Lecture 11

Parallel Data Processing with Streams

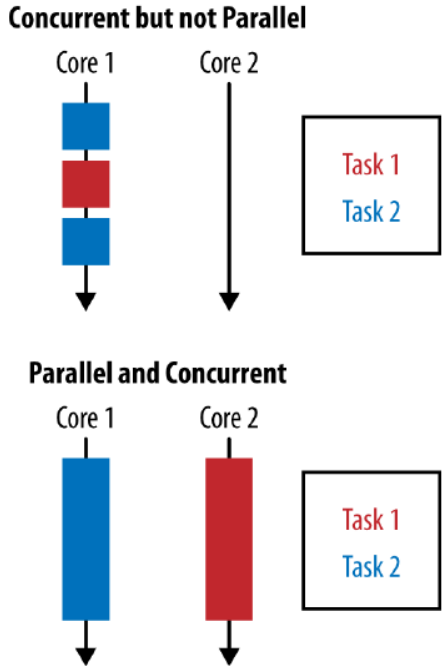
SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Concurrency vs. Parallelism



René Witte
Concordia University

Outline

René Witte

Concordia University

Parallel Streams

René Witte
Concordia University

Java 8

```
public static long sequentialSum(long n) {
  return Stream.iterate(1L, i -> i + 1)
               .limit(n)
               .reduce(0L, Long::sum);
}
```

Java 7

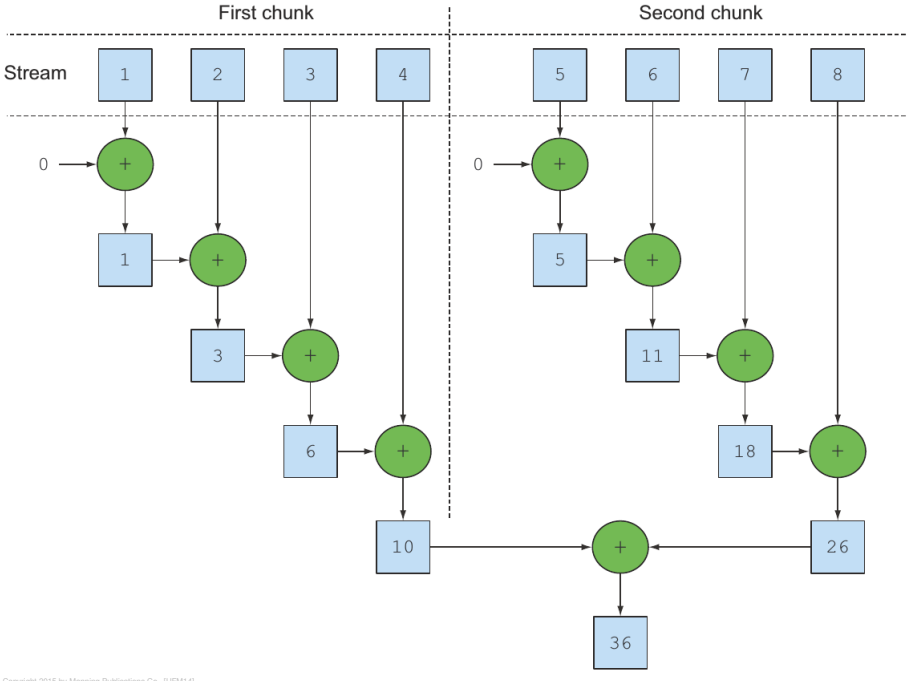
```
public static long iterativeSum(long n) {
  long result = 0;
  for (long i = 1L; i <= n; i++) {
    result += i;
  }
  return result;
}
```

Turning a sequential stream into a parallel one

```
public static long parallelSum(long n) {
  return Stream.iterate(1L, i -> i + 1)
               .limit(n)
               .parallel()
               .reduce(0L, Long::sum);
}
```

René Witte

A parallel reduction operation



René Witte
Concordia University

The methods `sequential` and `parallel`

```
stream.parallel()
      .filter(...)
      .sequential()
      .map(...)
      .parallel()
      .reduce();
```

