

Java 8

Stream Performance

Angelika Langer & Klaus Kreft

<http://www.AngelikaLanger.com/>

agenda

- **introduction**
- loop vs. sequential stream
- sequential vs. parallel stream

what is a stream?

- equivalent of *sequence* from functional programming languages
 - object-oriented view: *internal iterator pattern*
 - see GOF book for more details
- idea

```
myStream. forEach ( s -> System.out.println(s) );
```



stream operation



*user-defined functionality
applied to each element*

obtain a stream

- collection:

```
myCollection.stream(). . . .
```

- array:

```
Arrays.stream(myArray). . . .
```

- resulting stream

- does not store any elements
- just a view of the underlying stream source

- more stream factories, but not in this talk

parallel streams

- collection:

```
myCollection.parallelStream(). . . .
```

- array:

```
Arrays.stream(myArray).parallel(). . . .
```

- performs stream operations in parallel
 - i.e. with multiple worker threads from fork-join common pool

```
myParallelStream.forEach(s -> System.out.print(s));
```

stream functionality rivals loops

- Java 8 streams:

```
myStream.filter(s -> s.length() > 3)
         .mapToInt(s -> s.length)
         .forEach(System.out::print);
```

```
myStream.filter(s -> s.length() > 3)
         .forEach(s->System.out.print(s.length));
```

- since Java 5:

```
for (String s : myCol)
    if (s.length() > 3)
        System.out.print(s.length());
```

- pre-Java 5:

```
Iterator iter = myCol.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    if (s.length() > 3)
        System.out.print(s.length());
}
```

obvious question ...

... how does the performance compare ?

- loop vs. sequential stream vs. parallel stream

agenda

- introduction
- loop vs. sequential stream
- sequential vs. parallel stream

benchmarks ...

... done on an older desktop system with:

- Intel E8500,
 - 2 x 3,17GHz
 - 4GB RAM
- Win 7
- JDK 1.8.0_05
- disclaimer: *your mileage may vary*
 - i.e. parallel performance heavily depends on number of CPU-Cores

agenda

- introduction
- **loop vs. sequential stream**
- sequential vs. parallel stream

how do sequential stream work?

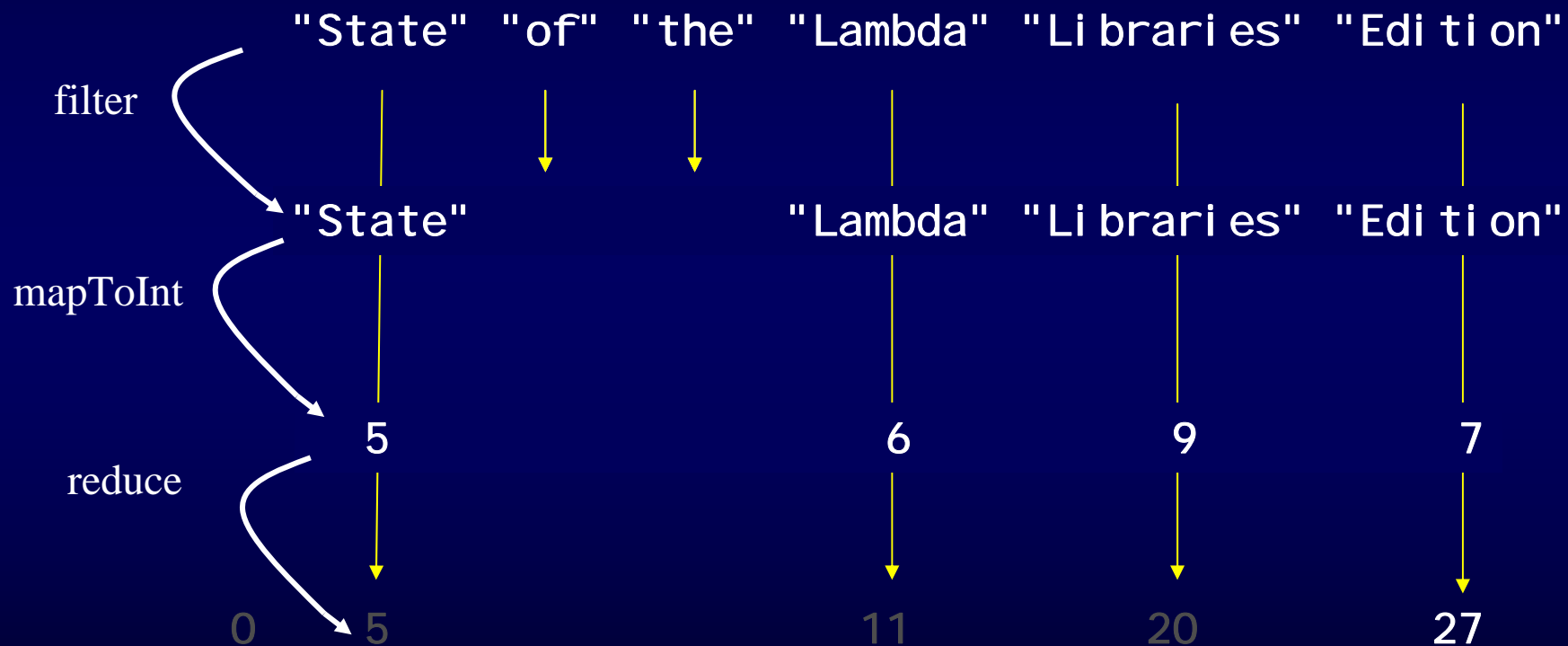
- example

```
String[] txt = { "State", "of", "the", "Lambda",  
                "Li brari es", "Edi ti on"};  
  
int result = Arrays.stream(txt).filter(s -> s.length() > 3)  
                    .mapToInt(s -> s.length())  
                    .reduce(0, (l1, l2) -> l1 + l2);
```

