Sayali Mahajan – (001576540)

Program Structures & Algorithms
Summer 2021
Assignment No. 5

Task

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.

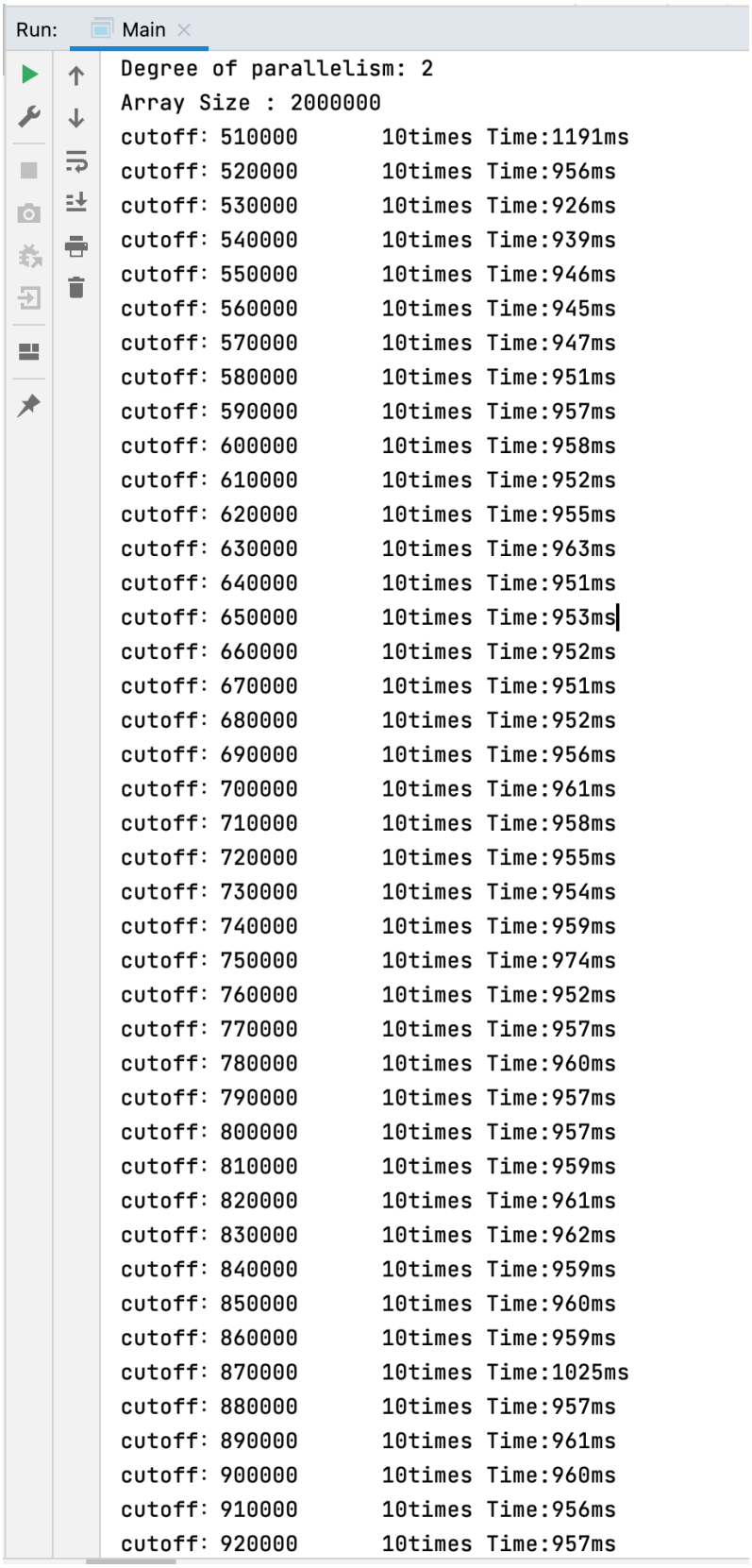
1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. 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.

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.

