Program Structures & Algorithms

Spring 2022

Assignment – 4

Name – Jayesh Kumar Khattar

NUID – 001568947

Task –

Please see the presentation on *Assignment on Parallel Sorting* under the *Exams. etc.* module.

Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (*t*) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of *lg t* is reached).
3. An appropriate combination of these.

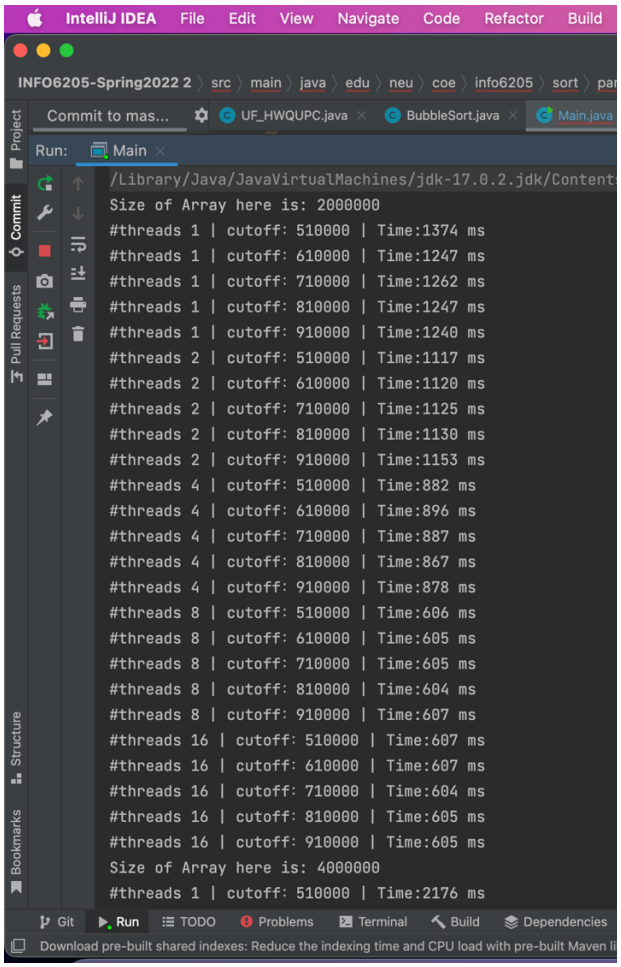
There is a *Main* class and the *ParSort* class in the *sort.par* package of the INFO6205 repository. The *Main* class can be used as is but the *ParSort* class needs to be implemented where you see "TODO..." [it turns out that these TODOs are already implemented].

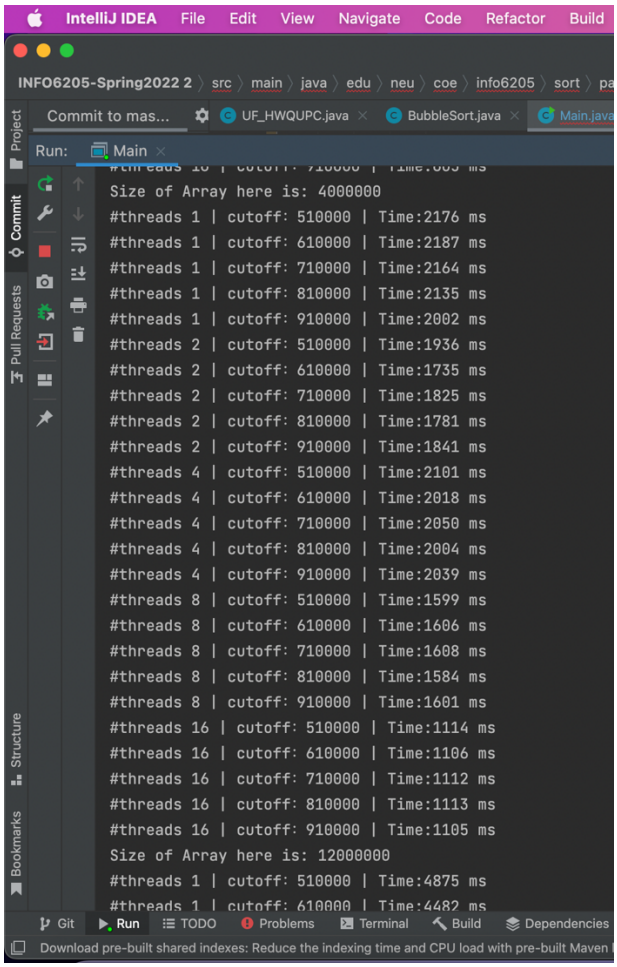
Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.

