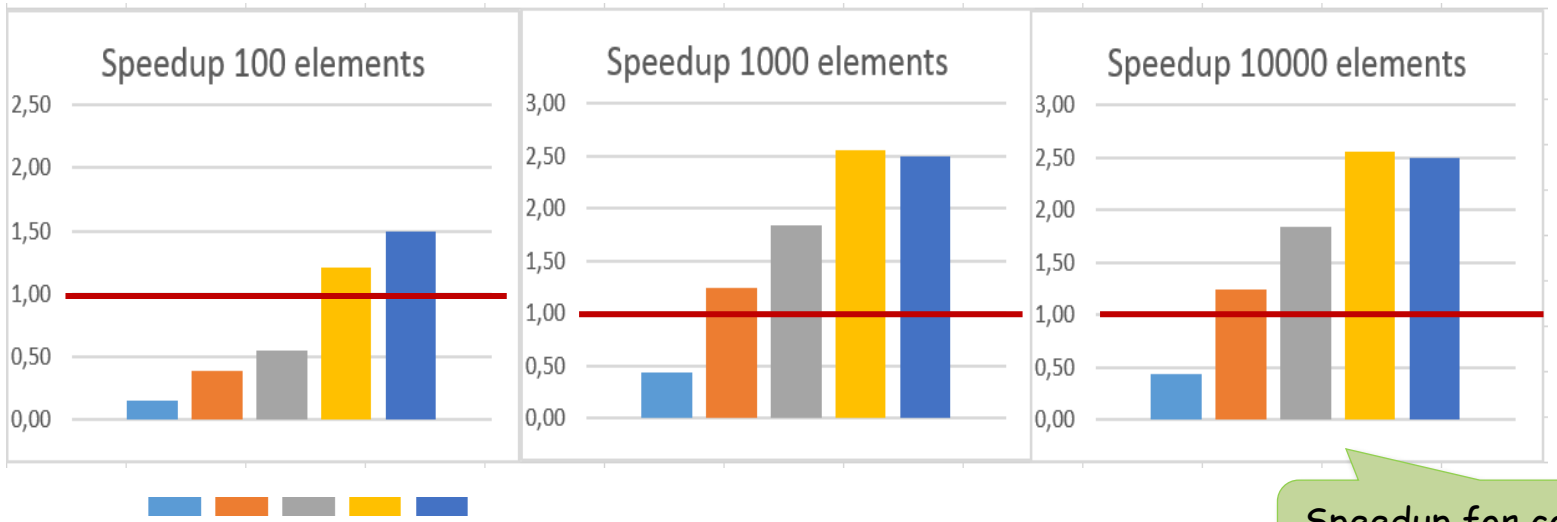
- with map function with different complexity of mapping operation

```
private static final int proc(int x, int n) {
  for (int i = 0; i < n; i++) {
    if (x % 2 == 0) x = x + i; else x = x - i;
  }
  return x;
}
```

≈ 4 Operations per loop iteration

- 2, 10, 20, 50, 100 loop iterations (compared to sequential stream processing)

Speedup
(> 1 is better)



Speedup for complex operations significant!

