Pallak Singh (NUID: 001063095) - Section 02

INFO 6205
Program Structures & Algorithms
Fall 2020
Assignment No 5

- **Task**

  The task given to us was to implement a parallel sorting algorithm by leveraging concurrent programming in Java. We use CompletableFuture provided by Java 8 which is a specific implementation of the Future interface which allows users to explicitly complete an asynchronous computation. In this assignment, we conduct various experiments to find a suitable value of the number of threads and the cutoff value for which we should stop our parallel computation and resort to system sort instead.

- **Implementation**

  In ParSort.java, I implemented the algorithm for merging (or combining) the two parallel partition computations and provided a variable for explicitly setting the thread pool size by using ForkJoinPool. Essentially, in this parallel sorting algorithm, we are simply doing merge sort in parallel. I used ForkJoinPool which is a specific implementation of Executor Service which provides efficient processing by means of work stealing where threads in the pool attempt to execute tasks created by other threads in the pool. In Main.java, I have different methods to implement different stages of my experiment but the main itself simply runs the parallel sorting algorithm at different thread pool sizes ranging from 1 to 16 (increasing by a factor of 2) and different cutoffs (ranging from 500,000 to 1,000,000) for an array size of 2,000,000. I plot results for every stage of my experiment using Plotter.java.

  Other stages of my experiment are conducted in methods:

  1. *experimentWithFixedThreadSizeVaryingCutoffs():* Choose a specific thread size and plot the cutoff vs time graph
  2. *experimentWithVaryingThreadSizeFixedCutoffs():* Choose a specific cutoff and plot the time vs size graph with varying thread sizes

- **Output**

  I ran my parallel sorting algorithm with cutoffs ranging from 500,000 to 1,000,000 for array sizes: 1,000,000, 2,000,000 and 4,000,000 and thread pool sizes: 1, 2, 4, 8, 16 at 20 runs each. I documented the average time taken for each array at each thread pool size, the minimum time taken and maximum time taken, the corresponding cutoff at which that value was taken and plotted the Cutoff vs Time graph for each. The results are presented in Table 3.1, 3.2, 3.3 and the graphs are presented in Figure 3.1, 3.2, 3.3.

Threads	Average	Minimum Time	Minimum Cutoff	Maximum Time	Maximum Cutoff
1	104 ms	97 ms	600,000	129 ms	530,000
2	63 ms	59 ms	590,000	67 ms	510,000
4	66 ms	63 ms	630,000	74 ms	730,000
8	63 ms	59 ms	750,000	77 ms	520,000
16	63 ms	59 ms	560,000	77 ms	910,000

Table 3.1: Results recording average time, minimum and maximum time and the corresponding cutoffs when the array size = 1,000,000