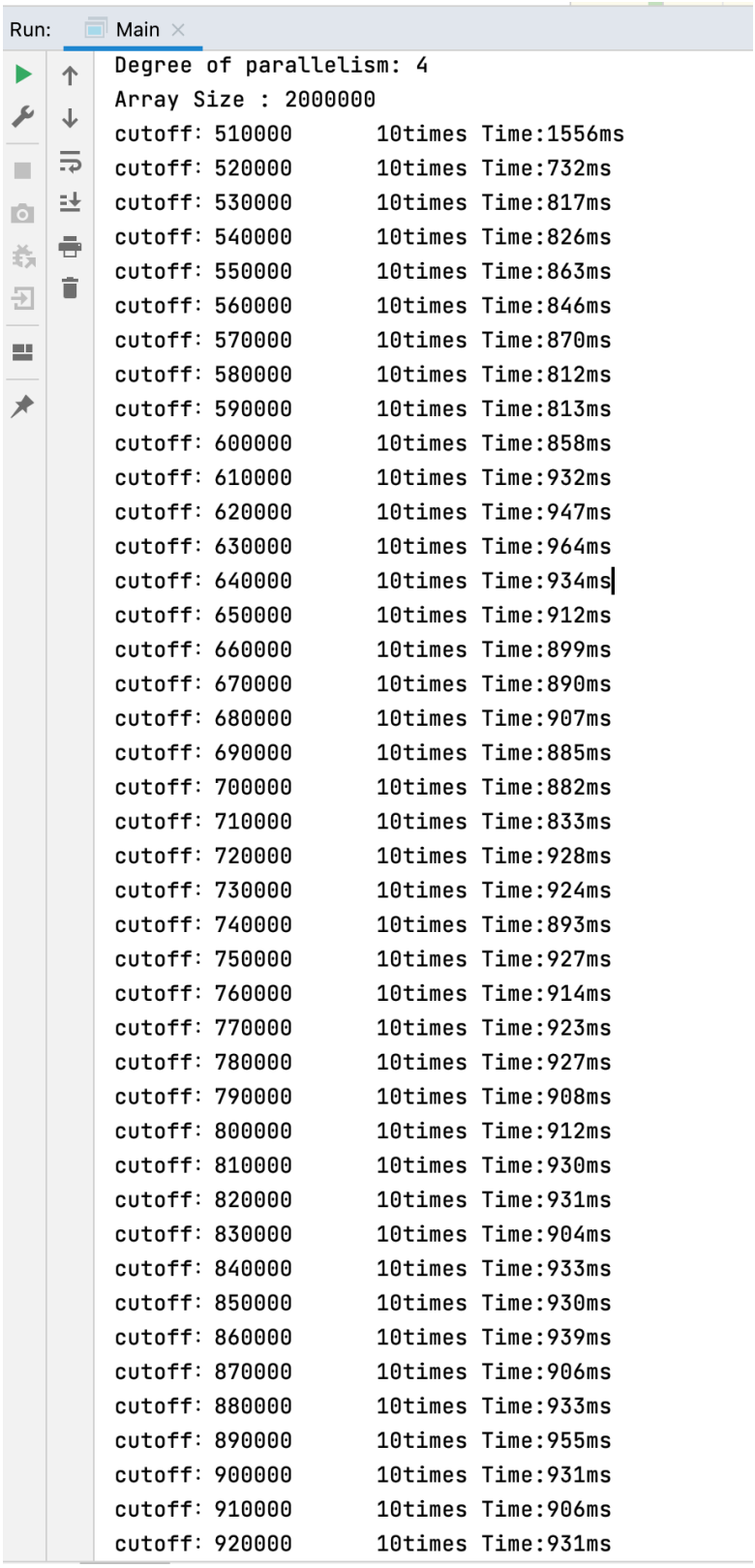
Output

First I wrote code in main.java for thread count (power of 2) and fixed array size.