- filter() and mapToInt() return streams
 - intermediate operations
- reduce() returns int
 - terminal operation,
 - that produces a single result from all elements of the stream

pipelined processing

```
Arrays.stream(txt).filter(s -> s.length() > 3)
                  .mapToInt(s -> s.length())
                  .reduce(0, (l1, l2) -> l1 + l2);
```



benchmark with int-array

- `int[500_000]`, find largest element

– for-loop:

```
int[] a = ints;  
int e = ints.length;  
int m = Integer.MIN_VALUE;  
  
for (int i = 0; i < e; i++)  
    if (a[i] > m) m = a[i];
```

– sequential stream:

```
int m = Arrays.stream(ints)  
                .reduce(Integer.MIN_VALUE, Math::max);
```

results

`for-loop:` `0.36 ms`

`seq. stream:` `5.35 ms`

- `for-loop` is ~15x faster
- are `seq. streams` always much slower than loops?
 - no, this is the most extreme example
 - lets see the same benchmark with an `ArrayList<Integer>`
 - underlying data structure is also an array
 - this time filled with `Integer` values, i.e. the boxed equivalent of `int`

benchmark with `ArrayList<Integer>`

- find largest element in an `ArrayList` with 500_000 elements

– for-loop:

```
int m = Integer.MIN_VALUE;
for (int i : myList)
    if (i > m) m = i;
```

– sequential stream:

```
int m = myList.stream()
    .reduce(Integer.MIN_VALUE, Math::max);
```

results

`ArrayList, for-loop:` 6.55 ms

`ArrayList, seq. stream:` 8.33 ms

- for-loop still faster, but only 1,27x
- iteration for `ArrayList` is more expensive
 - boxed elements require an additional memory access (indirection)
 - which does not work well with the CPU's memory cache
- bottom-line:
 - iteration cost dominates the benchmark result
 - performance advantage of the for-loop is insignificant

some thoughts

- previous situation:
 - costs of iteration are relative high, but
 - costs of functionality applied to each element are relative low
 - after JIT-compilation:
more or less the cost of a compare-assembler-instruction
- what if we apply a more expensive functionality to each element ?
 - how will this affect the benchmark results ?

expensive functionality

- `slowSin()`
 - from Apache Commons Mathematics Library
 - calculates a Taylor approximation of the sine function value for the parameter passed to this method
 - (normally) not in the public interface of the library
 - used to calculate values for an internal table,
 - which is used for interpolation by `FastCalcMath.sin()`

benchmark with `slowSin()`

- `int` array / `ArrayList` with 10_000 elements

– for-loop:

```
int[] a = ints;
int e = a.length;
double m = Double.MIN_VALUE;

for (int i = 0; i < e; i++) {
    double d = Sine.slowSin(a[i]);
    if (d > m) m = d;
}
```