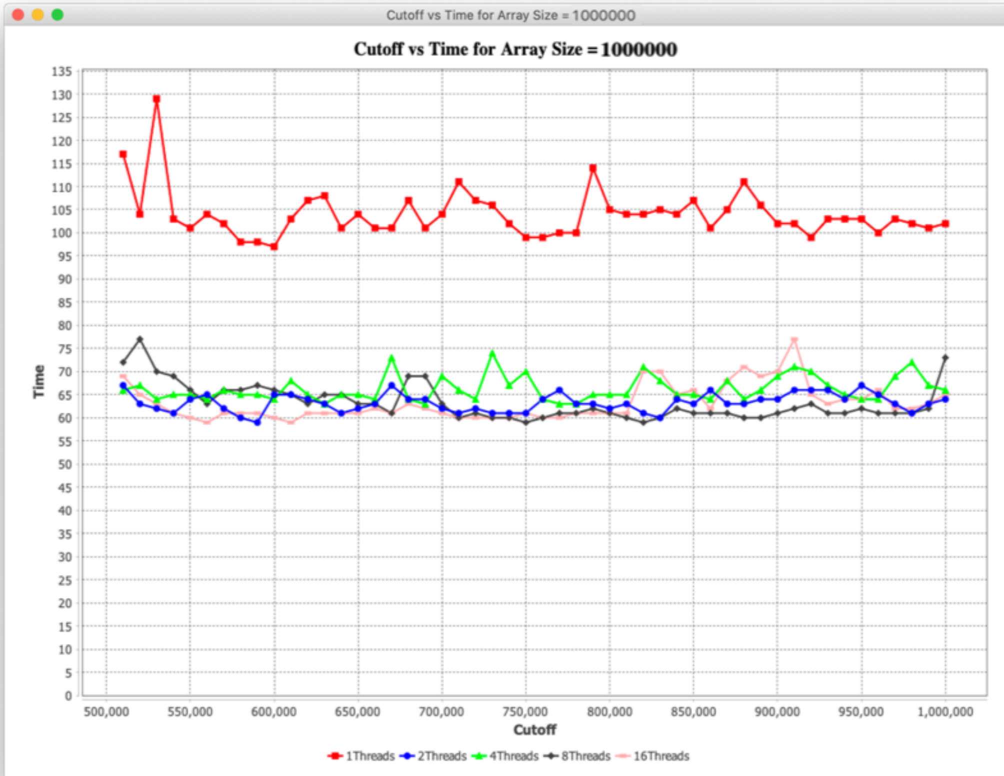

Figure 3.1: Graph recording Cutoff vs Time for array size = 1,000,000 at 20 runs

Threads	Average	Minimum Time	Minimum Cutoff	Maximum Time	Maximum Cutoff
1	195 ms	186 ms	970,000	216 ms	700,000
2	185 ms	176 ms	970,000	198 ms	950,000
4	130 ms	125 ms	700,000	143 ms	580,000
8	110 ms	103 ms	830,000	123 ms	530,000
16	116 ms	102 ms	102,000	171 ms	800,000

Table 3.2: Results recording average time, minimum and maximum time and the corresponding cutoffs when the array size = 2,000,000

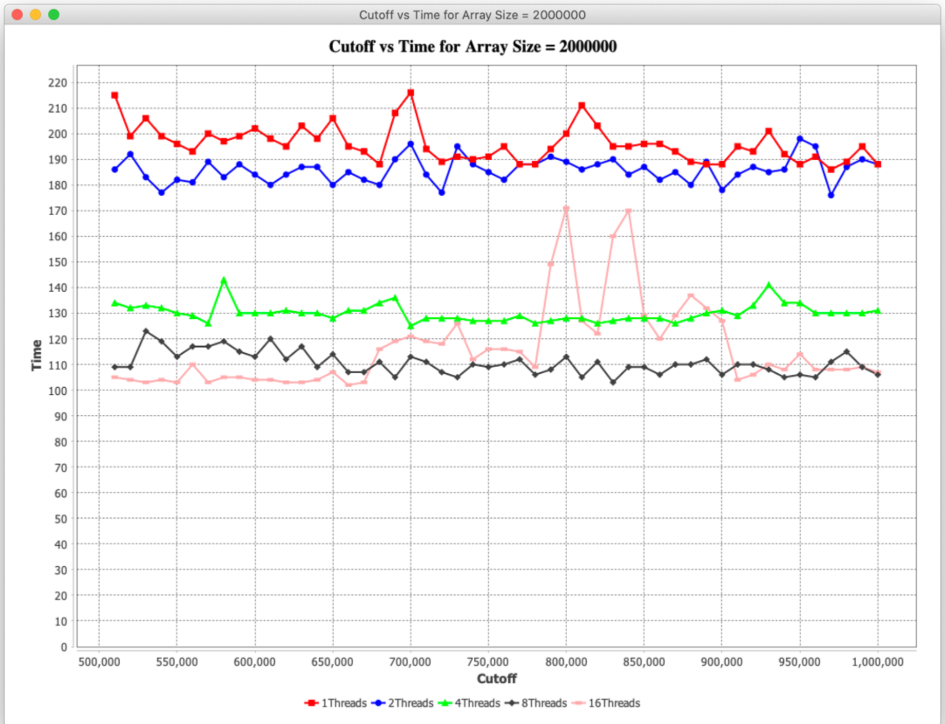

Figure 3.2: Graph recording Cutoff vs Time for array size = 2,000,000 at 20 runs

Threads	Average	Minimum Time	Minimum Cutoff	Maximum Time	Maximum Cutoff
1	383 ms	350 ms	910,000	538 ms	610,000
2	412 ms	360 ms	780,000	500 ms	840,000
4	352 ms	336 ms	880,000	408 ms	540,000
8	349 ms	310 ms	940,000	393 ms	550,000
16	291 ms	249 ms	660,000	508 ms	720,000

Table 2.3: Results recording average time, minimum and maximum time and the corresponding cutoffs when the array size = 4,000,000