Experiment 6: Collect Operations

- Collect operations with different result collections

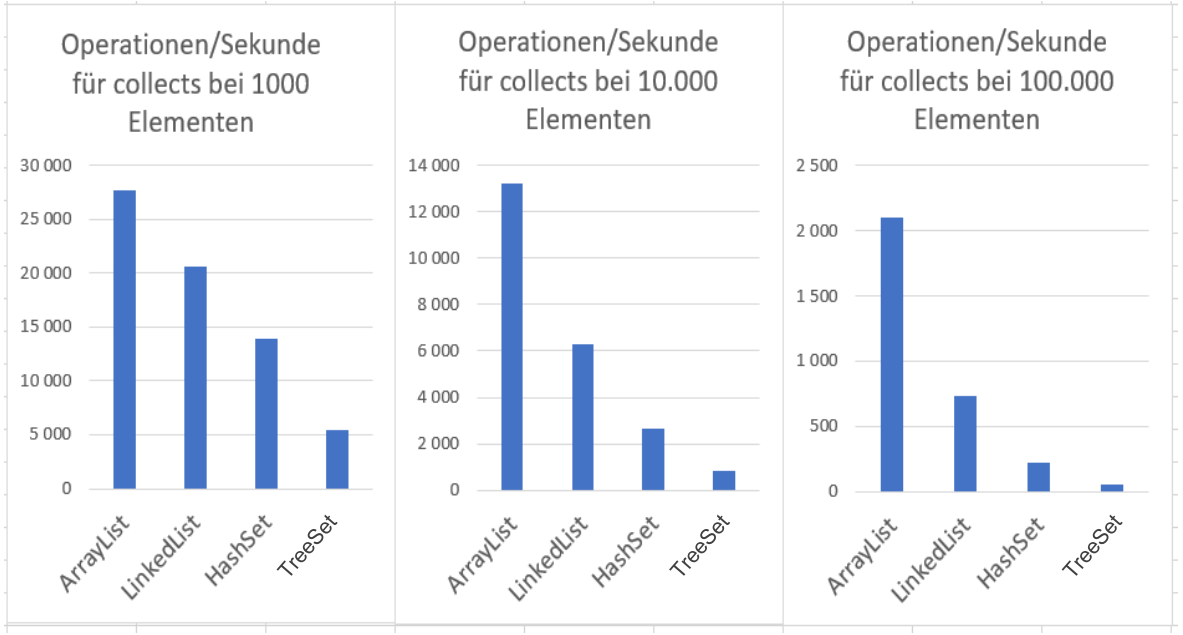
```
collection.parallelStream().collect(Collectors.toCollection(() -> new ArrayList<Integer>()));
collection.parallelStream().collect(Collectors.toCollection(() -> new LinkedList<Integer>()));
collection.parallelStream().collect(Collectors.toCollection(() -> new HashSet<Integer>()));
collection.parallelStream().collect(Collectors.toCollection(() -> new TreeSet<Integer>()));
```

  - where add operations have different complexity

- Comparing run time

Operations / seconds
(higher is better)

Result collection has significant impact on run time!



Experiment 7: findFirst versus findAny

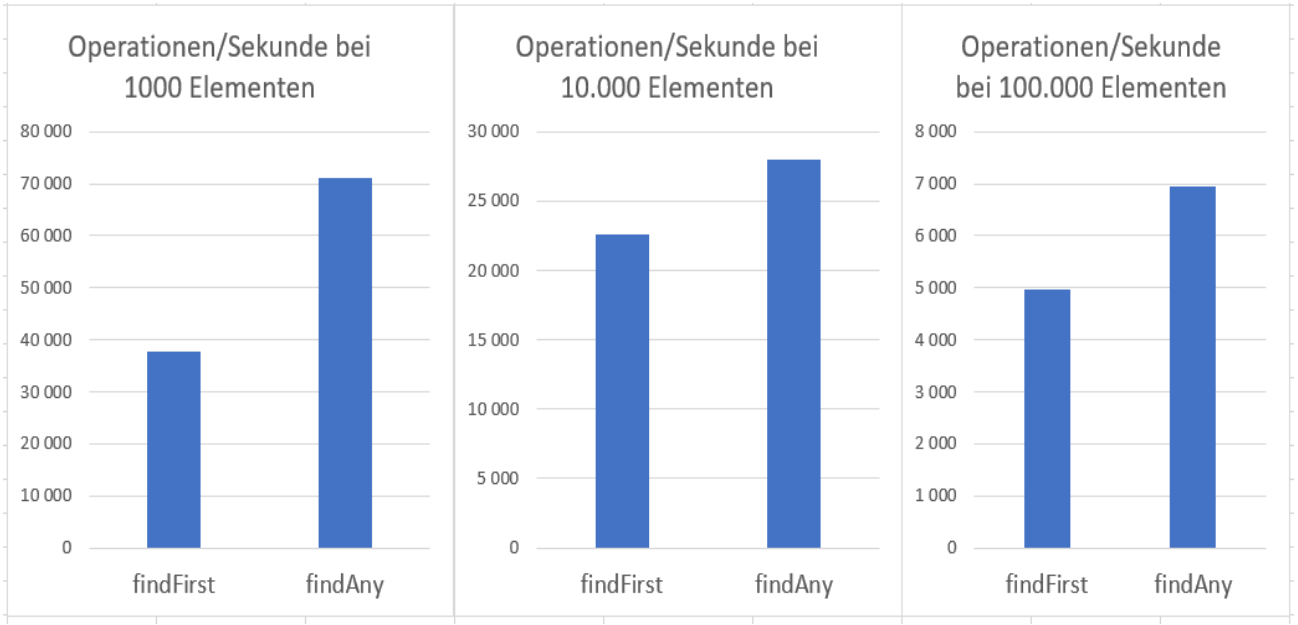
- Comparing **findFirst** and **findAny** in parallel execution
  - Looking for a negative element when negative elements randomly distributed within collections

```
collection.parallelStream().filter(x -> x < 0).findFirst();
collection.parallelStream().filter(x -> x < 0).findAny();
```