– sequential stream:

```
Arrays.stream(ints)
    .mapToDouble(Sine::slowSin)
    .reduce(Double.MIN_VALUE, Math::max);
```

– code for `ArrayList` changed respectively

results

<code>int[], for-loop:</code>	<code>11.72 ms</code>
<code>int[], seq. stream:</code>	<code>11.85 ms</code>
<code>ArrayList, for-loop:</code>	<code>11.84 ms</code>
<code>ArrayList, seq. stream:</code>	<code>11.85 ms</code>

- for-loop is not really faster
- reason:
 - applied functionality costs dominate the benchmark result
 - performance advantage of the for-loop has evaporated

other aspect (without benchmark)

- today, compilers (javac + JIT) can optimize loops better than stream code
- reasons:
 - linear code (loop) vs. injected functionality (stream)
 - lambdas + method references are new to Java
 - loop optimization is a very mature technology
 - ...

for-loop vs. seq. stream / re-cap

- sequential stream can be slower or as fast as for-loop
- depends on
 - costs of the iteration
 - costs of the functionality applied to each element
- the higher the cost (iteration + functionality)
the closer is stream performance
to for-loop performance

agenda

- introduction
- loop vs. sequential stream
- **sequential vs. parallel stream**
 - **introduction**
 - stateless functionality
 - stateful functionality

parallel streams

- library side parallelism
 - important feature
 - do not know anything about threads, etc.
 - very little implementation effort, just: parallel
- performance aspect
 - outperform loops, which are inherently sequential

how do parallel stream work?

- example

```
final int SIZE = 64;
int[] ints = new int[SIZE];
ThreadLocalRandom rand = ThreadLocalRandom.current();
for (int i=0; i<SIZE; i++) ints[i] = rand.nextInt();

Arrays.stream(ints)
    .parallel()
    .reduce(Math::max)
    .ifPresent(System.out::println(m -> "max is: " + m));
```

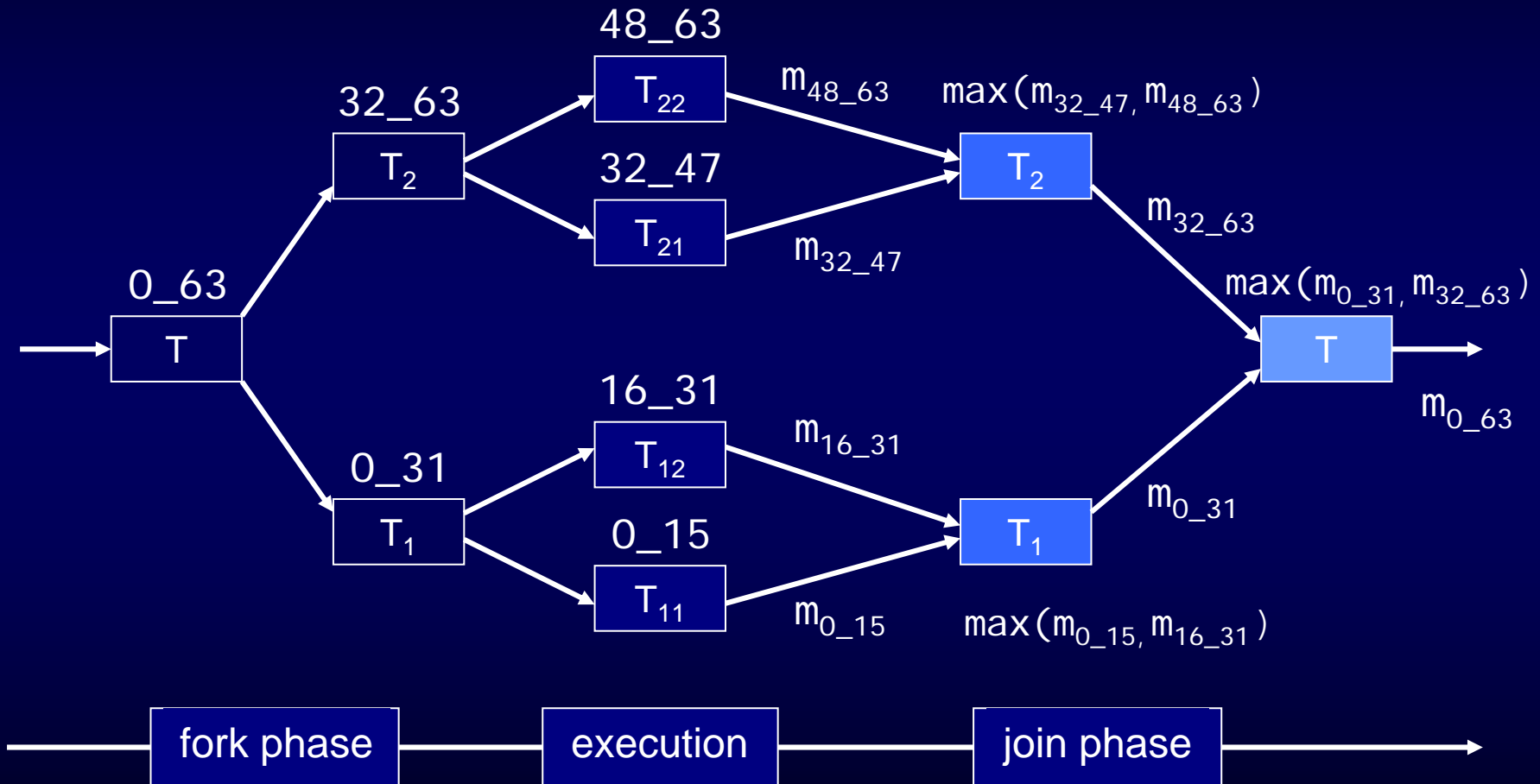
- `parallel()`'s functionality is based on
the fork-join framework