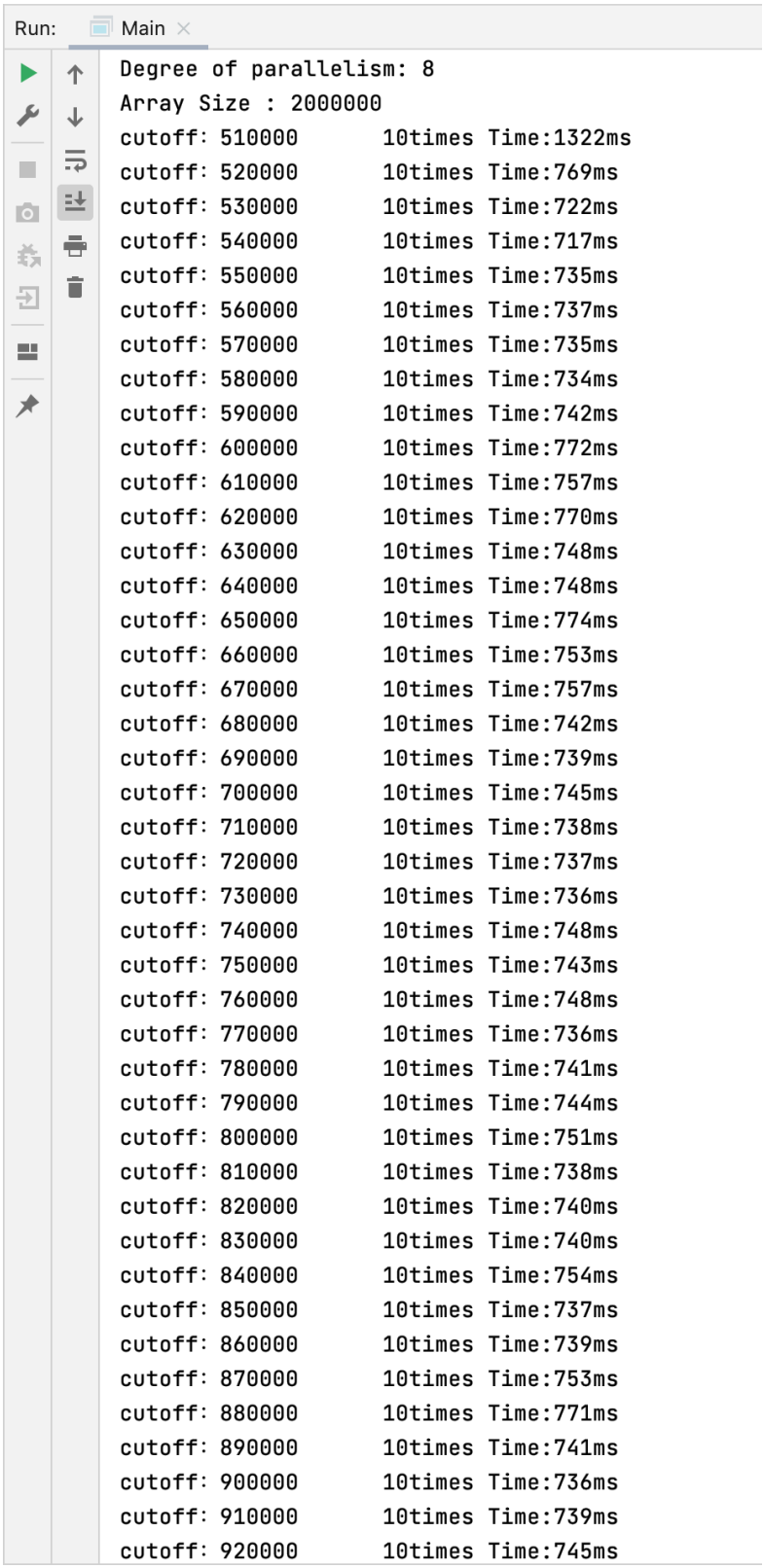
1. **Available thread count : 2 ; Array size: 2000000 ; Sorted 10 times(average)**