Pallak Singh (NUID: 001063095) - Section 02

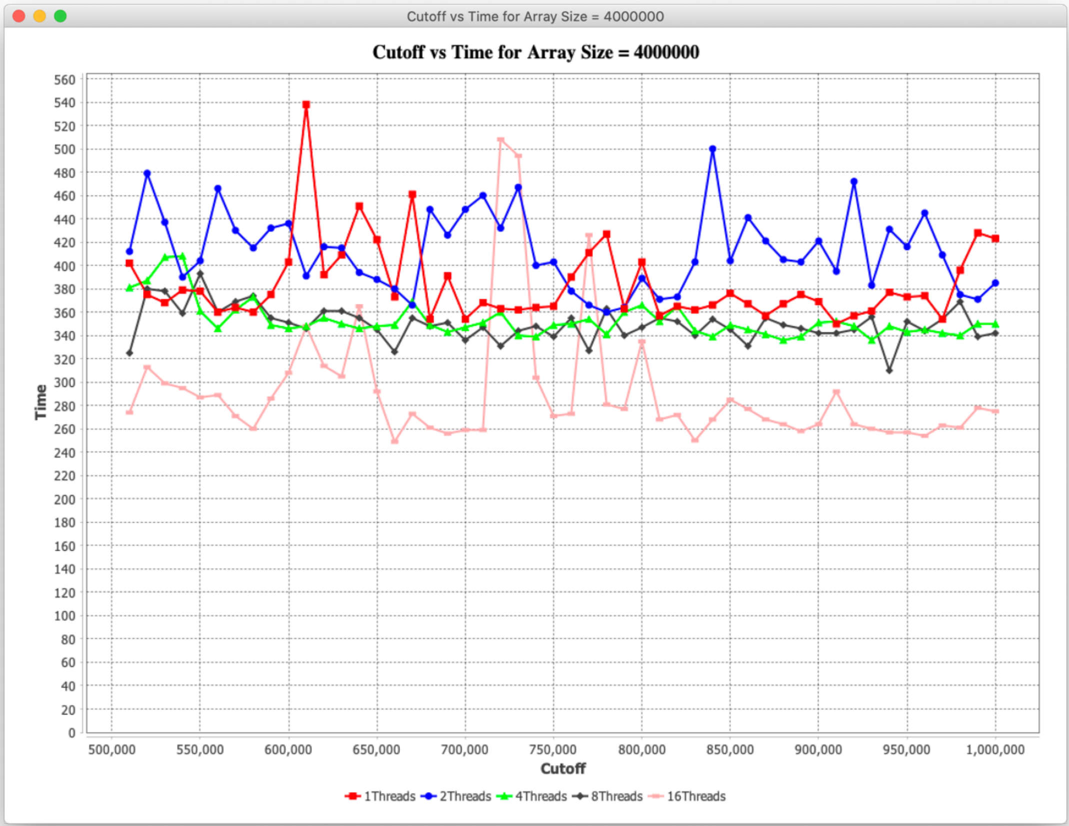

Figure 3.3 Graph recording Cutoff vs Time for array size = 4,000,000 at 20 runs

Console Output for Array Size = 4,000,000

```
cutoff and threads = 1: 510000          20 times Time:8053ms
cutoff and threads = 1: 520000          20 times Time:7514ms
cutoff and threads = 1: 530000          20 times Time:7374ms
cutoff and threads = 1: 540000          20 times Time:7590ms
cutoff and threads = 1: 550000          20 times Time:7573ms
cutoff and threads = 1: 560000          20 times Time:7212ms
cutoff and threads = 1: 570000          20 times Time:7292ms
cutoff and threads = 1: 580000          20 times Time:7216ms
cutoff and threads = 1: 590000          20 times Time:7507ms
…
…
…
Average time taken over all cutoffs for thread count: 1 = 383
Minimum time taken over all cutoffs for thread count: 1 = 350 at cutoff: 910000
Maximum time taken over all cutoffs for thread count: 1 = 538 at cutoff: 610000
…
…
…
cutoff and threads = 16: 950000         20 times Time:5143ms
cutoff and threads = 16: 960000         20 times Time:5094ms
cutoff and threads = 16: 970000         20 times Time:5278ms
cutoff and threads = 16: 980000         20 times Time:5221ms
cutoff and threads = 16: 990000         20 times Time:5565ms
cutoff and threads = 16: 1000000                20 times Time:5514ms
Average time taken over all cutoffs for thread count: 16 = 291
Minimum time taken over all cutoffs for thread count: 16 = 249 at cutoff: 660000
Maximum time taken over all cutoffs for thread count: 16 = 508 at cutoff: 720000
```

- **Conclusions and Supporting Evidence**

  The idea behind finding the cutoff at which the time was minimum and maximum was to observe a pattern but from the above results we can see that there is no specific value at which we are to find a minimum or a maximum. Now, the question arises that is our cutoff range not big enough? To answer this question, I increased my range to test the time for cutoff sizes from 100,000 to 1,000,000 for a fixed array size of 1,000,000 at a fixed thread pool size of 16. I chose 16 because from the results above it was clear that the minimum time is always found at thread pool size of 16.

