```scala
package lectures
package algorithms

import org.scalameter._
import common._

object MergeSort {
  // a bit of reflection to access the private sort1 method, which takes an offset and an argument
  private val sort1 = {
    val method = scala.util.Sorting.getClass.getDeclaredMethod("sort1", classOf[Array[Int]], classOf[Int], classOf[Int])
    method.setAccessible(true)
    (xs: Array[Int], offset: Int, len: Int) => {
      method.invoke(scala.util.Sorting, xs, offset.asInstanceOf[AnyRef], len.asInstanceOf[AnyRef])
    }
  }

  def quickSort(xs: Array[Int], offset: Int, length: Int): Unit = {
    sort1(xs, offset, length)
  }

  @volatile var dummy: AnyRef = null

  def parMergeSort(xs: Array[Int], maxDepth: Int): Unit = {
    // 1) Allocate a helper array.
    // This step is a bottleneck, and takes:
    // - ~76x less time than a full quickSort without GCs (best time)
    // - ~46x less time than a full quickSort with GCs (average time)
    // Therefore:
    // - there is a almost no performance gain in executing allocation concurrently to the sort
    // - doing so would immensely complicate the algorithm
    val ys = new Array[Int](xs.length)
    dummy = ys

    // 2) Sort the elements.
    // The merge step has to do some copying, and is the main performance bottleneck of the algorithm.
    // This is due to the final merge call, which is a completely sequential pass.
    def merge(src: Array[Int], dst: Array[Int], from: Int, mid: Int, until: Int) {
      var left = from
      var right = mid
      var i = from
      while (left < mid && right < until) {
        while (left < mid && src(left) <= src(right)) {
          dst(i) = src(left)
          i += 1
          left += 1
        }
        while (right < until && src(right) <= src(left)) {
          dst(i) = src(right)
          i += 1
          right += 1
        }
      }
      while (left < mid) {
        dst(i) = src(left)
        i += 1
        left += 1
```

```
57          }
58          while (right < mid) {
59            dst(i) = src(right)
60            i += 1
61            right += 1
62          }
63        }
64        // Without the merge step, the sort phase parallelizes almost linearly.
65        // This is because the memory pressure is much lower than during copying in the third step.
66        def sort(from: Int, until: Int, depth: Int): Unit = {
67          if (depth == maxDepth) {
68            quickSort(xs, from, until - from)
69          } else {
70            val mid = (from + until) / 2
71            val right = task {
72              sort(mid, until, depth + 1)
73            }
74            sort(from, mid, depth + 1)
75            right.join()
76
77            val flip = (maxDepth - depth) % 2 == 0
78            val src = if (flip) ys else xs
79            val dst = if (flip) xs else ys
80            merge(src, dst, from, mid, until)
81          }
82        }
83        sort(0, xs.length, 0)
84
85        // 3) In parallel, copy the elements back into the source array.
86        // Executed sequentially, this step takes:
87        // - ~23x less time than a full quickSort without GCs (best time)
88        // - ~16x less time than a full quickSort with GCs (average time)
89        // There is a small potential gain in parallelizing copying.
90        // However, most Intel processors have a dual-channel memory controller,
91        // so parallel copying has very small performance benefits.
92        def copy(src: Array[Int], target: Array[Int], from: Int, until: Int, depth: Int): Unit = {
93          if (depth == maxDepth) {
94            Array.copy(src, from, target, from, until - from)
95          } else {
96            val mid = (from + until) / 2
97            val right = task {
98              copy(src, target, mid, until, depth + 1)
99            }
100           copy(src, target, from, mid, depth + 1)
101           right.join()
102         }
103       }
104       copy(ys, xs, 0, xs.length, 0)
105     }
106
107   val standardConfig = config(
108     Key.exec.minWarmupRuns ->> 20,
109     Key.exec.maxWarmupRuns ->> 60,
110     Key.exec.benchRuns ->> 60,
111     Key.verbose ->> true
112   ) withWarmer(new Warmer.Default)
```

```
113
114   def initialize(xs: Array[Int]) {
115     var i = 0
116     while (i < xs.length) {
117       xs(i) = i % 100
118       i += 1
119     }
120   }
121
122   def main(args: Array[String]) {
123     val length = 10000000
124     val maxDepth = 7
125     val xs = new Array[Int](length)
126     val seqtime = standardConfig setUp {
127       _ => initialize(xs)
128     } measure {
129       quickSort(xs, 0, xs.length)
130     }
131     println(s"sequential sum time: $seqtime ms")
132
133     val partime = standardConfig setUp {
134       _ => initialize(xs)
135     } measure {
136       parMergeSort(xs, maxDepth)
137     }
138     println(s"fork/join time: $partime ms")
139     println(s"speedup: ${seqtime / partime}")
140   }
141
142 }
```