- Operations / seconds for 1000, 10 000, 100 000 elements

Operations / seconds
(higher is better)

findAny faster



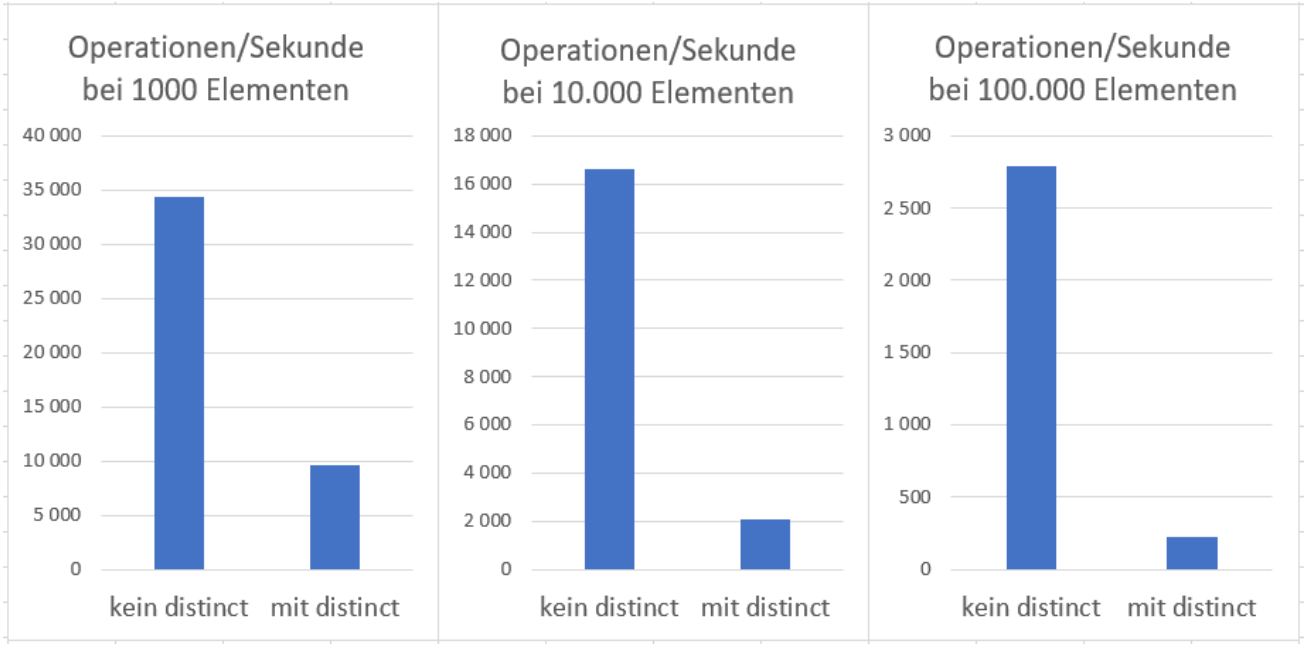
Experiment 9: Effect of distinct

- distinct removes duplicated elements
- Comparing run time with and without **distinct** in parallel execution

```
collection.parallelStream().filter(x -> x < 0).reduce((x, y) -> x + y);
collection.parallelStream().unordered().filter(x -> x < 0).distinct().reduce((x, y) -> x + y);
```