fork join tasks

- original task is divided into two sub-tasks
 - by splitting the stream source into two parts
 - original task's result are based on sub-tasks' results
 - sub-tasks are divided again ... *fork phase*
- at a certain depth partitioning stops
 - tasks at this level (leaf tasks) are executed
 - *execution phase*
- completed sub-task results
 - are 'combined' to super-task results
 - *join phase*

find largest element with parallel stream

```
reduce((i , j) -> Math.max(i , j));
```



split level

- deeper split level than shown !!!
 - execution/leaf tasks: $\sim 4 * \text{numberOfCores}$
 - 8 tasks for a dual core CPU (only 4 in the previous diagram)
 - i.e. one additional split (only 2 in the previous graphic)
- key abstractions
 - `java.util.Spliterator`
 - `java.util.concurrent.ForkJoinPool.commonPool()`

what is a Spliterator ?

- spliterator = splitter + iterator
- each type of stream source has its own spliterator type
 - knows how to split the stream source
 - e.g. `ArrayListSpliterator`
 - knows how to iterate the stream source
 - during execution phase
 - also used by sequential streams
 - to iterate the whole stream source

what is the CommonPool ?

- *common pool* is a singleton fork-join pool instance
 - introduced with Java 8
 - all parallel stream operations use the common pool
 - so does other parallel JDK functionality (e.g. `CompletableFuture`), too
- default: parallel execution of stream tasks uses
 - (current) thread that invoked terminal operation, and
 - (number of cores – 1) many threads from common pool
 - if (number of cores) > 1
- this default configuration used for all benchmarks