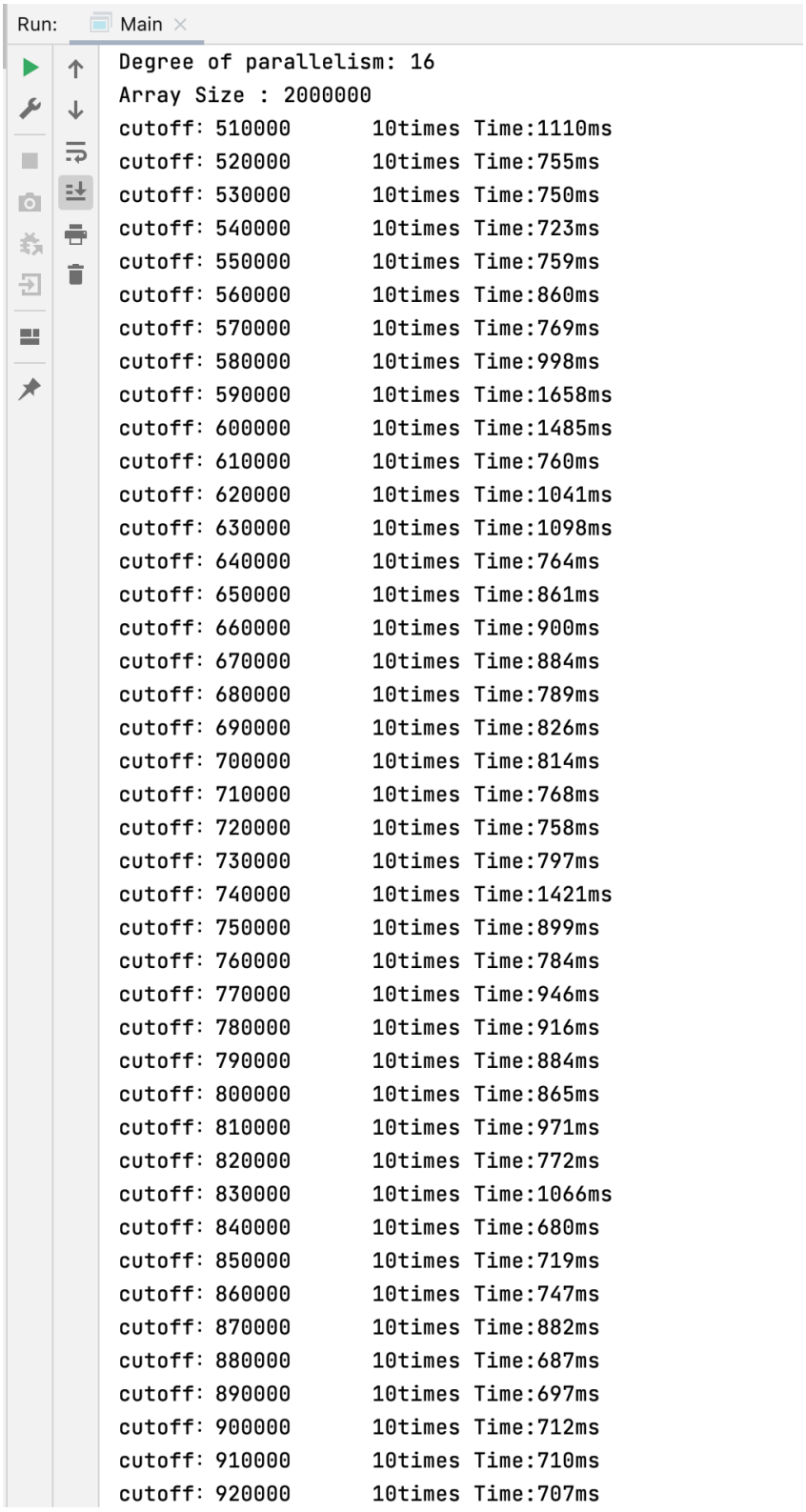
2. **Available thread count : 4 ; Array size: 2000000 ; Sorted 10 times(average)**