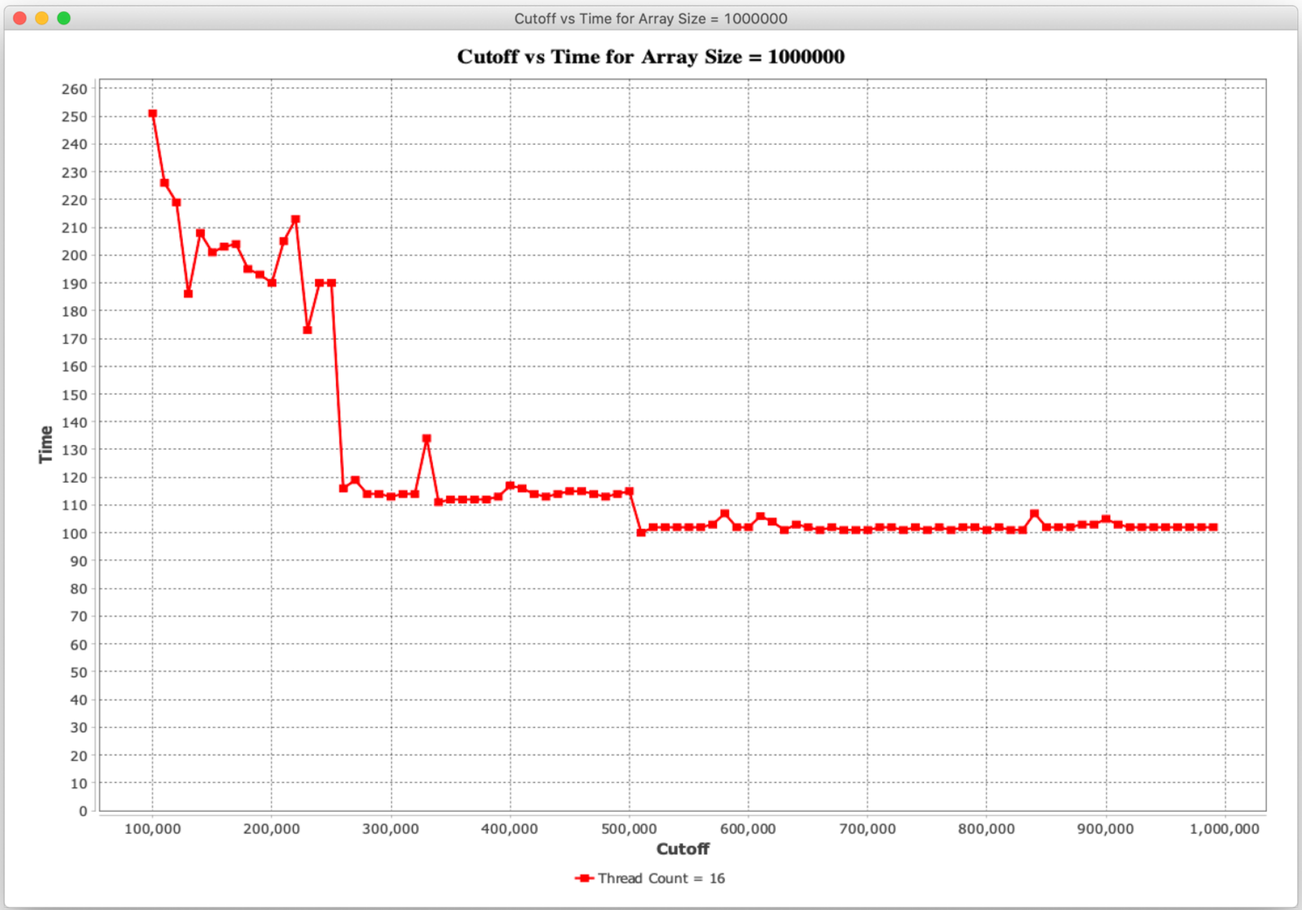

Figure 4.1 Graph recording time to do parallel sorting for cutoff range 100,000 – 1,000,000, fixed thread pool size = 16, and array size = 1,000,000 at 50 runs