parallel streams + intermediate operations

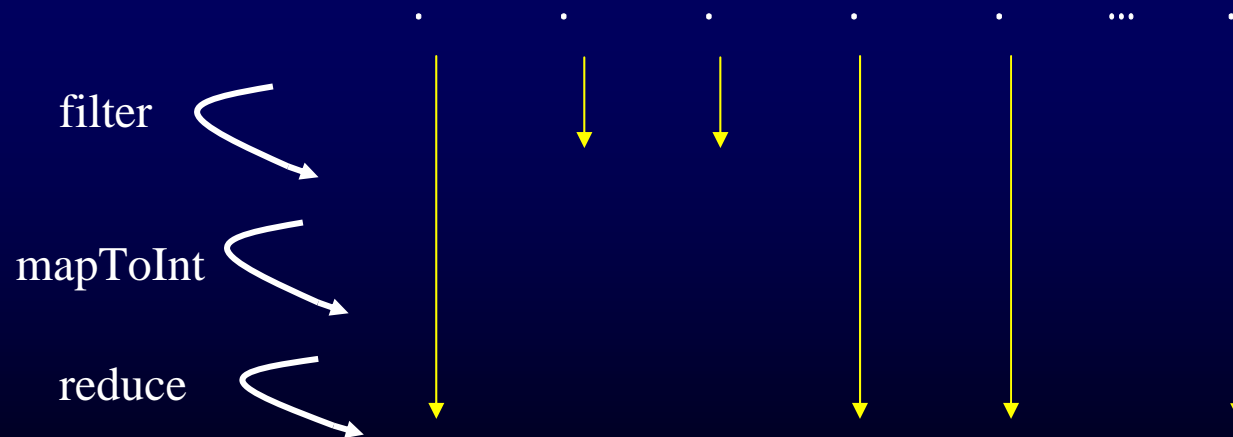
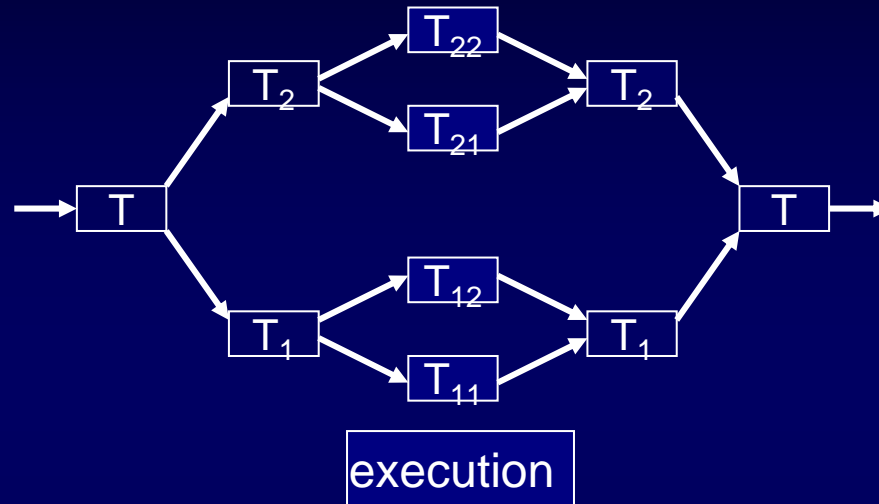
- what if the stream contains
upstream intermediate operations

```
... .parallelStream().filter(...)  
    .mapToInt(...)  
    .reduce((i, j) -> Math.max(i, j));
```

when/where are these applied to the stream ?

find largest element in parallel

```
filter(...).mapToInt(...).reduce((i, j) -> Math.max(i, j));
```



parallel overhead ...

... compared to sequential stream algorithm

- algorithm is more complicated / resource intensive
 - create fork-join-task objects
 - splitting
 - fork-join-task objects creation
 - thread pool scheduling
 - ...
- plus additional GC costs
 - fork-join-task objects have to be reclaimed

agenda

- introduction
- loop vs. sequential stream
- **sequential vs. parallel stream**
 - introduction
 - **stateless functionality**
 - stateful functionality

back to the first example / benchmark parallel

- find largest element, array / collection, 500_000 elements
 - sequential stream:

```
int m = Arrays.stream(ints)
               .reduce(Integer.MIN_VALUE, Math::max);
```

```
int m = myCollection.stream()
                  .reduce(Integer.MIN_VALUE, Math::max);
```

- parallel stream:

```
int m = Arrays.stream(ints).parallel()
               .reduce(Integer.MIN_VALUE, Math::max);
```

```
int m = myCollection.parallelStream()
                  .reduce(Integer.MIN_VALUE, Math::max);
```

results

	seq.	par.	seq./par.
<code>int-Array</code>	5.35 ms	3.35 ms	1.60
<code>ArrayList</code>	8.33 ms	6.33 ms	1.32
<code>LinkedList</code>	12.74 ms	19.57 ms	0.65
<code>HashSet</code>	20.76 ms	16.01 ms	1.30
<code>TreeSet</code>	19.79 ms	15.49 ms	1.28

result discussion

- why is parallel `LinkedList` performance so bad ?
 - hard to split
 - needs 250_000 iterator's `next()` invocations for the first split
 - with `ArrayList`: just some index computation
- performance of the other collections is also not so great
 - functionality applied to each element is not very CPU-expensive
 - after JIT-compilation: cost of a compare-assembler-instruction
 - iteration (element access) is relative expensive (indirection !)
 - but not CPU expensive
 - but more CPU-power is what we have with parallel streams

result discussion (cont.)

