**Varun Rajamudi – (002108570)**

**Program Structures & Algorithms Fall 2021**

**Assignment No. 5 (Parallel Sorting)**

**Task**

Implementing 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**

I have written code in **Main.java** for thread count (power of 2) by varying array size.

**Thread count: 2; Array size: 2000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 4; Array size: 2000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 8; Array size: 2000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 16; Array size: 2000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 32; Array size: 2000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 16; Array size: 3000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 16; Array size: 4000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 16; Array size: 5000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Thread count: 16 ; Array size: 10000000; Sorted 10 times**

Graphical user interface, text

Description automatically generated

**Conclusion**

From experiments carried out above, it confirms that, even when we increased the Array size from 2000000, 3000000, 4000000, 5000000, 10000000, it might have taken more time but when cut off value is 650000 and thread count is 16, algorithm work with the most efficiency.

**Cut off Time: 650000**

**Thread count: 16**