From the graph above, we see that the time taken when the cutoff is 100,000 is very high, it slowly starts decreasing as our cutoff increases and more or less stabilizes to a consistent value from cutoff value = 500,000. However, in order to confirm if the time ever goes beyond our specific range, I decided to run the parallel sorting algorithm for a cutoff starting from 100, which as expected gave an out of memory error. So, I started the sort algorithm for a cutoff range: 1,000 to 591,000 with increments of 10,000 and plotted the results. I run the experiment for thread pool size = 1 and thread pool size = 16 as they gave the worst and best performance respectively.

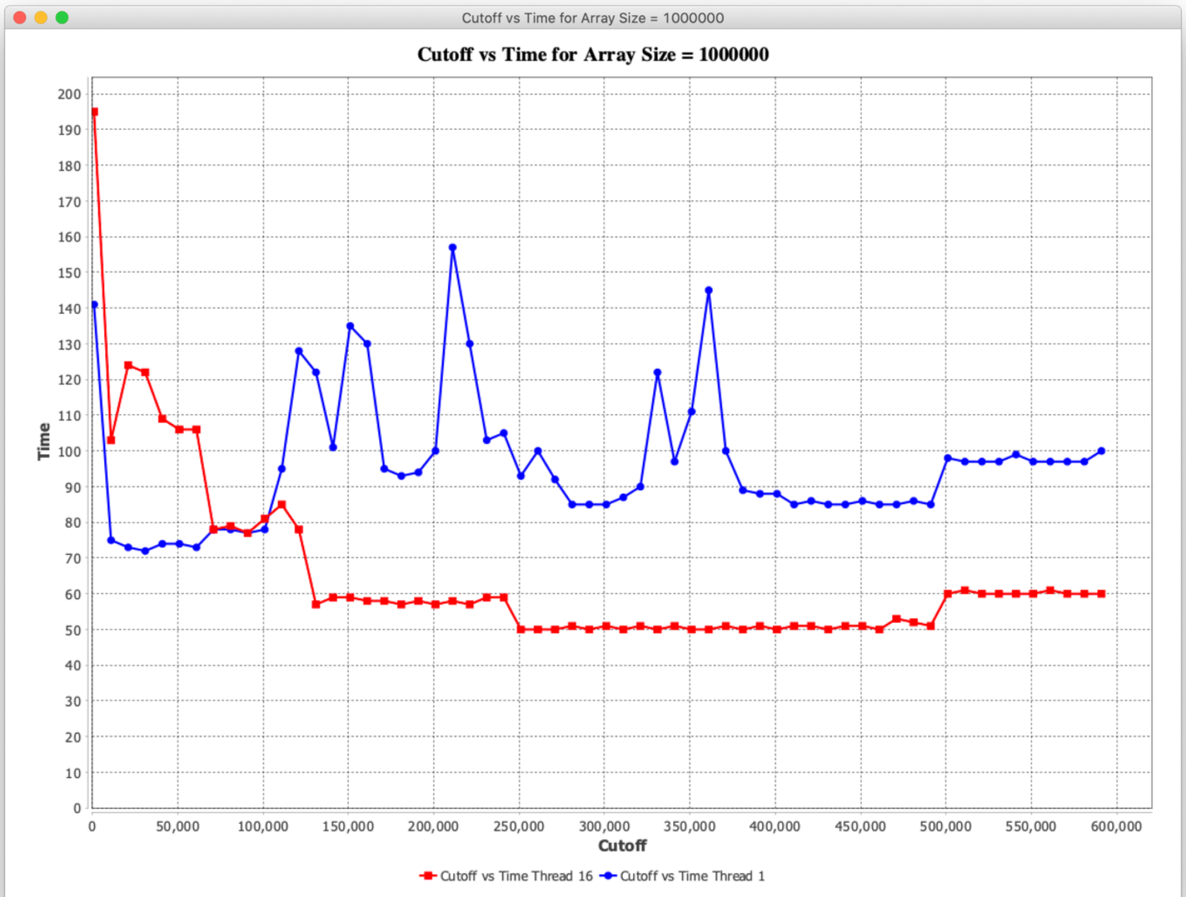

Figure 4.2 Graph comparing time to do parallel sorting for cutoff range 1000 – 591,000 and fixed thread size = 16, fixed thread size = 1, and array size = 1,000,000 at 50 runs