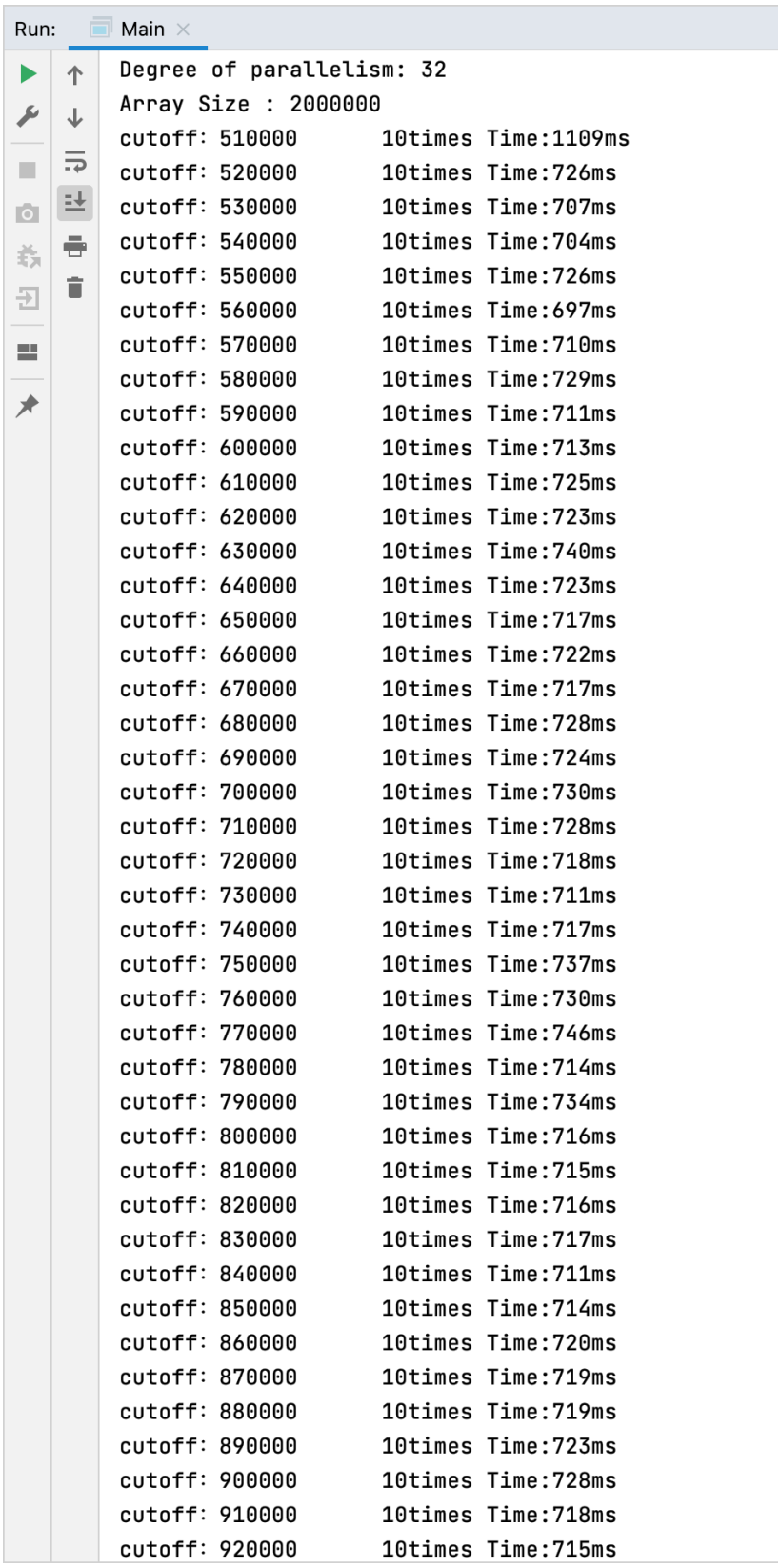
3. **Available thread count : 8 ; Array size: 2000000 ; Sorted 10 times(average)**



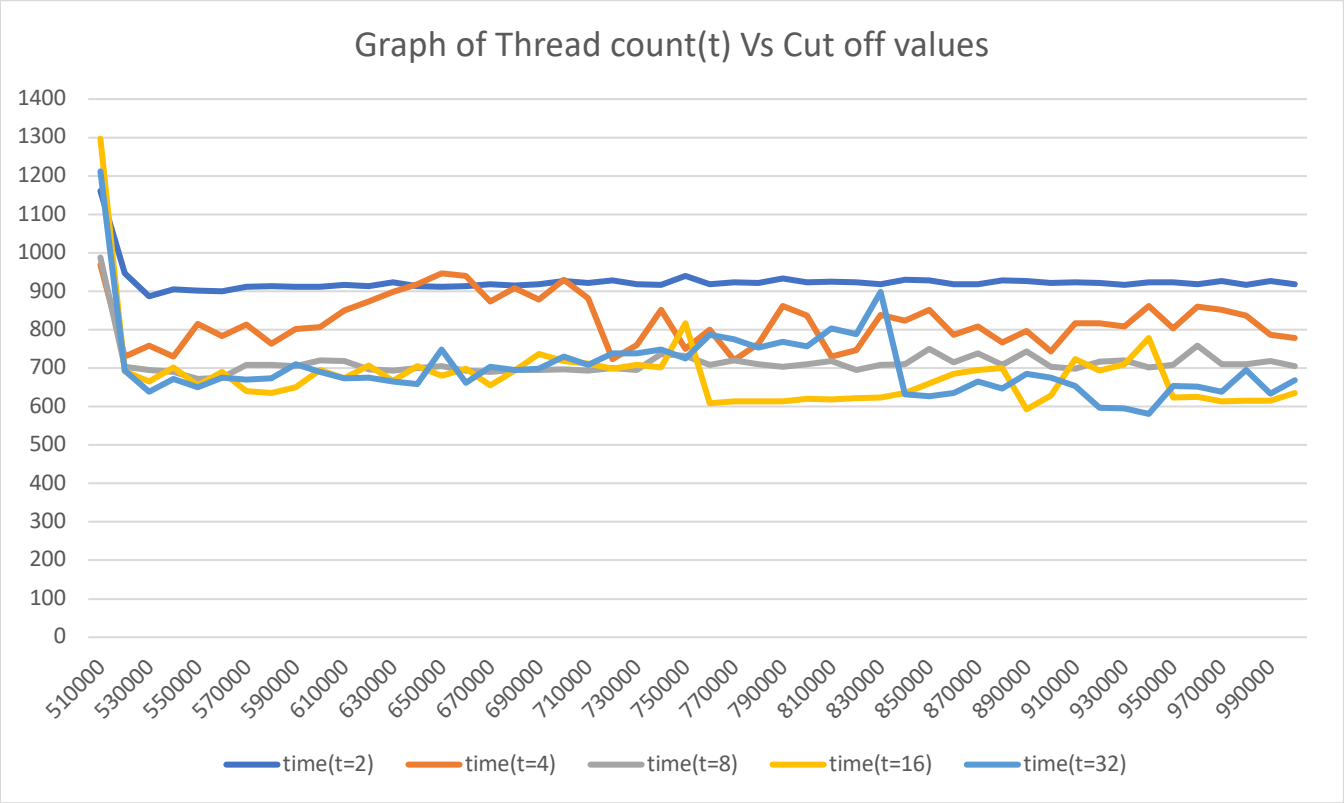
4. **Available thread count : 16 ; Array size: 2000000 ; Sorted 10 times(average)**