René Witte

Thread Pools

René Witte

`ForkJoinPool`

```
Runtime.getRuntime().availableProcessors()
```

System property `java.util.concurrent.ForkJoinPool.common.parallelism`

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism",
                   "12");
```

Measuring Stream Performance

```
public long measureSumPerf(Function<Long, Long> adder, long n) {
  long fastest = Long.MAX_VALUE;
  for (int i = 0; i < 10; i++) {
    long start = System.nanoTime();
    long sum = adder.apply(n);
    long duration = (System.nanoTime() - start) / 1_000_000;
    System.out.println("Result: " + sum);
    if (duration < fastest) fastest = duration;
  }
  return fastest;
}
```

René Witte

Some tests

Sequential Sum

```
System.out.println("Sequential sum done in: "
    + measureSumPerf(ParallelStreams::sequentialSum, 10_000_000)
    + " msecs");
```

```
Sequential sum done in: 97 msecs
```

Iterative Sum

```
System.out.println("Iterative sum done in: "
    + measureSumPerf(ParallelStreams::iterativeSum, 10_000_000)
    + " msecs");
```

```
Iterative sum done in: 2 msecs
```

Parallel Sum

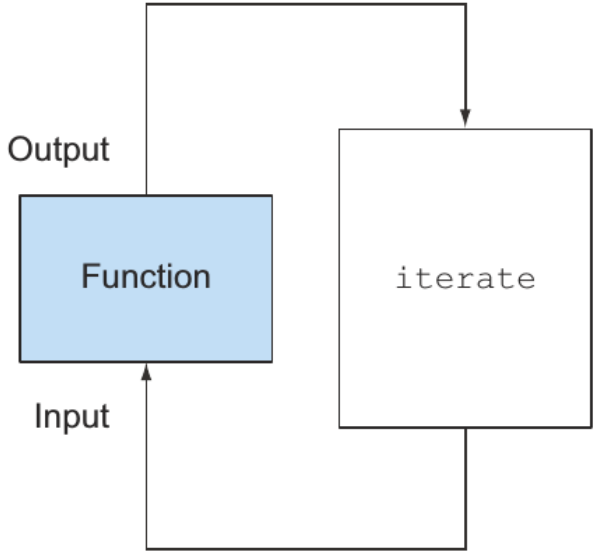
```
System.out.println("Parallel sum done in: "
    + measureSumPerf(ParallelStreams::parallelSum, 10_000_000)
    + " msecs" );
```

```
Parallel sum done in: 164 msecs
```

René Witte

The problem with `iterate`

René Witte



Using more specialized methods

`LongStream.rangeClosed`

- works on primitive `long` – no boxing/unboxing!
- produces ranges of numbers that can be split into chunks

Performance without unboxing

```
public static long rangedSum(long n) {
  return LongStream.rangeClosed(1, n)
                   .reduce(0L, Long::sum);
}

Ranged sum done in: 17 msecs
```

Performance with parallel stream

```
public static long parallelRangedSum(long n) {
  return LongStream.rangeClosed(1, n)
                   .parallel()
                   .reduce(0L, Long::sum);
}

Parallel range sum done in: 1 msecs
```

René Witte

Using parallel streams correctly

```
public static long sideEffectSum(long n) {
  Accumulator accumulator = new Accumulator();
  LongStream.rangeClosed(1, n)
            .forEach(accumulator::add);
  return accumulator.total;
}

public class Accumulator {
  public long total = 0;
  public void add(long value) { total += value; }
}
```

René Witte

Now with `parallel()`

René Witte
Concordia University


```
public static long sideEffectParallelSum(long n) {
  Accumulator accumulator = new Accumulator();
  LongStream.rangeClosed(1, n)
            .parallel()
            .forEach(accumulator::add);
  return accumulator.total;
}

...

System.out.println("SideEffect␣parallel␣sum␣done␣in:␣"
  + measurePerf(ParallelStreams::sideEffectParallelSum, 10_000_000L)
  + "msecs" );
```

Result?