Operations / seconds
(higher is better)

stateful operation
distinct very bad



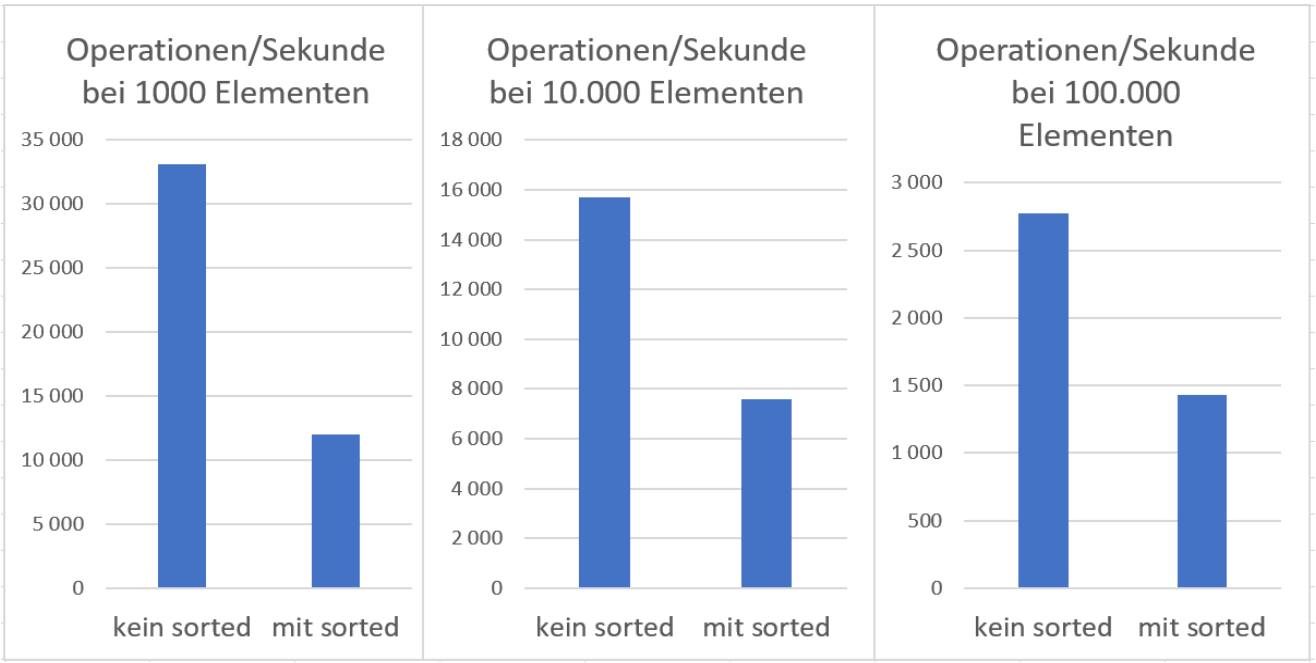
Experiment 10: Effect of sorted

- sorted sorts elements in stream
- Comparing run time with and without sort in parallel execution

```
collection.parallelStream().filter(x -> x < 0).reduce((x, y) -> x + y);
collection.parallelStream().unordered().filter(x -> x < 0).sorted().reduce((x, y) -> x + y);
```