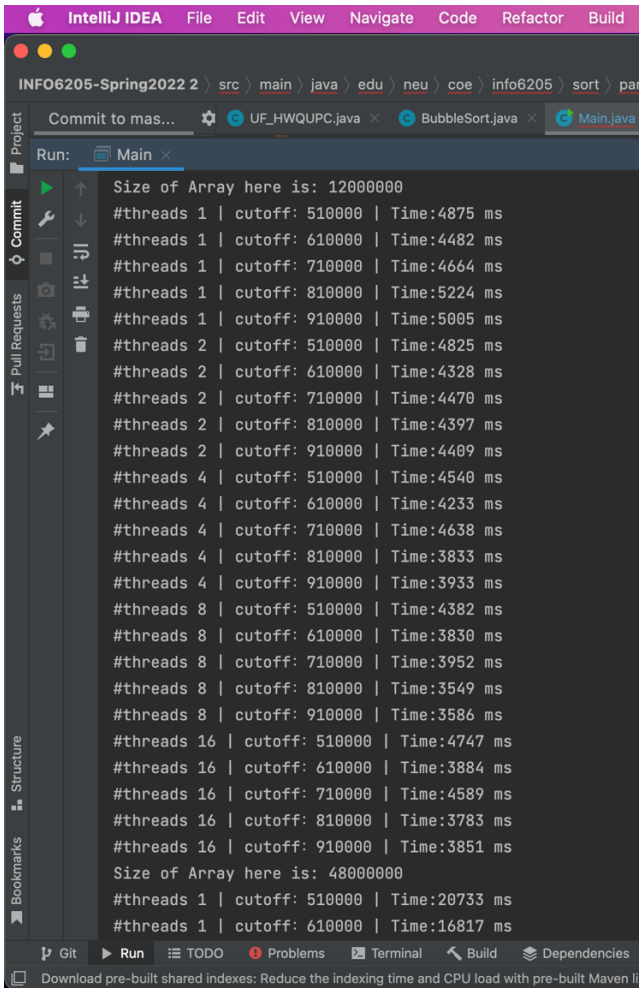
You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

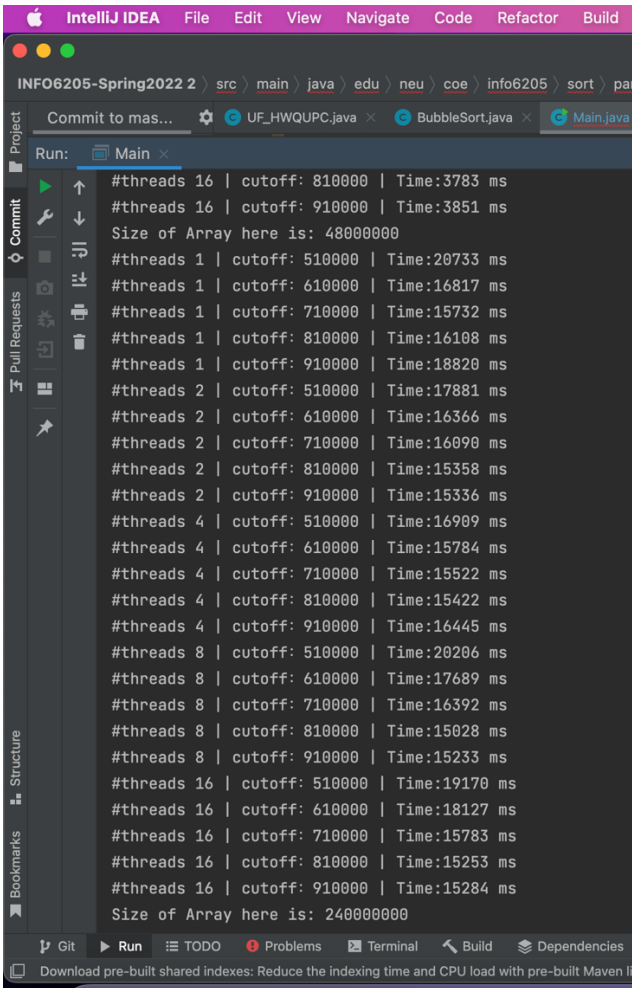
Output –

Output of the above code is shown in the below picture. We have executed experiments with different threads and array sizes, and it shows the following output –









Relationship Conclusion –

- The system used in this assignment is an 8 core CPU. Since we can utilize 2 threads per core at a time, so at max we can run 16 threads together.
- We can see that as the number of threads increases, the time taken to execute the system reduces.
- After running experiments with different threads, we can observe that the performance increase with more threads but with large values and large # of threads, it's not the same. With array size equal to 12 million and high cutoff, 8 threads perform better than 16 threads and array size equal to 24 million eventually 4 thread performs best with large values.