```
Result: 5959989000692
Result: 7425264100768
Result: 6827235020033
Result: 7192970417739
Result: 6714157975331
Result: 7497810541907
Result: 6435348440385
Result: 6999349840672
Result: 7435914379978
Result: 7715125932481
SideEffect parallel sum done in: 49 msecs
```

René Witte

Using parallel streams effectively
Some general guidelines

- Always benchmark the performance
- Avoid boxing – use primitive specializations (`IntStream`, `LongStream`, `DoubleStream`)
- Consider the stream operations – e.g., `limit` and `findFirst` are more expensive on parallel streams
  - use `findAny` unless you must know the first element
  - can turn ordered stream into unordered by calling `unordered`
  - calling `limit` more efficient on unordered parallel stream
- Consider the total computational cost:
  - N items, Q cost of processing/item: total $= N \times Q$
  - parallel streams have better performance for higher values of Q
- Don't use parallel streams for small data sizes

René Witte
Concordia University

René Witte

Concordia University

Decomposition Process

Splitting `ArrayList` more efficient than `LinkedList`

Source	Decomposability
`ArrayList`	Excellent
`LinkedList`	Poor
`IntStream.range`	Excellent
`Stream.iterate`	Poor
`HashSet`	Good
`TreeSet`	Good

René Witte
Concordia University

Decomposition Process (II)

Splitting a `SIZED` stream more efficient than a filtered stream (unknown size)

Stream flag	Interpretation
`SIZED`	The size of the stream is known
`DISTINCT`	The elements of the stream are distinct (using `Object.equals()` for objects; == for primitives)
`SORTED`	The elements of the stream are sorted
`ORDERED`	The stream has a meaningful encounter order

Outline

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Programming Model

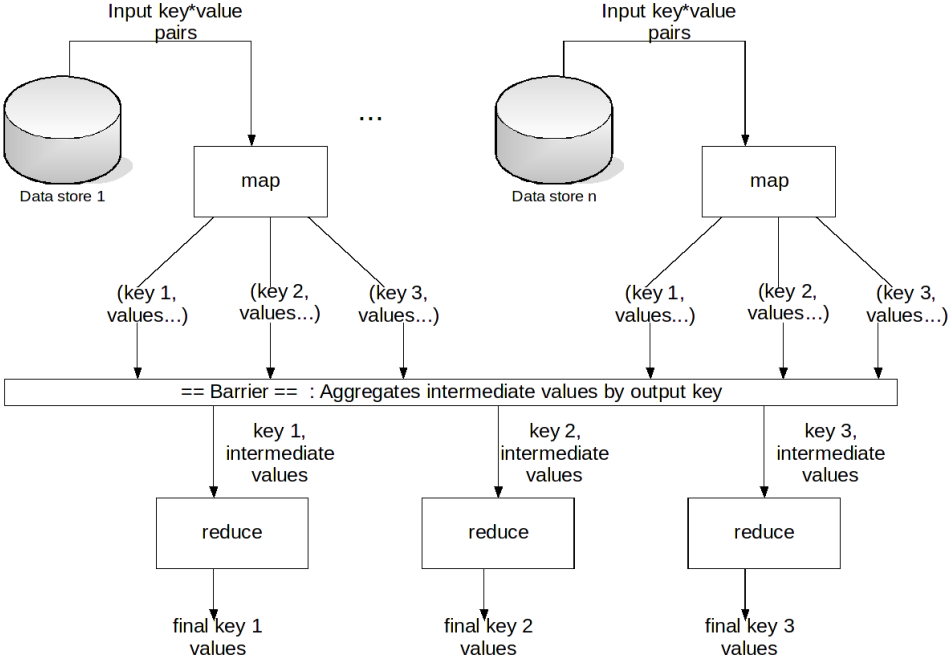
Map

