Test rig: Intel i7 10870H, 8 cores, 16 threads, 16 GB Ram

Quick Sort:

Sequential vs naive parallelism

Method	ElementsCount	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
QuickSortSequential	1000000	291.5 ms	5.74 ms	8.77 ms	500.0000	500.0000	500.0000	3.82 MB
QuickSortParallel	1000000	318.8 ms	4.16 ms	3.90 ms	3000.0000	500.0000	500.0000	29.24 MB

Takeway: naive parallelism is not only slower, but consumes a huge amount of memory as well.

Limited Parallelism

Method	ElementsCount	Threshold	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
QuickSortParallelWithThreshold	1000000	10000	309.7 ms	5.21 ms	6.58 ms	500.0000	500.0000	500.0000	3.83 MB
QuickSortParallelWithThreshold	1000000	25000	296.4 ms	3.74 ms	3.50 ms	500.0000	500.0000	500.0000	3.82 MB
QuickSortParallelWithThreshold	1000000	50000	291.1 ms	4.21 ms	3.73 ms	500.0000	500.0000	500.0000	3.82 MB
QuickSortParallelWithThreshold	1000000	75000	283.4 ms	5.22 ms	4.89 ms	500.0000	500.0000	500.0000	3.82 MB
QuickSortParallelWithThreshold	1000000	100000	282.2 ms	5.57 ms	5.21 ms	500.0000	500.0000	500.0000	3.82 MB

Takeaway: for sorting 1 000 000 elements, using parallelism improves the performance when there is task spawn for chunks greater than 75 000 and 1 00 000, which makes sense because 1 000 000 / 75 000 is ~13 and 1 000 000 / 1 00 000 is 10, which is below the number of cores on the testing machine. However, the speedup is barely noticeable (10 ms).

Merge Sort

Sequential vs naive parallelism

Method	ElementsCount	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
MergeSortSequential	1000000	375.8 ms	7.49 ms	17.36 ms	69000.0000	7000.0000	3000.0000	557.64 MB
MergeSortParallel	1000000	415.8 ms	8.06 ms	10.76 ms	91000.0000	6000.0000	3000.0000	733.09 MB

Takeaway: same result as above. Unlimited parallelism is slower and consumes more memory.

Limited Parallelism

Method	ElementsCount	Threshold	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
MergeSortWithLimitedParallelism	1000000	10000	384.1 ms	7.15 ms	10.70 ms	69000.0000	6000.0000	3000.0000	557.65 MB
MergeSortWithLimitedParallelism	1000000	25000	366.6 ms	7.30 ms	11.15 ms	69000.0000	6000.0000	3000.0000	557.65 MB
MergeSortWithLimitedParallelism	1000000	50000	356.6 ms	7.06 ms	15.49 ms	69000.0000	7000.0000	3000.0000	557.64 MB
MergeSortWithLimitedParallelism	1000000	75000	349.9 ms	6.78 ms	12.74 ms	69000.0000	6000.0000	3000.0000	557.64 MB
MergeSortWithLimitedParallelism	1000000	100000	348.7 ms	6.82 ms	11.58 ms	69000.0000	7000.0000	3000.0000	557.62 MB

Interesting observation: a small speedup was reached with 25 000 threshold out of 1 000 000 elements, which would result in 40 threads. The speedup with 1 00 000 threshold is also more noticeable (25 ms) and the memory overhead is neglectable because merge sort itself consume too much memory.

Running the same benchmarks with 10 million elements instead of 1 million lead to the following results: the results are kind of random. There is a small speedup with 25000 and 50000 threshold.

Method	ElementsCount	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
MergeSortSequential	10000000	4.024 s	0.0353 s	0.0392 s	773000.0000	38000.0000	4000.0000	6.33 GB
MergeSortParallel	10000000	4.448 s	0.0303 s	0.0253 s	993000.0000	36000.0000	4000.0000	8.04 GB

Method	ElementsCount	Threshold	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
MergeSortWithLimitedParallelism	10000000	10000	4.109 s	0.0679 s	0.0635 s	773000.0000	37000.0000	4000.0000	6.33 GB
MergeSortWithLimitedParallelism	10000000	25000	3.860 s	0.0414 s	0.0346 s	773000.0000	38000.0000	4000.0000	6.33 GB
MergeSortWithLimitedParallelism	10000000	50000	3.843 s	0.0725 s	0.1820 s	773000.0000	35000.0000	4000.0000	6.33 GB
MergeSortWithLimitedParallelism	10000000	75000	4.176 s	0.0800 s	0.0709 s	773000.0000	39000.0000	4000.0000	6.33 GB
MergeSortWithLimitedParallelism	10000000	100000	4.091 s	0.0687 s	0.0941 s	773000.0000	38000.0000	4000.0000	6.33 GB