5. **Available thread count : 32 ; Array size: 2000000 ; Sorted 10 times(average)**



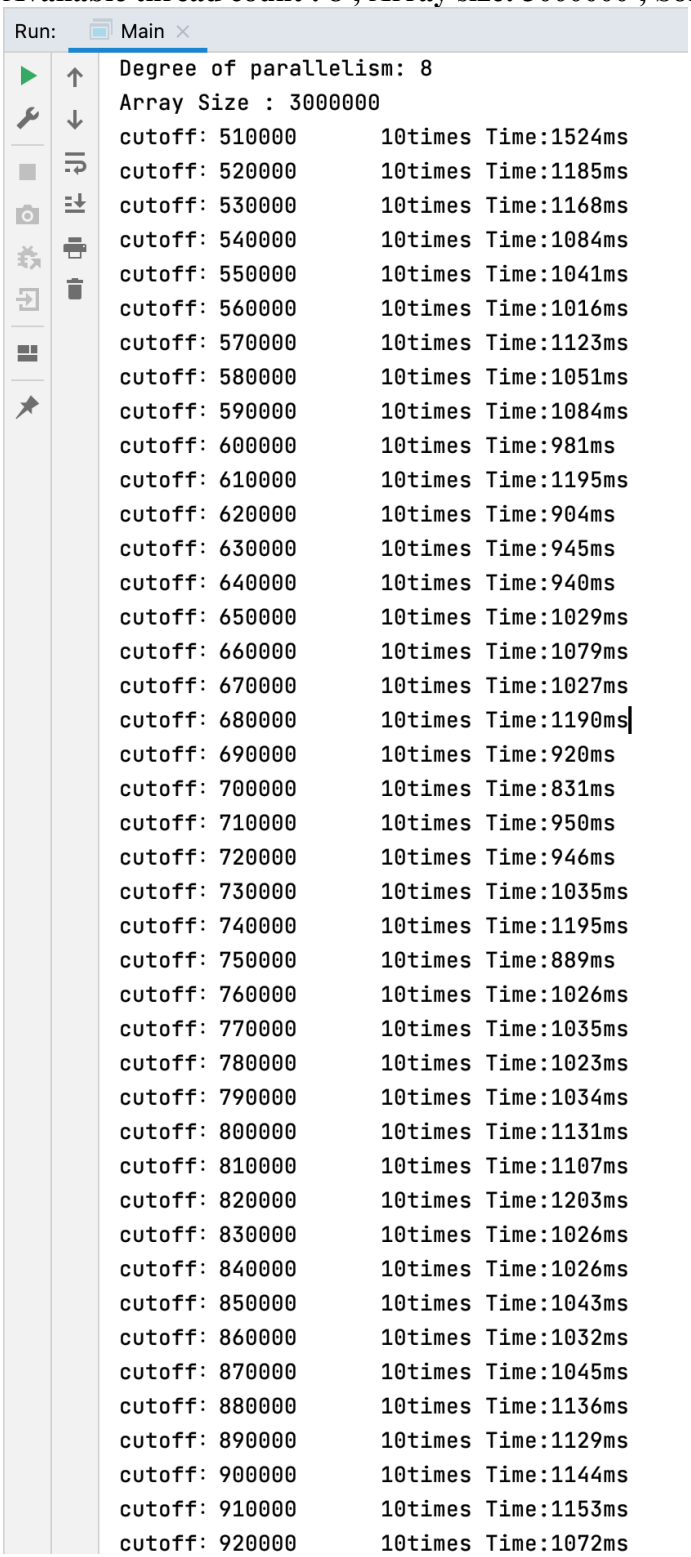
Graphical representation of evidence :