```
map (in_key, in_value) -> list(out_key, intermediate_value)
```

Reduce

```
reduce (out_key, list(intermediate_value)) -> list(out_value)
```

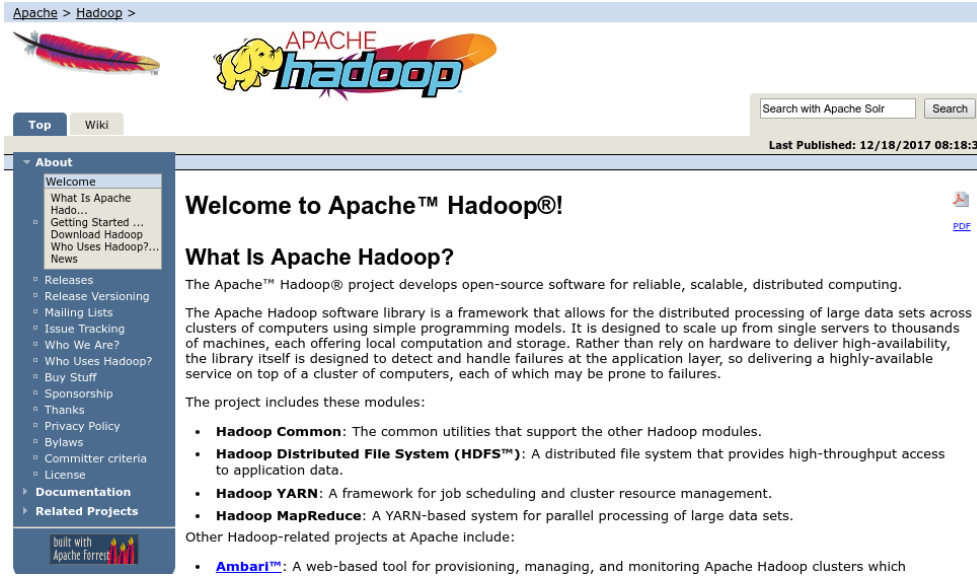
MapReduce Workflow



René Witte
Concordia University

Apache Hadoop

René Witte



Hadoop Programming Example

René Witte

Task: Count words

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02

$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

See http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Map Phase

René Witte
Concordia University

`map` function

```
public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

Output

First file:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

Second file:

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

Reduce Phase

`reduce` function

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context)
              throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

Output

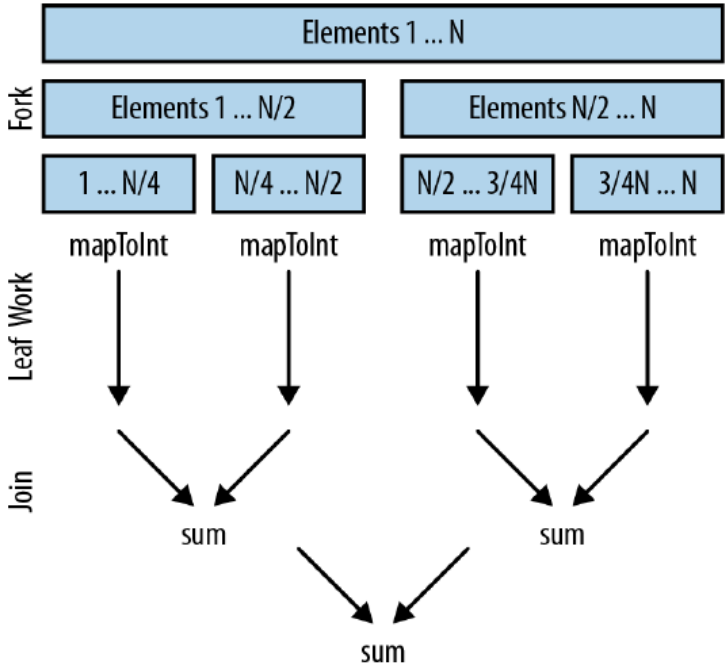
Final result:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

René Witte

Outline

The fork/join framework (Java 7)



René Witte
Concordia University

Working with RecursiveTask

René Witte

To submit tasks to `ForkJoinPool`
create a subclass of either:

`RecursiveTask<R>` where `R` is type of the result

`RecursiveAction` when no result is returned (e.g., side-effects)