- why is parallel int-array performance relatively good ?
 - iteration (element access) is not so expensive (no indirection !)

CPU-expensive functionality

- back to `slowSin()`
 - calculates a Taylor approximation of the sine function value for the parameter passed to this method
 - CPU-bound functionality
 - needs only the initial parameter from memory
 - calculation based on it's own (intermediate) results
 - ideal to be speed up by parallel streams with multiple cores

benchmark parallel with `slowSin()`

- array / collection with 10_000 elements

– array:

```
Arrays.stream(ints) // .parallel()
    .mapToDouble(Sine::slowSin)
    .reduce(Double.MIN_VALUE, (i, j) -> Math.max(i, j));
```

– collection:

```
myCollection.stream() // .parallelStream()
    .mapToDouble(Sine::slowSin)
    .reduce(Double.MIN_VALUE, (i, j) -> Math.max(i, j));
```

results

	seq.	par.	seq./par.
<code>int-Array</code>	10.81 ms	6.03 ms	1.79
<code>ArrayList</code>	10.97 ms	6.10 ms	1.80
<code>LinkedList</code>	11.15 ms	6.25 ms	1.78
<code>HashSet</code>	11.15 ms	6.15 ms	1.81
<code>TreeSet</code>	11.14 ms	6.30 ms	1.77

result discussion

- performance improvements for all stream sources
 - by a factor of ~ 1.8
 - even for `LinkedList`
- the ~ 1.8 is the maximum improvement on our platform
 - the remaining 0.2 are
 - overhead of the parallel algorithm
 - sequential bottlenecks (Amdahl's law)

sufficient size (without benchmark)

- stream source must have a sufficient size,
so that it benefits from parallel processing
- Doug Lea mentioned 10_000 for CPU-inexpensive funct.
 - <http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html>
- 500_000 respectively 10_000 in our examples
 - size can be smaller for CPU-expensive functionality

dynamic overclocking (without benchmark)

- modern multi-core CPU typically increases the CPU-frequency when not all of its cores are active
 - Intel call this feature: turbo boost
- benchmark sequential versus parallel stream
 - seq. test might run with a dynamically overclocked CPU
 - will this also happen in the real environment or only in the test?
- no issue with our test system
 - CPU too old
 - no dynamic overclocking

agenda

- introduction
- loop vs. sequential stream
- **sequential vs. parallel stream**
 - introduction
 - stateless functionality
 - **stateful functionality**

stateful functionality ...

... with parallel streams / multiple threads boils down to
shared mutable state

- costs performance to handle this
 - lock, requires waiting
 - lock-free CAS, requires retries in case of collision
- traditionally not supported with *sequences*
 - functional programming languages don't have mutable types, and
 - (often) no parallel sequences either
- new solutions/approaches in Java 8 streams

