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

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

- *stream operation*
- *user-defined functionality applied to each element*
fluent programming

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

- **Intermediate operations**
- **Terminal operation**

- stream operation
- user-defined functionality applied to each element
obtain a stream

• collection:
  ```java
  myCollection.stream(). ... 
  ```

• array:
  ```java
  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:
  
  \[\text{myCollection.parallelStream() \ldots}\]

- array:
  
  \[\text{Arrays.stream(myArray).parallel() \ldots}\]

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

  \[\text{myParallelStream.forEach(s -> System.out.print(s))};\]
stream functionality rivals loops

- Java 8 streams:

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

- since Java 5:

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

- pre-Java 5:

```java
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

```java
String[] txt = {"State", "of", "the", "Lambda",
                "Libraries", "Edition"};

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:
  ```java
  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:
  ```java
  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
  - let's 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 \texttt{ArrayList\<Integer\>}

- find largest element in an \texttt{ArrayList} with 500_000 elements
  - \texttt{for}-loop:
    
    ```java
    int m = \texttt{Integer.MIN\_VALUE};
    for (int i : myList)
        if (i > m) m = i;
    ```
  
  - sequential stream:
    
    ```java
    int m = myList.stream()
        .reduce(\texttt{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:
    ```java
    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:
    ```java
    Arrays.stream(ints)
    .mapToDouble(Sine::slowSin)
    .reduce(Double.MIN_VALUE, Math::max);
    ```
  - code for ArrayList changed respectively
results

```plaintext
int[], for-loop:          11.72 ms
int[], seq. stream:      11.85 ms
ArrayList, for-loop:     11.84 ms
ArrayList, seq. stream:  11.85 ms
```

- 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

```java
final int SIZE = 64;
int[] ints = new int[SIZE];
ThreadLocal Random rand = ThreadLocal Random 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

\[
\text{reduce}((i, j) -> \text{Math.max}(i, j));
\]
split level

• deeper split level than shown !!!
  – execution/leaf tasks: \( \sim 4 \times \text{numberOfCores} \)
    › 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. `ArrayList.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

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

when/where are these applied to the stream?
find largest element in parallel

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

**Diagram:**
- **Filter**
- **MapToInt**
- **Reduce**

**Execution:**

```
.  .  .  .  .  .  .  .  .
```

\( T \)

\( T_1 \)

\( T_2 \)

\( T_1 \)

\( T_2 \)

\( T_1 \)

\( T_2 \)

\( T_1 \)
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:
    ```java
    int m = Arrays.stream(ints)
               .reduce(Integer.MIN_VALUE, Math::max);
    
    int m = myCollection.stream()
               .reduce(Integer.MIN_VALUE, Math::max);
    ```
  - parallel stream:
    ```java
    int m = Arrays.stream(ints).parallel()
               .reduce(Integer.MIN_VALUE, Math::max);
    
    int m = myCollection.parallelStream()
              .reduce(Integer.MIN_VALUE, Math::max);
    ```
<table>
<thead>
<tr>
<th></th>
<th>seq.</th>
<th>par.</th>
<th>seq./par.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int-Array</td>
<td>5.35 ms</td>
<td>3.35 ms</td>
<td>1.60</td>
</tr>
<tr>
<td>ArrayList</td>
<td>8.33 ms</td>
<td>6.33 ms</td>
<td>1.32</td>
</tr>
<tr>
<td>LinkedList</td>
<td>12.74 ms</td>
<td>19.57 ms</td>
<td>0.65</td>
</tr>
<tr>
<td>HashSet</td>
<td>20.76 ms</td>
<td>16.01 ms</td>
<td>1.30</td>
</tr>
<tr>
<td>TreeSet</td>
<td>19.79 ms</td>
<td>15.49 ms</td>
<td>1.28</td>
</tr>
</tbody>
</table>
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 no so expensive (no indirection!)
CPU-expensive functionality

• back to \texttt{slowSin()}:
  \begin{itemize}
    \item calculates a Taylor approximation of the sine function value for the parameter passed to this method
    \item CPU-bound functionality
      \begin{itemize}
        \item needs only the initial parameter from memory
        \item calculation based on it’s own (intermediate) results
      \end{itemize}
    \item ideal to be speed up by parallel streams with multiple cores
  \end{itemize}
benchmark parallel with \texttt{slowSin()} 

- array / collection with 10,000 elements

  - array:

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

  - collection:

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

<table>
<thead>
<tr>
<th></th>
<th>seq.</th>
<th>par.</th>
<th>seq./par.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int-Array</td>
<td>10.81 ms</td>
<td>6.03 ms</td>
<td>1.79</td>
</tr>
<tr>
<td>ArrayList</td>
<td>10.97 ms</td>
<td>6.10 ms</td>
<td>1.80</td>
</tr>
<tr>
<td>LinkedList</td>
<td>11.15 ms</td>
<td>6.25 ms</td>
<td>1.78</td>
</tr>
<tr>
<td>HashSet</td>
<td>11.15 ms</td>
<td>6.15 ms</td>
<td>1.81</td>
</tr>
<tr>
<td>TreeSet</td>
<td>11.14 ms</td>
<td>6.30 ms</td>
<td>1.77</td>
</tr>
</tbody>
</table>
result discussion

- performance improvements for all stream sources
  - by a factor of $\sim 1.8$
    - even for LinkedList

- the $\sim 1.8$ 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.

- 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.

```java
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)

```
.parallelStream().statelessOps().distinct().statelessOps().terminal();
```
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

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

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

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

- results:

<table>
<thead>
<tr>
<th></th>
<th>seq.</th>
<th>par. ordered</th>
<th>par. unordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.39 ms</td>
<td>34.09 ms</td>
<td>9.1 ms</td>
</tr>
</tbody>
</table>
benchmark with `distinct() + slowSin()`

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

```java
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:

<table>
<thead>
<tr>
<th>seq.</th>
<th>par. ordered</th>
<th>par. unordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.59 ms</td>
<td>6.83 ms</td>
<td>6.81 ms</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>seq.</th>
<th>par.</th>
<th>seq./par.</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.74 ms</td>
<td>19.57 ms</td>
<td>0.65</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>seq.</th>
<th>par.</th>
<th>seq./par.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.24 ms</td>
<td>4.84 ms</td>
<td>1.08</td>
</tr>
</tbody>
</table>
authors

Angelika Langer

Klaus Kreft

http://www.AngelikaLanger.com
stream performance

Q & A