Defining a `RecursiveTask`
Implement the abstract method

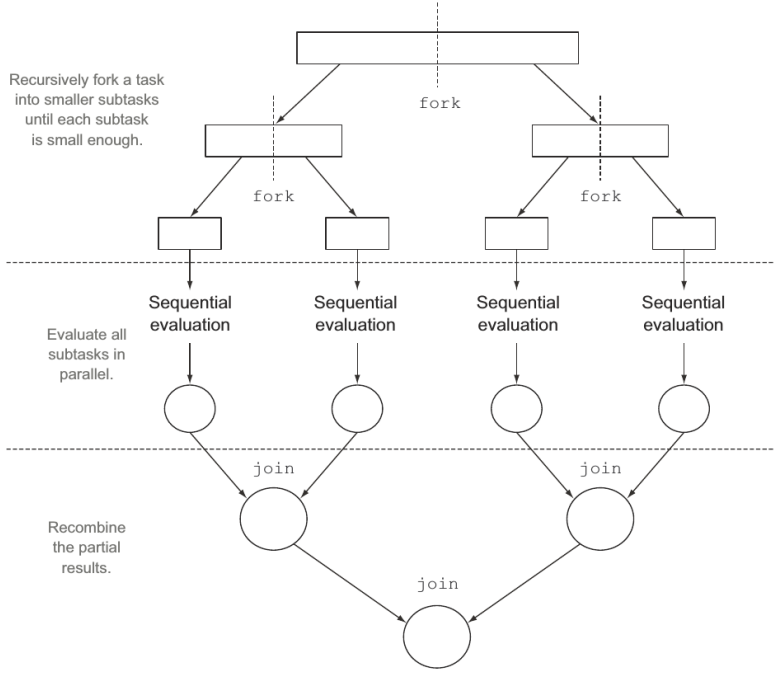
```
protected abstract R compute();
```

Implementing `compute` (Pseudocode)

```
if (task is small enough or no longer divisible) {
  compute task sequentially
} else {
  split task in two subtasks
  call this method recursively, possibly further splitting each subtask
  wait for the completion of all subtasks
  combine the results of each subtask
}
```

René Witte
Concordia University

The fork/join process



René Witte

Parallel sum using fork/join framework

```
public class ForkJoinSumCalculator
       extends java.util.concurrent.RecursiveTask<Long> {

  private final long[] numbers;
  private final int start;
  private final int end;

  public static final long THRESHOLD = 10_000;

  public ForkJoinSumCalculator(long[] numbers) {
    this(numbers, 0, numbers.length);
  }

  private ForkJoinSumCalculator(long[] numbers, int start, int end) {
    this.numbers = numbers;
    this.start = start;
    this.end = end;
  }

  ...
}
```

René Witte

Parallel sum using fork/join framework: `compute`

```
@Override
protected Long compute() {
  int length = end - start;
  if (length <= THRESHOLD) {
    return computeSequentially();
  }
  ForkJoinSumCalculator leftTask =
    new ForkJoinSumCalculator(numbers, start, start + length/2);
  leftTask.fork();
  ForkJoinSumCalculator rightTask =
    new ForkJoinSumCalculator(numbers, start + length/2, end);
  Long rightResult = rightTask.compute();
  Long leftResult = leftTask.join();
  return leftResult + rightResult;
}

private long computeSequentially() {
  long sum = 0;
  for (int i = start; i < end; i++) {
    sum += numbers[i];
  }
}
```