For maximum depth (d) and threads (t), we can observe that –

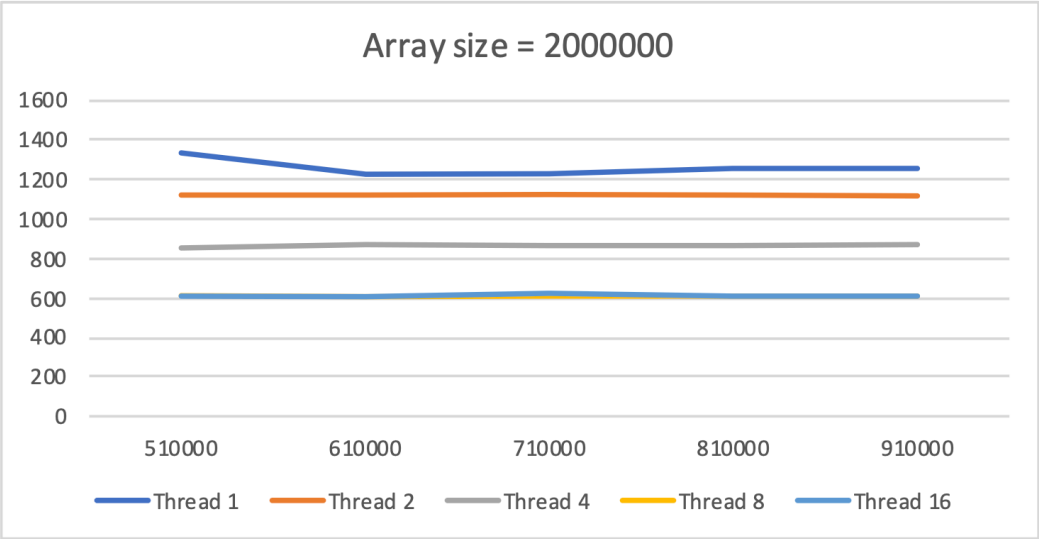
$$t = 2^d$$

Hence, maximum depth possible is $\lg\left(\frac{array\ size}{cutoff}\right)$

Evidence –

Array size	2000000

Cutoff	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
510000	1336	1120	854	612	610
610000	1228	1124	871	607	607
710000	1229	1125	870	610	624
810000	1254	1124	870	609	608
910000	1251	1118	871	608	612

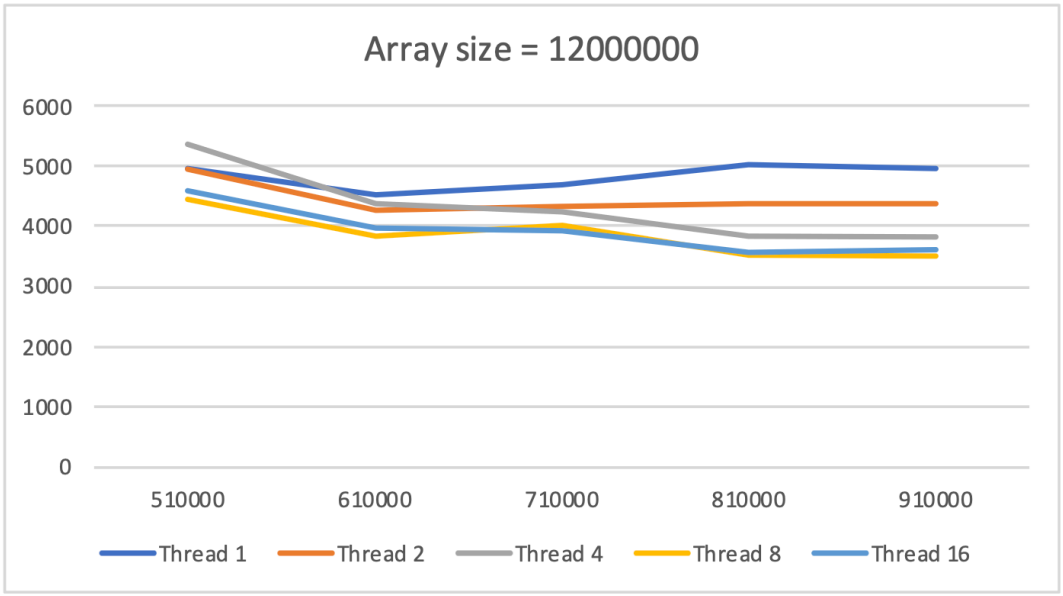


Array size	4000000

Cutoff	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
510000	2144	2162	2110	1594	1145
610000	2020	1953	2061	1599	1117
710000	1981	1932	2065	1609	1138
810000	2007	1936	2063	1590	1121
910000	2051	1831	2012	1611	1135

Array size	12000000

Cutoff	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
510000	4944	4945	5358	4445	4587
610000	4520	4264	4388	3835	3983
710000	4709	4341	4259	4035	3932
810000	5020	4387	3828	3516	3566
910000	4966	4370	3822	3505	3610



Array size	48000000

Cutoff	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
510000	18471	19618	17214	17207	18812
610000	17461	16498	15892	15931	16193
710000	17329	16132	15448	15226	17052
810000	15821	16199	15190	15753	16131
910000	17151	16063	15201	15977	16016