We can see that the graph is initially inverse of what we got for the range 500,000 to 1,000,000 where Thread Pool Size = 1 performed the poorest and Thread Pool Size = 16 performed the best. Instead, here we see that for thread pool size = 1, we get better results initially as opposed to thread pool size = 16. As we slowly start moving towards the cutoff range > 100,000, our results altogether start improving, following the improvement with increase in thread pool size trend. We can credit this decrease in performance with an imbalance in the cost of splitting work, queue and job handling and aggregating jobs results. There comes a point in the parallel algorithms where the overhead from waiting for threads to finish, creating and managing threads simply outweighs any performance improvements we achieve from concurrent computations. Hence, the cutoff value is extremely important from where the algorithm switches to sequential sorting. In fact, the time taken by system sort at array size = 1,000,000 was 105 ms which is lower than the time achieved by both thread pool = 1, thread pool = 16 at cutoff = 1,000.

Choosing cutoff = 1,000, running the parallel sorting algorithm for varying thread sizes and comparing it with the system sort we find results that confirm our hypothesis that it is significantly better to use normal sequential merge sort as the partition becomes smaller.

Array Size	Thread Pool Size = 1	Thread Pool Size = 2	Thread Pool Size = 4	Thread Pool Size = 8	Thread Pool Size = 16	System Sort
200,000	36 ms	42 ms	64 ms	48 ms	59 ms	21 ms
400,000	55 ms	62 ms	51 ms	68 ms	94 ms	40 ms
800,000	118 ms	126 ms	127 ms	127 ms	147 ms	83 ms
1,600,000	351 ms	446 ms	510 ms	511 ms	483 ms	194 ms
3,200,000	1464 ms	1654 ms	1151 ms	970 ms	984 ms	365 ms

Table 4.1: Results recording time taken by parallel sorting algorithm at various thread pool sizes at cutoff = 1,000 and the system sort algorithm