René Witte

Calling the `ForkJoinSumCalculator`

René Witte
Concordia University

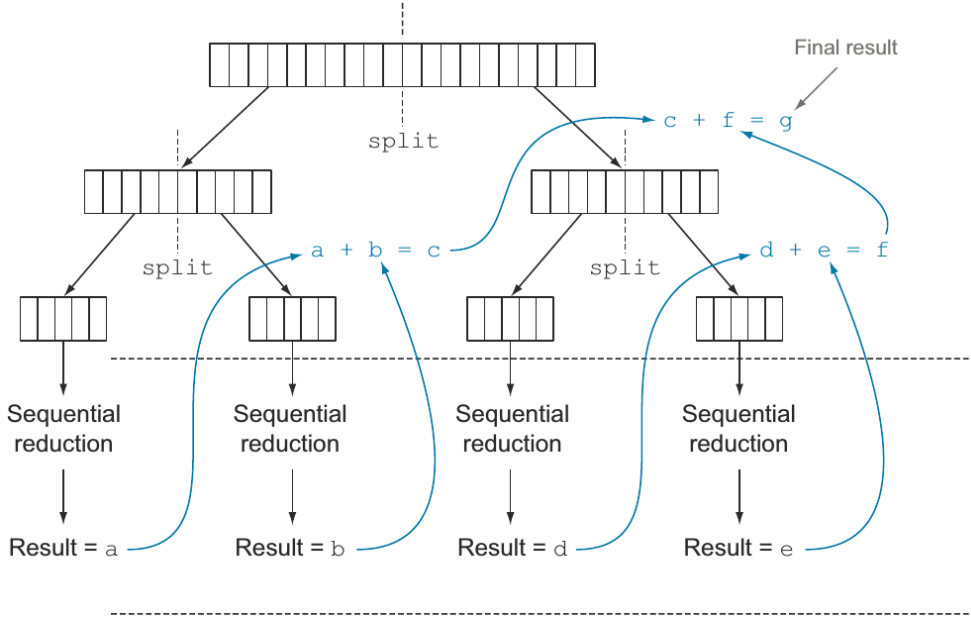

```
public static long forkJoinSum(long n) {
  long[] numbers = LongStream.rangeClosed(1, n).toArray();
  ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);
  return new ForkJoinPool().invoke(task);
}

...

System.out.println("ForkJoin sum done in: "
  + measureSumPerf(ForkJoinSumCalculator::forkJoinSum, 10_000_000)
  + " msecs" );
```

```
ForkJoin sum done in: 41 msecs
```

Running the `ForkJoinSumCalculator`



René Witte
Concordia University

Best practices for using the fork/join framework

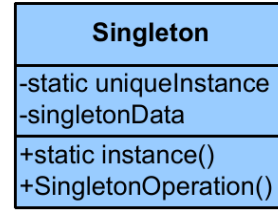
- Invoking `join` blocks the caller: only call it after both subtasks have started
- Do not call `invoke` from within a `RecursiveTask`:
  - call `compute` or `fork` directly
  - only use `invoke` from sequential code to start the fork/join process
- Do not call `fork` twice (creating three threads):
  - re-use current thread for either *left* or *right* subtask
- Measure the performance
  - Be aware of “warm-up” runs and compiler optimizations
- Make `ForkJoinPool` a Singleton in your application

René Witte
Concordia University

Singleton

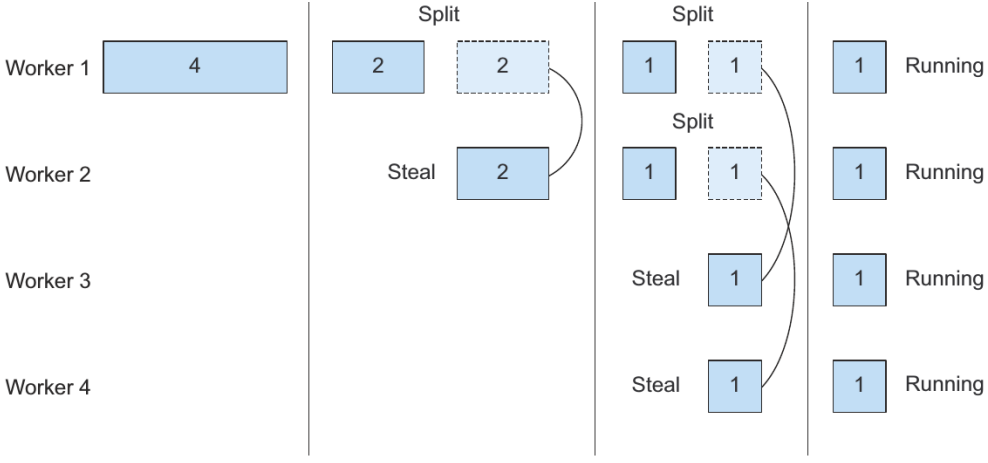
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Work stealing