Operations / seconds
(higher is better)

stateful operation
sorted bad



Experiment 11: Effect of Boxing

- Comparing Stream operations with **int** und **Integer** values

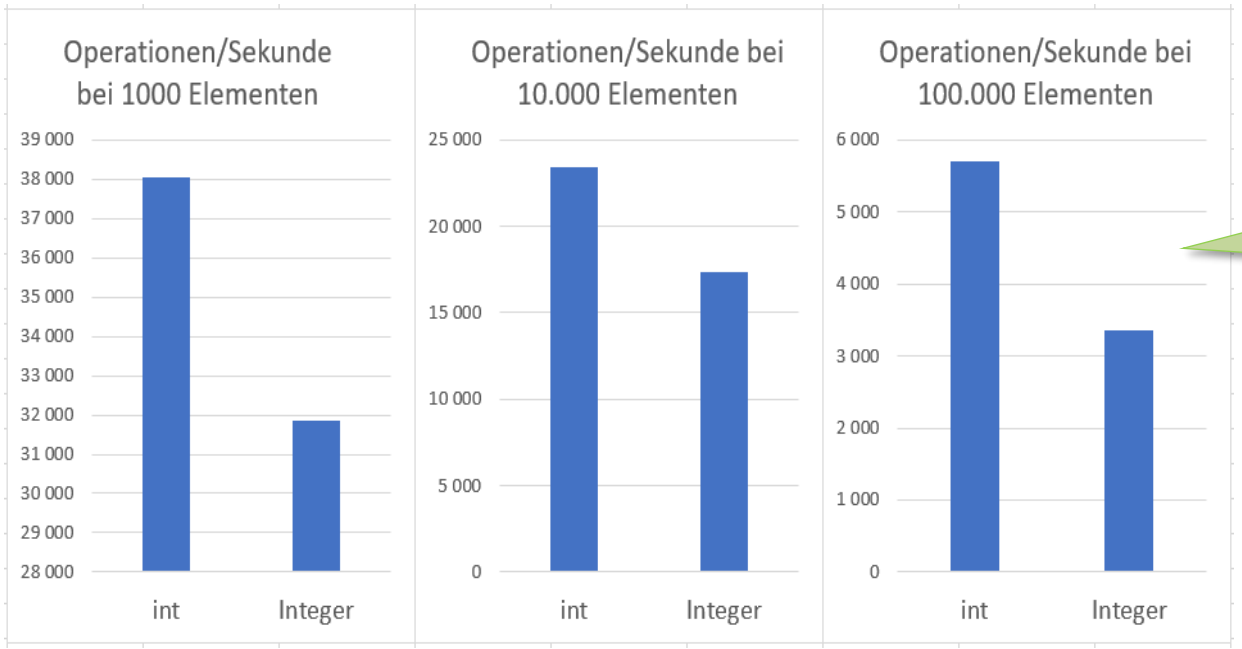
```
int[] intArray = ...
OptionalInt ox = Arrays.stream(intArray).parallel().reduce((x, y) -> x + y);
```

Works with IntStream

```
Integer[] integerArray = ...
Optional<Integer> ox = Arrays.stream(integerArray).parallel().reduce((x, y) -> x + y);
```

Works with Stream<Integer>

Operations / seconds
(higher is better)