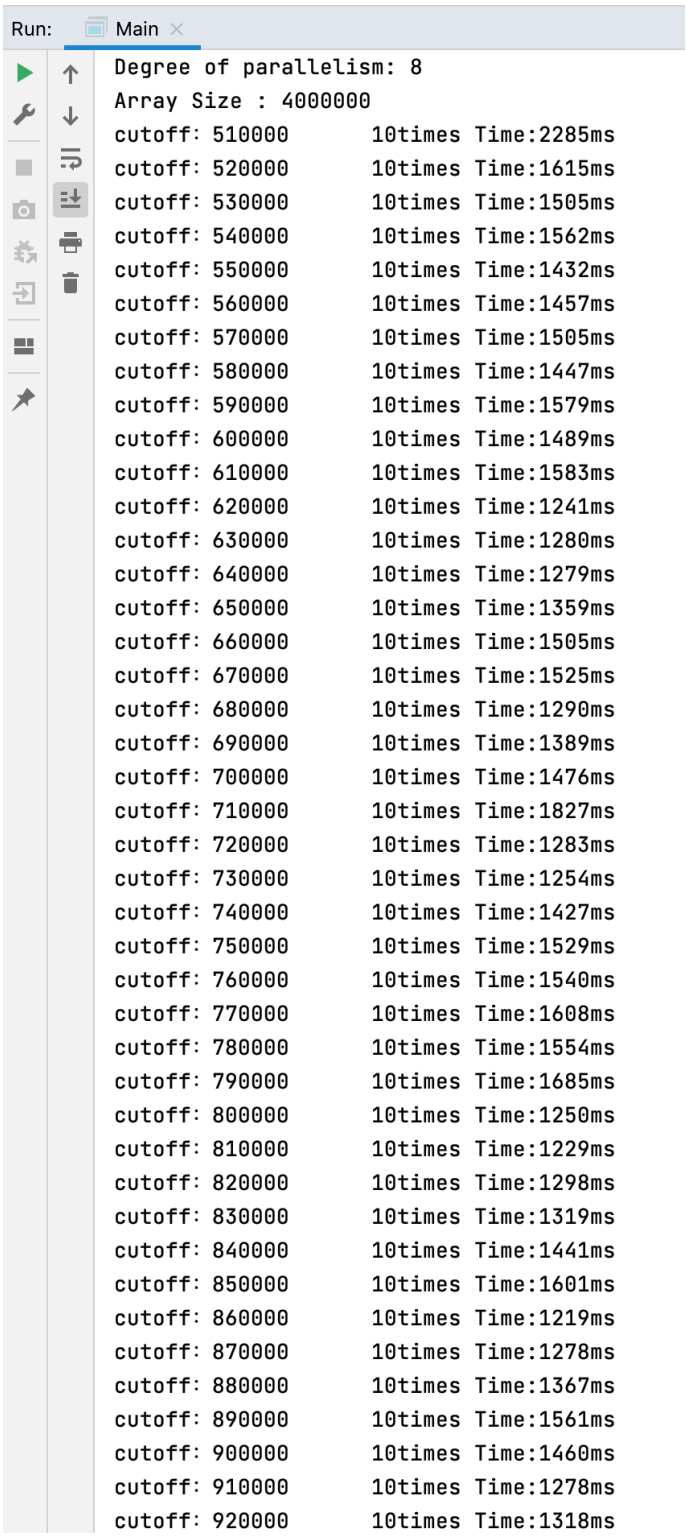
Above experiment carried out with cut off range from 510000 – 920000 and array size of 2000000 and thread count from 2- 32.
From above experiments, it is observed that, ideal cut off value between 580000-650000. Hence, 620000 and it is most efficient when thread count 8.

