23.12 `sort/parallelSort` Timings with the Java SE 8 Date/Time API

8

In Section 7.15, we used class `Arrays`'s `static` method `sort` to sort an array and we introduced `static` method `parallelSort` for sorting large arrays more efficiently on multi-core systems. Figure 23.29 uses both methods to sort 100,000,000 element arrays of random `int` values so that we can demonstrate `parallelSort`'s performance improvement of over `sort` on a multi-core system (we ran this example on a quad-core system).4

4. To create the 100,000,000 element array in this example, we used `Random` rather than `SecureRandom`, because `Random` executes significantly faster.

```
   1   // Fig. 23.29: SortComparison.java
   2   // Comparing performance of Arrays methods sort
   3   import java.time.Duration;
   4   import java.time.Instant;
   5   import java.text.NumberFormat;
   6   import java.util.Arrays;
   7   import java.util.Random;
   8
   9   public class SortComparison {
```

```
10      public static void main(String[] args) {
11          Random random = new Random();
12
13          // create array of random ints, then copy
14          int[] array1 = random.ints(100_000_000).to
15          int[] array2 = array1.clone();
16
17          // time the sorting of array1 with Arrays
18          System.out.println("Starting sort");
19          Instant sortStart = Instant.now();
20          Arrays.sort(array1);
21          Instant sortEnd = Instant.now();
22
23          // display timing results
24          long sortTime = Duration.between(sortStart
25          System.out.printf("Total time in milliseco
26
27          // time the sorting of array2 with Arrays
28          System.out.println("Starting parallelSort"
29          Instant parallelSortStart = Instant.now();
30          Arrays.parallelSort(array2);
31          Instant parallelSortEnd = Instant.now();
32
33          // display timing results
34          long parallelSortTime =
35             Duration.between(parallelSortStart, par
36          System.out.printf("Total time in milliseco
37             parallelSortTime);
38
39          // display time difference as a percentage
40          String percentage = NumberFormat.getPercen
41             (double) (sortTime - parallelSortTime)
42          System.out.printf("sort took %s more time
43             percentage);
44       }
45    }
```

```
Starting sort
```

```
            Total time in milliseconds: 8883

                 Starting parallelSort
            Total time in milliseconds: 2143

      sort took 315% more time than parallelSort
```


Fig. 23.29

Comparing performance of `Arrays` methods `sort` and `parallelSort`.

Creating the Arrays

Line 14 uses `Random` method `ints` to create an `IntStream` of 100,000,000 random `int` values, then calls `IntStream` method `toArray` to place the values into an array. Line 15 calls method `clone` to make a copy of `array1` so that the calls to both `sort` and `parallelSort` work with the same set of values.

Timing `Arrays` Method `sort` with Date/Time API Classes `Instant` and

`Duration`

Lines 19 and 21 each call class `Instant`'s `static` method `now` to get the current time before and after the call to `sort`. To determine the difference between two `Instant`s, line 24 uses class `Duration`'s `static` method `between`, which returns a `Duration` object containing the time difference. Next, we call `Duration` method `toMillis` to get the difference in milliseconds.

Timing `Arrays` Method `parallelSort` with Date/Time API Classes `Instant` and `Duration`

Lines 29–31 time the call to `Arrays` method `parallelSort`. Then, lines 34–35 calculate the difference between the `Instant`s.

Displaying the Percentage Difference Between the Sorting Times

Lines 40–41 use a `NumberFormat` (package `java.text`)

to format the ratio of the sort times as a percentage. `NumberFormat` `static` method `getPercentInstance` returns a `NumberFormat` that's used to format a number as a percentage. `NumberFormat` method `format` per-forms the formatting. As you can see in the sample output, the `sort` method took over *300% more time* than `parallelSort` to sort the 100,000,000 random `int` values.5

5. Depending on your computer's setup, number of cores, whether your operating system is running in a virtual machine, etc., you could see significantly different performance.

Other Parallel Array Operations

In addition to method `parallelSort`, class `Arrays` now contains methods `parallelSetAll` and `parallelPrefix`, which perform the following tasks:

- `parallelSetAll`—Fills an array with values produced by a generator function that receives an `int` and returns a value of type `int`, `long`, `double` or the array's element type. Depending on which overload of method `parallelSetAll` is used, the generator function is an implementation of `IntUnaryOperator` (for `int` arrays), `IntToLongFunction` (for `long` arrays), `IntToDoubleFunction` (for `double` arrays) or `IntFunction<T>` (for arrays of any non-primitive type).
- `parallelPrefix`—Applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element. For example, consider:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x + y);
```

This call to `parallelPrefix` uses a `BinaryOperator` that *adds* two values. After the call completes, the array contains 1, 3, 6, 10 and 15. Similarly, the following call to `parallelPrefix`, uses a `BinaryOperator` that *multiplies* two values. After the call completes, the array contains 1, 2, 6, 24 and 120:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x * y);
```