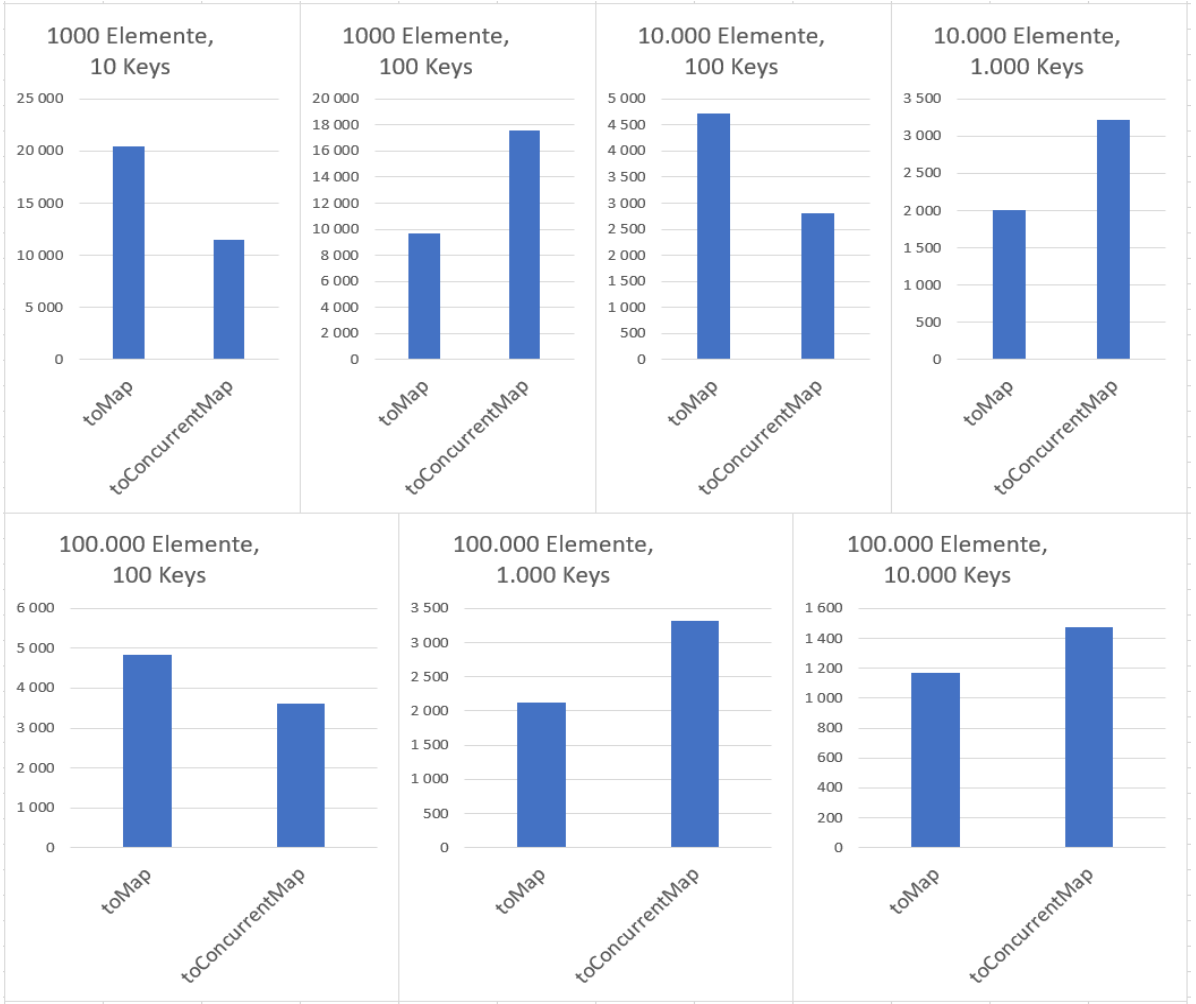
overhead through boxing is significant

Experiment 12: toMap vs. toConcurrentMap

```
Map<Integer, Integer> m =
   integerArrList.parallelStream() .collect (Collectors.toMap(x -> x % N, x -> 1, (s, z) -> s + z));
Map<Integer, Integer> m =
   integerArrList.parallelStream().collect (Collectors.toConcurrentMap(x -> x % N, x -> 1, (s, z) -> s + z));
```

Operations / seconds
(higher is better)