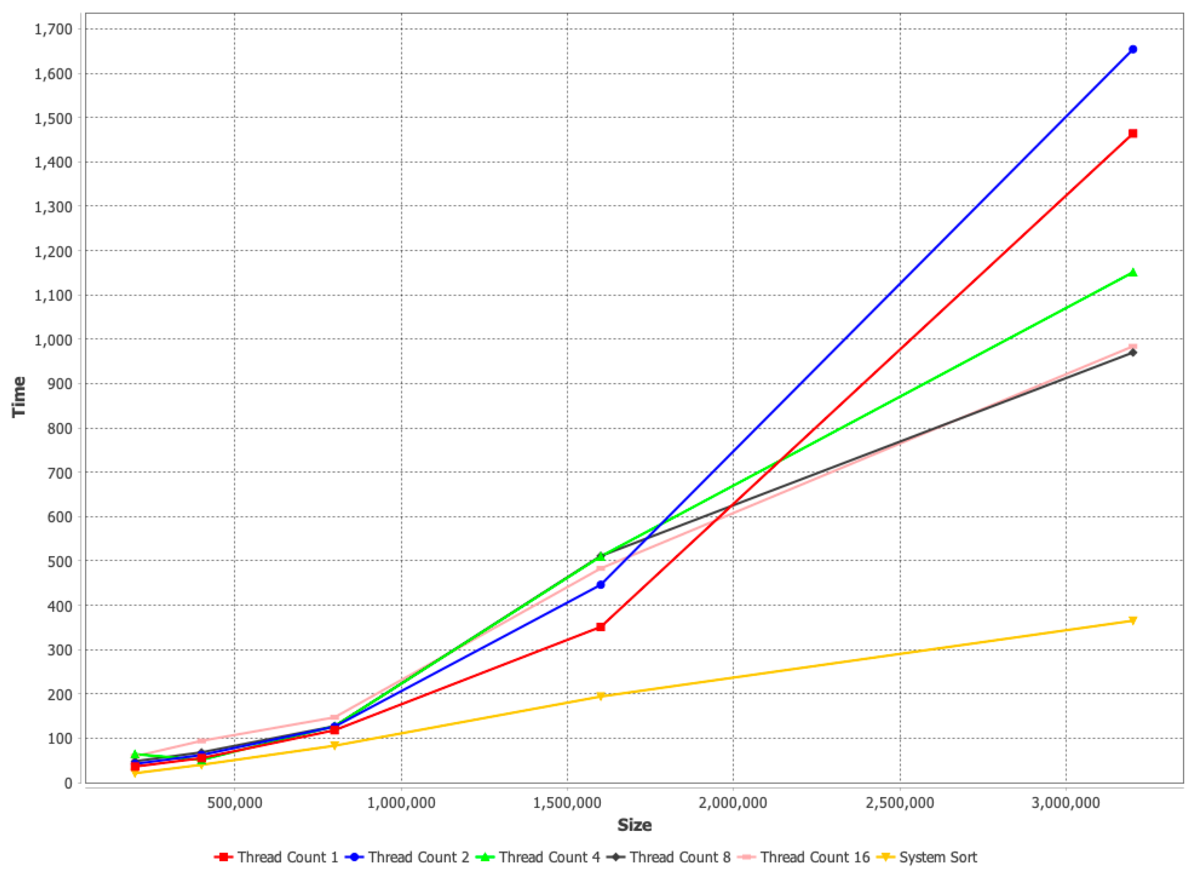

Figure 4.3: Graph plotting Size vs Time graph for parallel sorting algorithm with different thread pool sizes at cutoff = 1,000 and the system sort

Finally, choosing a good cutoff value = 600,000, I reported and plotted below the results of the performance of my parallel sorting algorithm with each thread pool sizes, system sort and array size ranging from 800,000 – 12,800,000.

Array Size	Thread Pool Size = 1	Thread Pool Size = 2	Thread Pool Size = 4	Thread Pool Size = 8	Thread Pool Size = 16	System Sort
800,000	102 ms	49 ms	49 ms	48 ms	49 ms	78 ms
1,600,000	168 ms	165 ms	111 ms	127 ms	90 ms	168 ms
3,200,000	296 ms	306 ms	290 ms	285 ms	213 ms	337 ms
6,400,000	668 ms	595 ms	606 ms	628 ms	563 ms	760 ms
12,800,000	1385 ms	1321 ms	1356 ms	1399 ms	1447 ms	1652 ms

Table 4.2: Results recording time taken by parallel sorting algorithm at various thread pool sizes at cutoff = 600,000 and the system sort algorithm

