

23.13 Java SE 8: Sequential vs. Parallel Streams

8

In [Chapter 17](#), you learned about Java SE 8 lambdas and streams. We mentioned that streams are easy to *parallelize*, enabling programs to benefit from enhanced performance on multi-core systems. Using the timing capabilities introduced in [Section 23.12](#), [Fig. 23.30](#) demonstrates both *sequential* and *parallel* stream operations on a 50,000,000-element array of random long values (created at line 15) to compare the performance.

```
1 // Fig. 23.30: StreamStatisticsComparison.java
2 // Comparing performance of sequential and paral
3 import java.time.Duration;
4 import java.time.Instant;
5 import java.util.Arrays;
6 import java.util.LongSummaryStatistics;
7 import java.util.stream.LongStream;
8 import java.util.Random;
9
10 public class StreamStatisticsComparison {
11     public static void main(String[] args) {
12         Random random = new Random();
13
14         // create array of random long values
15         long[] values = random.longs(50_000_000, 1
16
```

```
17      // perform calculations separately
18      Instant separateStart = Instant.now();
19      long count = Arrays.stream(values).count()
20      long sum = Arrays.stream(values).sum();
21      long min = Arrays.stream(values).min().get
22      long max = Arrays.stream(values).max().get
23      double average = Arrays.stream(values).ave
24          Instant separateEnd = Instant.now();
25
26      // display results
27      System.out.println("Calculations performed")
28      System.out.printf(" count: %,d%n", count);
29      System.out.printf("      sum: %,d%n", sum)
30      System.out.printf("      min: %,d%n", min)
31      System.out.printf("      max: %,d%n", max)
32      System.out.printf(" average: %f%n", avera
33      System.out.printf("Total time in milliseco
34          Duration.between(separateStart, separat
35
36      // time summaryStatistics operation with s
37      LongStream stream1 = Arrays.stream(values)
38      System.out.println("Calculating statistics")
39      Instant sequentialStart = Instant.now();
40      LongSummaryStatistics results1 = stream1.s
41      Instant sequentialEnd = Instant.now();
42
43      // display results
44          displayStatistics(results1);
45      System.out.printf("Total time in milliseco
46          Duration.between(sequentialStart, sequen
47
48      // time sum operation with parallel stream
49      LongStream stream2 = Arrays.stream(values)
50      System.out.println("Calculating statistics")
51      Instant parallelStart = Instant.now();
52      LongSummaryStatistics results2 = stream2.s
53          Instant parallelEnd = Instant.now();
54
55      // display results
56          displayStatistics(results1);
```

```
57     System.out.printf("Total time in milliseco
58         Duration.between(parallelStart, paralle
59             }
60
61     // display's LongSummaryStatistics values
62     private static void displayStatistics(LongSum
63         System.out.println("Statistics");
64         System.out.printf("    count: %,d%n", stat
65         System.out.printf("    sum: %,d%n", stat
66         System.out.printf("    min: %,d%n", stat
67         System.out.printf("    max: %,d%n", stat
68         System.out.printf("    average: %f%n", stats
69             }
70     }
```



```
Calculations performed separately
    count: 50,000,000
    sum: 25,025,212,218
        min: 1
        max: 1,000
        average: 500.504244
    Total time in milliseconds: 710
Calculating statistics on sequential stream Statistic
    count: 50,000,000
    sum: 25,025,212,218
        min: 1
        max: 1,000
        average: 500.504244
    Total time in milliseconds: 305

Calculating statistics on parallel stream
    Statistics
    count: 50,000,000
    sum: 25,025,212,218
        min: 1
        max: 1,000
        average: 500.504244
    Total time in milliseconds: 143
```



Fig. 23.30

Comparing performance of sequential and parallel stream operations.

Performing Stream Operations with Separate Passes of a Sequential Stream

Section 17.7 demonstrated various numerical operations on `IntStreams`. Lines 18–24 of Fig. 23.30 perform and time the `count`, `sum`, `min`, `max` and `average` stream operations each performed individually on a `LongStream` returned by `Arrays` method `stream`. Lines 27–34 then display the results and the total time required to perform all five operations.

Performing Stream Operations with a Single Pass of a Sequential

Stream

Lines 37–46 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` to determine the count, sum, minimum value, maximum value and average in one pass of a *sequential* `LongStream`—all streams are sequential by default. This operation took approximately 43% of the time required to perform the five operations separately.

Performing Stream Operations with a Single Pass of a Parallel Stream

Lines 49–50 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` on a *parallel* `LongStream`. To obtain a parallel stream that can take advantage of multi-core processors, simply invoke method `parallel` on an existing stream. As you can see from the sample output, performing the operations on a parallel stream decreased the total time required even further—taking approximately 47% of the calculation time for the sequential `LongStream` and just 20% of the time required to perform the five operations separately.