stateful functionality with Java 8 streams

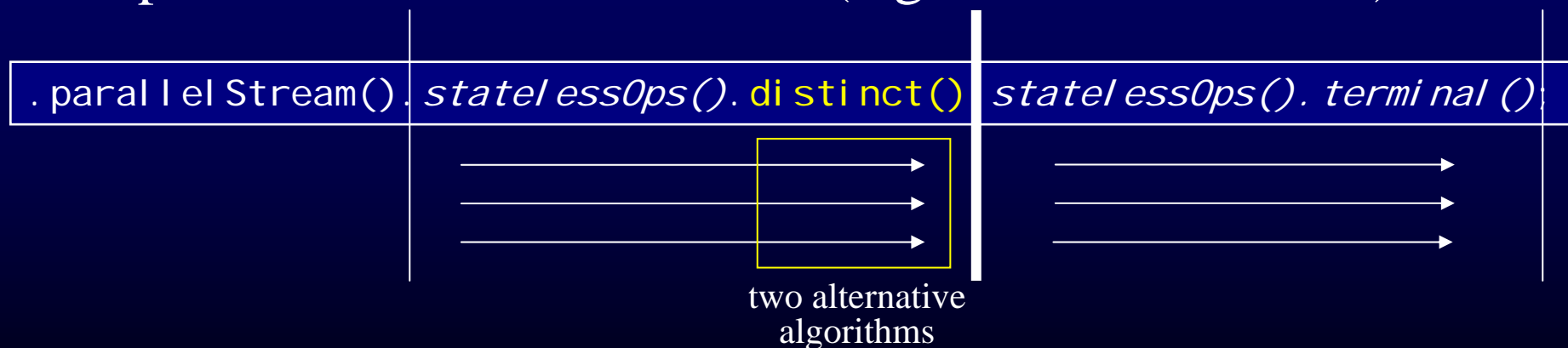
- intermediate stateful operations, e.g. `distinct()`
 - see javadoc: *This is a stateful intermediate operation.*
 - shared mutable state handled by stream implementation (JDK)
- (terminal) operations that allow stateful functional parameters, e.g.
 - `forEach(Consumer<? super T> action)`
 - see javadoc: *If the action accesses shared state, it is responsible for providing the required synchronization.*
 - shared mutable state handled by user/client code

stateful functionality with Java 8 streams (cont.)

- stream's overloaded terminal operation: `collect()`
 - shared mutable state handled by stream implementation, and
 - collector functionality
 - standard collectors from `Collectors` (JDK)
 - user-defined collector functionality (JDK + user/client code)
- don't have time to discuss all situations
 - only discuss `distinct()`
 - shared mutable state handled by stream implementation (JDK)

distinct()

- element goes to the result stream,
if it hasn't already appeared before
 - appeared before, in terms of equals()
 - shared mutable state: elements already in the result stream
 - have to compare the current element to each element of the output stream
- parallel introduces a barrier (algorithmic overhead)



two algorithms for parallel `distinct()`

- `ordering + distinct()`
 - normally elements go to the next stage, in the same order in which they appear for the first time in the current stage
- javadoc from `distinct()`
 - *Removing the **ordering** constraint with `unordered()` may result in significantly **more efficient execution** for `distinct()` in parallel pipelines, if the semantics of your situation permit.*
- two different algorithms for parallel `distinct()`
 - one for ordered result stream + one for unordered result stream

benchmark with distinct()

- Integer[100_000], filled with 50_000 distinct values

```
// sequential  
Arrays.stream(integers).distinct().count();
```

```
// parallel ordered  
Arrays.stream(integers).parallel().distinct().count();
```

```
// parallel unordered  
Arrays.stream(integers).parallel().unordered().distinct().count();
```

- results:

seq.	par. ordered	par. unordered
6.39 ms	34.09 ms	9.1 ms

benchmark with `distinct()` + `slowSin()`

- `Integer[10_000]`, filled with numbers 0 ... 9999

```
Arrays.stream(newIntegers) // .parallel().unordered()
    .map(i -> new Double(2200 * Sine.slowSin(i * 0.001)).intValue())
    .distinct()
    .count();
```

– after the mapping 5004 distinct values

- results:

<code>seq.</code>	<code>par. ordered</code>	<code>par. unordered</code>
11.59 ms	6.83 ms	6.81 ms

sequential vs. parallel stream / re-cap

to benefit from parallel stream usage ...

- ... stream source ...
 - must have sufficient size
 - should be easy to split
- ... operations ...
 - should be CPU-expensive
 - should not be stateful

advice

- benchmark on target platform !
- previous benchmark:
 - find largest element, `LinkedList`, 500_000 elements

seq.	par.	seq./par.
12.74 ms	19.57 ms	0.65

- what if we use a quad-core-CPU (Intel i5-4590) ?
 - will the parallel result be worse, better, ... better than seq. ... ?

seq.	par.	seq./par.
5.24 ms	4.84 ms	1.08

authors

Angelika Langer

Klaus Kreft

<http://www.AngelikaLanger.com>

stream performance

Q & A