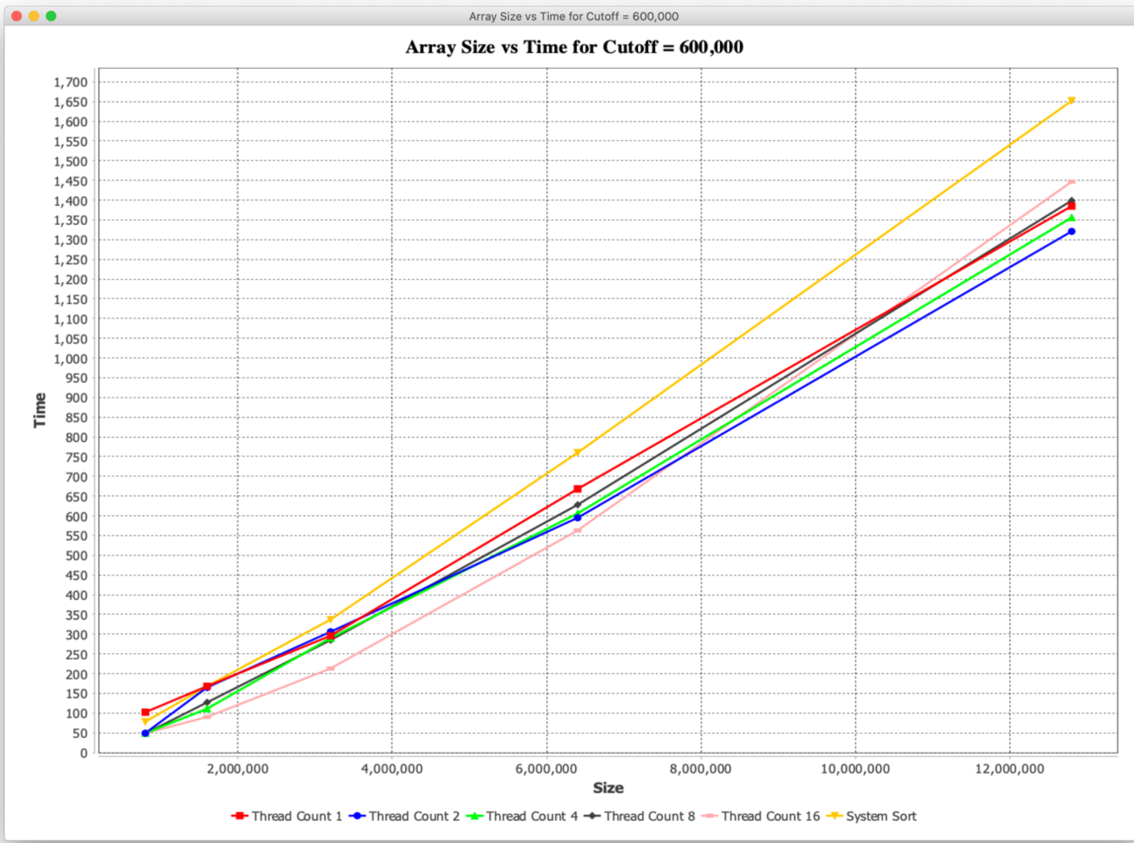

Figure 4.4: Graph plotting Size vs Time graph for parallel sorting algorithm with different thread pool sizes at cutoff = 600,000 and the system sort

Console Output:

```
For thread count: 1 and size: 800000 the time taken is: 102
```

```
For thread count: 2 and size: 800000 the time taken is: 49
For thread count: 4 and size: 800000 the time taken is: 49
For thread count: 8 and size: 800000 the time taken is: 48
For thread count: 16 and size: 800000 the time taken is: 49
Time taken by system sort is: 78
For thread count: 1 and size: 1600000 the time taken is: 168
For thread count: 2 and size: 1600000 the time taken is: 165
For thread count: 4 and size: 1600000 the time taken is: 111
For thread count: 8 and size: 1600000 the time taken is: 127
For thread count: 16 and size: 1600000 the time taken is: 90
Time taken by system sort is: 168
For thread count: 1 and size: 3200000 the time taken is: 296
For thread count: 2 and size: 3200000 the time taken is: 306
For thread count: 4 and size: 3200000 the time taken is: 290
For thread count: 8 and size: 3200000 the time taken is: 285
For thread count: 16 and size: 3200000 the time taken is: 213
Time taken by system sort is: 337
For thread count: 1 and size: 6400000 the time taken is: 668
For thread count: 2 and size: 6400000 the time taken is: 595
For thread count: 4 and size: 6400000 the time taken is: 606
For thread count: 8 and size: 6400000 the time taken is: 628
For thread count: 16 and size: 6400000 the time taken is: 563
Time taken by system sort is: 760
For thread count: 1 and size: 12800000 the time taken is: 1385
For thread count: 2 and size: 12800000 the time taken is: 1321
For thread count: 4 and size: 12800000 the time taken is: 1356
For thread count: 8 and size: 12800000 the time taken is: 1399
For thread count: 16 and size: 12800000 the time taken is: 1447
Time taken by system sort is: 1652
```

From the above results, we can see that our parallel algorithm almost follows the same linearithmic pattern as the system sort but the constant value changes according to the thread pool count. We can see that for a good cutoff value, we can benefit from using a parallel sorting algorithm with a bigger thread pool size.

- **Conclusions**

Summarizing our observations from our experiments, we find that:

1. This experiment will differ majorly on different systems and may produce different results.
2. When the cutoff value lies between 100,000 – 1,000,000, our parallel sorting algorithm performs the best with a peak performance between 500,000 – 700,000 which may vary given different conditions. (Fig. 4.1)
3. In the cutoff values between 500,000 – 1,000,000, our parallel sorting algorithm performs the best when thread pool size = 16, and the worst when thread pool size = 1. (Table 3.1 – 3.3, Fig 3.1 – 3.3).
4. When the cutoff size is too small (most of our work is being done in parallel), we face the consequences of high overhead of creating, managing and synchronization and the parallel sorting algorithm becomes an overkill. The cost of these outweighs any improvement in performance, hence, we must be careful in the cutoff value we choose. (Fig 4.2, Fig 4.3)
5. When cutoff size is too small, sequential system sort (normal merge sort) performs better. (Fig 4.3)
6. The parallel sorting algorithm has an approximately linearithmic performance similar to the system sort algorithm, however, the constant changes with respect to the thread pool size being used. (Fig 4.4, Table 4.2).
7. For parallel programming, we must always perform experiments to provide a reasonable cutoff because there comes a point where the cost of creating, managing and synchronizing threads exceeds any performance improvement we could've gotten.