toConcurrent only for maps with many entries

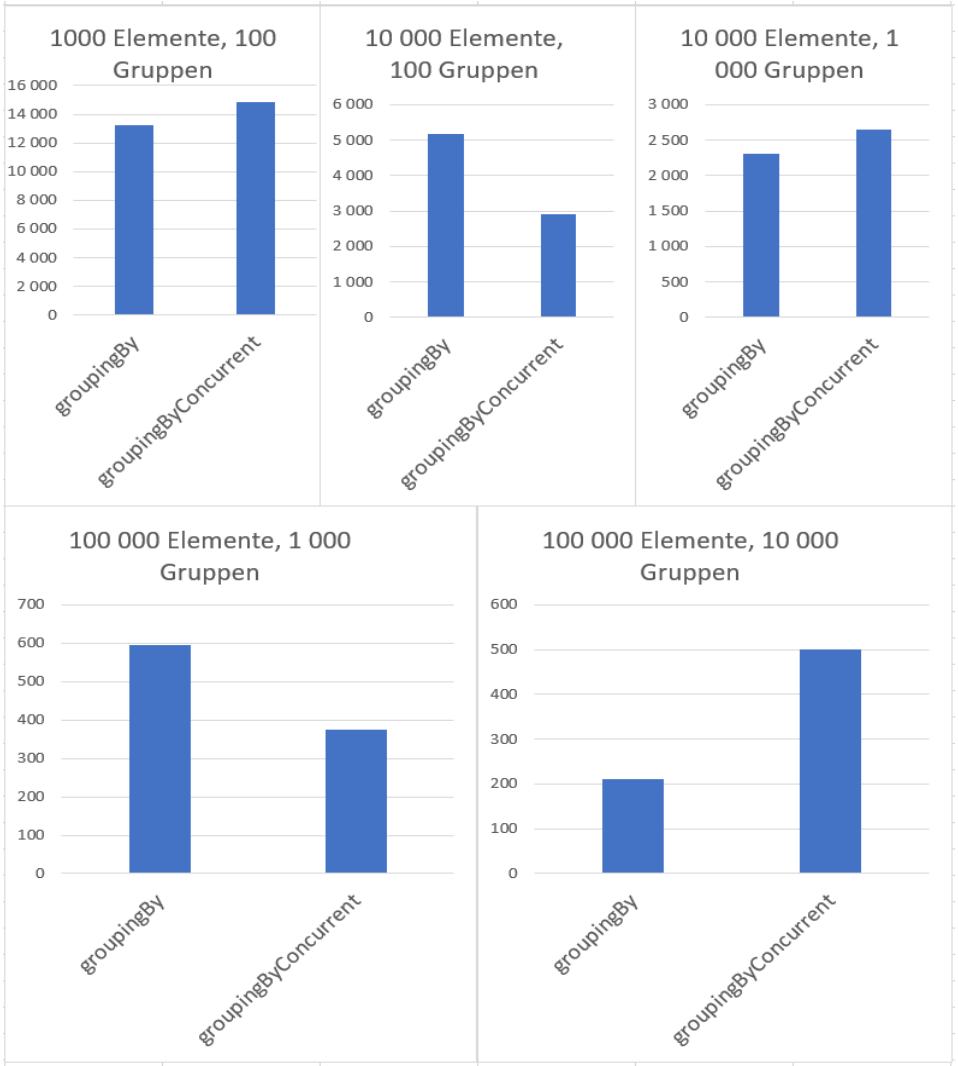


Experiment 12: groupingBy vs. groupingByConcurrent

```
Map<Integer, List<Integer>> m = integerArrList.parallelStream().collect(Collectors.groupingBy(x -> x % N_GROUPS));
Map<Integer, List<Integer>> m = integerArrList.parallelStream().collect(Collectors.groupingByConcurrent(x -> x % N_GROUPS));
```

Operations / seconds
(higher is better)

groupingByConcurrent only for many groups with few members



Summary Run time Performance

- Effort for processing elements has greatest impact on parallel speedup
- Data source should be large
  (but more important is complexity of processing elements)
- Data source has to support splitting well
  - Arrays, ArrayList, range good
  - HashSet, TreeSet bad, iterate very bad
- Data sinks with efficient add und addAll operations
  - ArrayList, LinkedList good
  - HashSet, TreeSet bad
- Specific operations cannot be parallelized and should be avoided
  - e.g., findFirst, limit, distinct are inherently sequential
- Avoid boxing when processing primitive data types
  - use IntStream, DoubleStream, etc. instead of Stream<Integer>, Stream<Double> etc.
- Only use concurrent mapping and grouping when maps with many entries

9 Parallel Streams

- Basics
- Execution by Spliterators
- Conditions
- Parallel collect
- Performance
- Summary

Summary

- Parallel streams work based on the Divide & Conquer principle
- Spliterators combine splitting and sequential processing of streams
- Fork-Join thread pool used for parallel execution
- Achievable speedup dependent on several different factors
  - complexity of processing elements
  - size and type of data source
  - result collections
  - intermediate operations used