René Witte



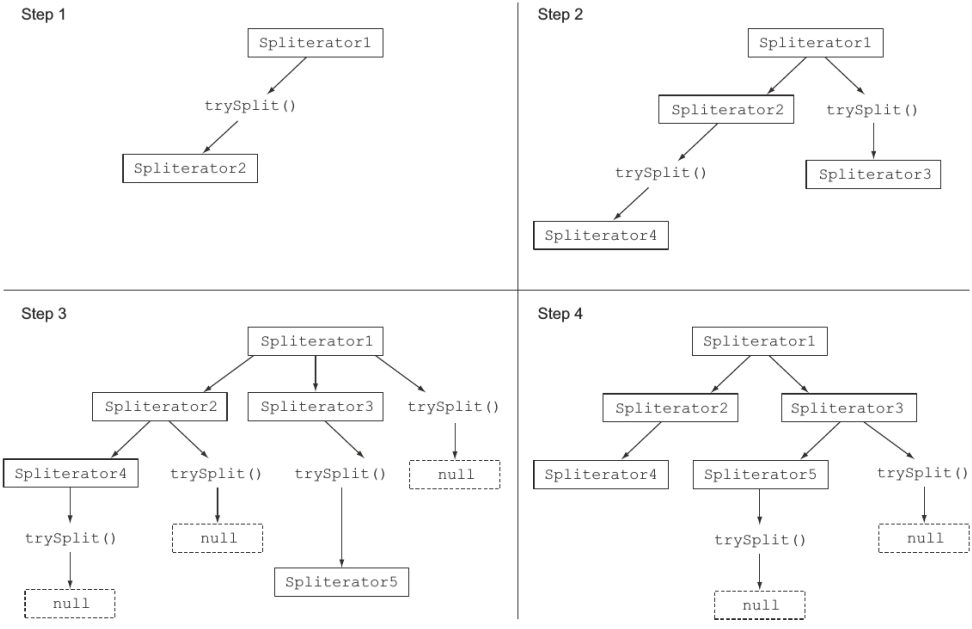
Outline

Spliterator

```
public interface Spliterator<T> {
  boolean tryAdvance(Consumer<? super T> action);
  Spliterator<T> trySplit();
  long estimateSize();
  int characteristics();
}
```

René Witte

The splitting process



René Witte

`Spliterator`'s characteristics

Characteristic	Meaning
`ORDERED`	Elements have a defined order (for example, a `List`), so the `Spliterator` enforces this order when traversing and partitioning them.
`DISTINCT`	For each pair of traversed elements `x` and `y`, `x.equals(y)` returns `false`.
`SORTED`	The traversed elements follow a predefined sort order.
`SIZED`	This `Spliterator` has been created from a source with a known size (for example, a `Set`), so the value returned by `estimatedSize()` is precise.
`NONNULL`	It's guaranteed that the traversed elements won't be `null`.
`IMMUTABLE`	The source of this `Spliterator` can't be modified. This implies that no elements can be added, removed, or modified during their traversal.
`CONCURRENT`	The source of this `Spliterator` may be safely concurrently modified by other threads without any synchronization.
`SUBSIZED`	Both this `Spliterator` and all further `Spliterators` resulting from its split are `SIZED`.

René Witte
Concordia University

Summary

René Witte
Concordia University

Streams

- Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code
- However, always measure your performance and be aware what influences parallel behavior
- Parallel stream execution of an operation on a set of data can provide a significant performance boost on multi-core processors
- From a performance point of view, using the right data structure is very important (e.g., use primitive streams whenever possible)
- Internally, parallel streams make use of Java 7's fork/join framework
- The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the results of each subtask in order to produce the overall result
- Spliterators define how a parallel stream can split the data it traverses

Outline

Reading Material

Required

- [UFM14, Chapter 7] (Parallel Data Processing)

Supplemental

- [War14, Chapter 6] (Data Parallelism)
- [GHJV95] (Singleton Design Pattern)

Further Reading

- [DG04] (MapReduce)
- Apache Hadoop, http://hadoop.apache.org/

References

[DG04] Jeffrey Dean and Sanjay Ghemawat.
MapReduce: Simplified Data Processing on Large Clusters.
In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, USA, December 2004.
https://research.google.com/archive/mapreduce.html.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.

René Witte