Let's re-confirm this by carrying out experiment with thread count as 8 and range 510000-920000 for different sizes of array by modifying main.java with respect to array size.

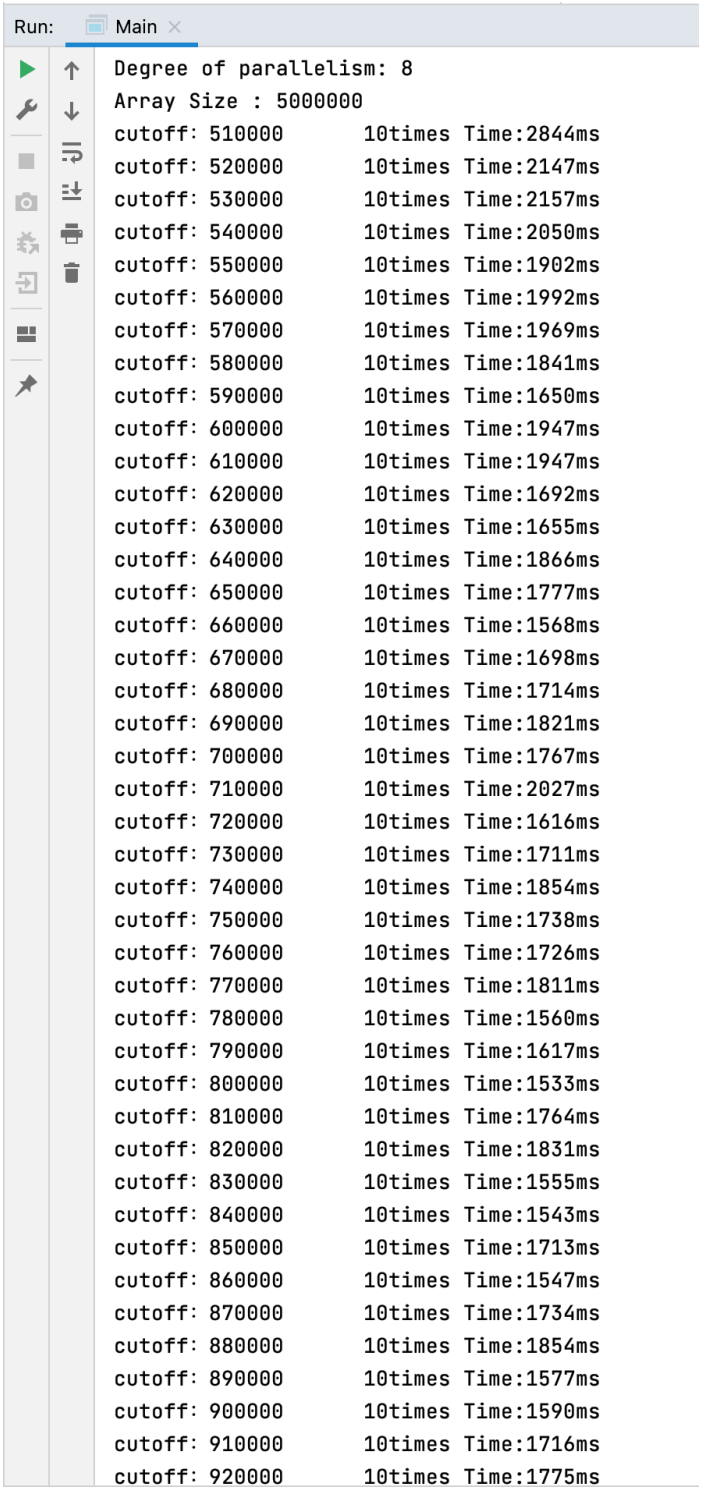
Available thread count : 8 ; Array size: 3000000 ; Sorted 10 times(average)



Available thread count : 8 ; Array size: 4000000 ; Sorted 10 times(average)



Available thread count : 8 ; Array size: 5000000 ; Sorted 10 times(average)



Conclusion

From experiments carried out above, it confirms that, even when we increased the Array size from 2000000 till 5000000 by difference of 1000000, It might have taken more time but when cut off value is 620000 and thread count is 8, algorithm work with the most efficiency.

Cut off Time : 620000
Thread count : 8