Assignment – 5

Program Structure and Algorithms

Pranav Agarwal

NUID – 001099801

1. Parallel Sort algorithm used –

The program aims to sort an array leveraging the multithreading capability of the processor to improve performance. Merge Sort is a good candidate for this, as it consists of distinct segments that can be processed in parallel then merged together. As such, bottom up merge sort with multithreading using a fixed thread pool has been used. Assume size of the array is N and cutoff size is C.

Algorithm:

1. Create a fixed thread pool of desired size
2. Create a callable task queue
3. Create N/C System sort tasks each operating on C elements and push them to the queue
4. Call InvokeAll on the queue to complete the tasks in parallel till they are all completed
5. Create merge tasks that operate on twice as many elements as the previous cutoff and push them to the queue
6. Call InvokeAll on the queue to complete the tasks in parallel till they are all completed
7. Repeat steps 5-6 until the cutoff size includes the whole array

2. Benchmark Data –

The benchmarking was done on an Intel i5-6200U with 2 cores and 4 threads.
Thread Pool sizes tested were 1,2,4,8.
Array Sizes tested were from 2^{20} to 2^{24}.
Cutoffs tested were from 2^{14} to 2^{24}.

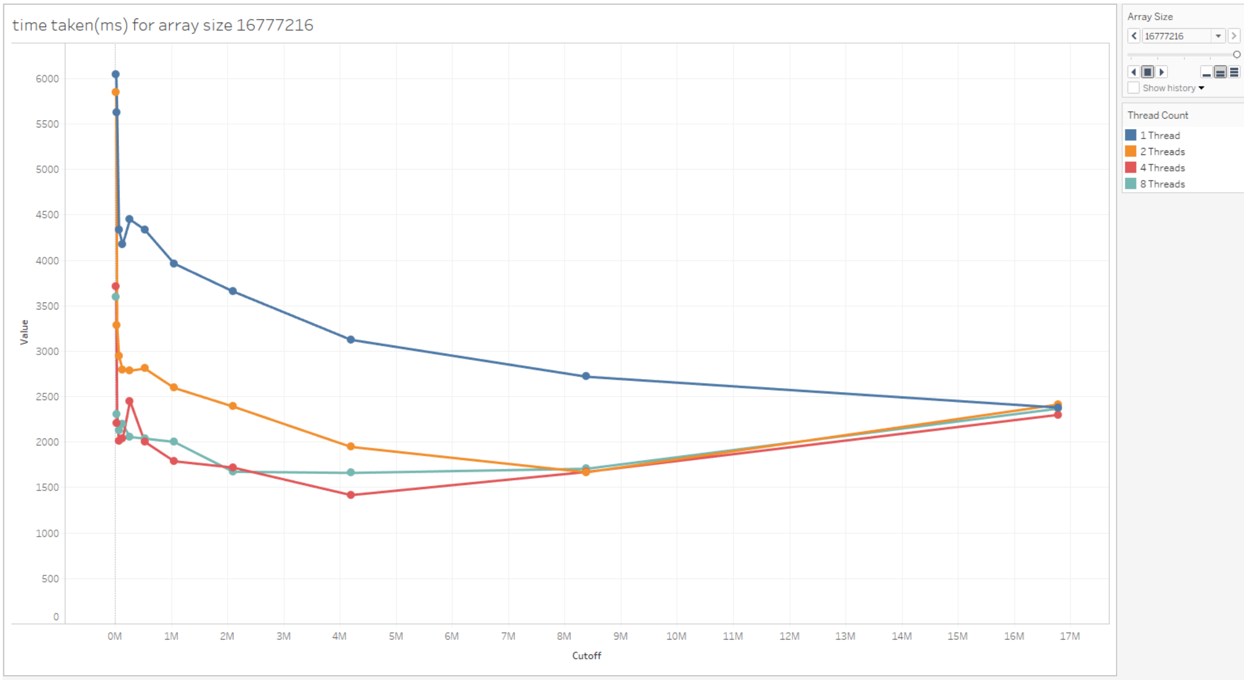
The goal is to measure time taken to sort with respect to:

1. Size of the array
2. Cutoff size
3. Thread count in the pool

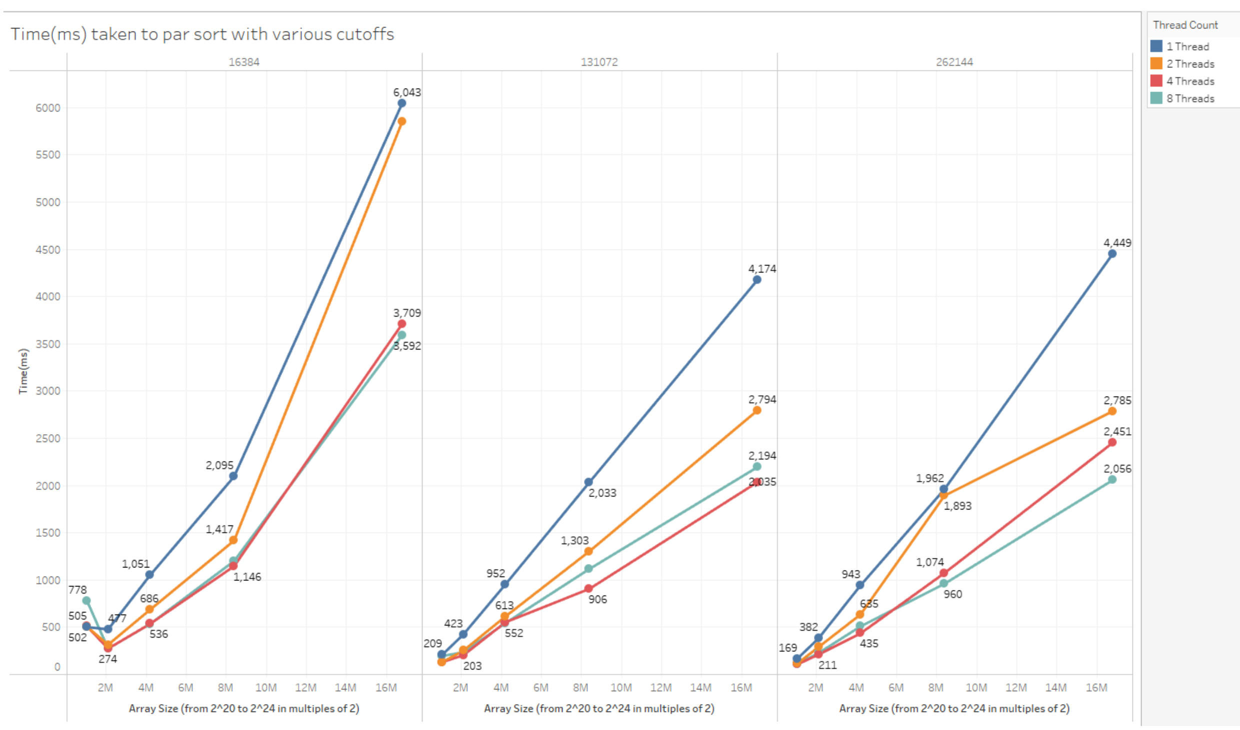
Raw Data –

data_all.csv

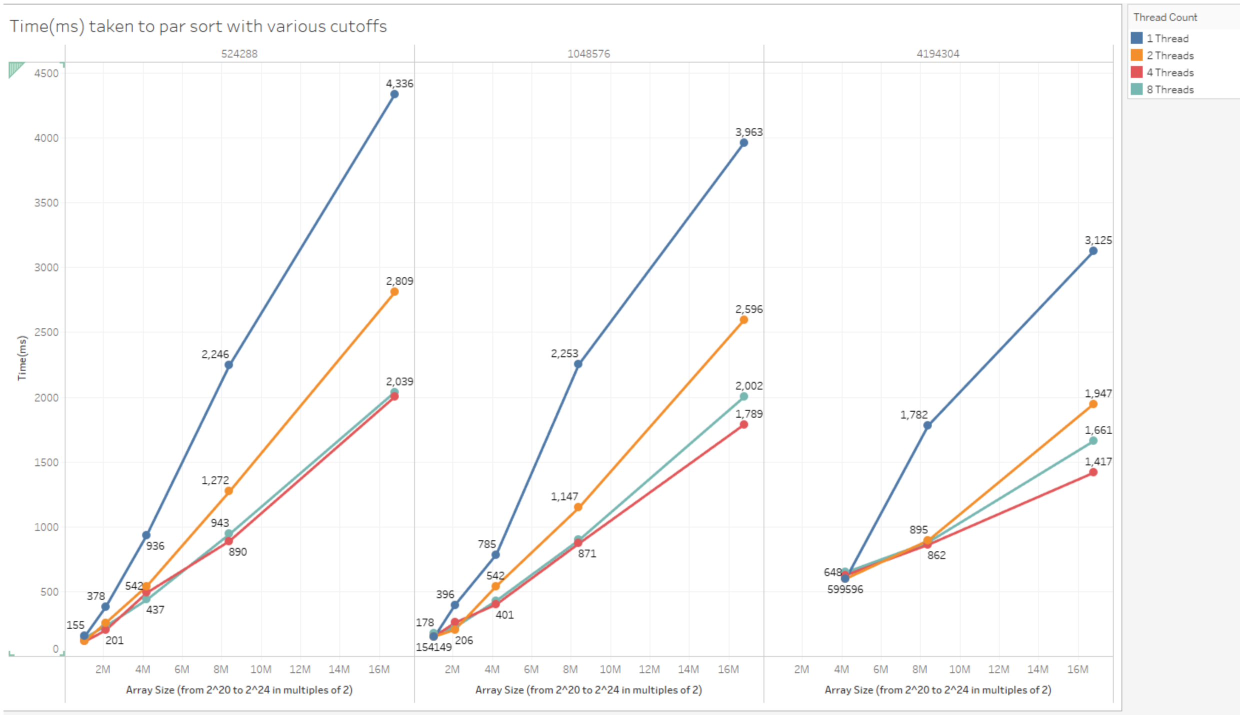
Graphs –



1 Time taken vs Cutoff Size



***2** Time Taken vs Array Size/Number of Threads*



3 Time Taken vs Array Size/Number of Threads

3. Observations/Conclusions –

- For all graphs in fig 2 and fig 3, the growth of time w.r.t size of the array is approximately NlgN. The only thing that the cutoff and thread pool size changes is the constant multiplier.
- The time taken to sort on a single thread decreases as cutoff size increases (fig 1). This makes sense as more of the array is sorted using the highly efficient system sort rather than the custom merge.
- However, for sorting on multiple threads, the smaller cutoff means more segments that can be processed in parallel improving the time. For 4 threads, the trend almost reverses where the time taken increases with cutoff size as it becomes less able to leverage multithreading (fig 1).
- There is a consistent and considerable performance difference on the number of threads used as seen in fig 2 and fig 3. The time taken for 2 threads is approximately 0.6 that of 1 thread, and the time taken for 4 threads is approximately 0.4 that of 1 thread.
- However, since the processor used has only 4 threads, using a pool of 8 threads doesn't improve the performance and, in some cases, even degrades it as the thread overhead is increased without increase in multiprocessing.

4. Notes regarding assignment –

- The simulation code is in bottomUpPar.java in the par folder in sort.
- The simulation benchmark code is in bottomUpSortDriver.java in the par folder in sort.
- The “Insurance policy” optimization on merge sort has been used.
- The simulation data is stored in results/parsort/ . There are separate csv files for different pool sizes.
- The provided par sort and main classes have not been used, rather a new class has been made from scratch. I have spoken with the professor already regarding this and he has allowed me to try out this bottom up algo.