Introduction

The project aimed to investigate and analyze the implications of multithreaded sorting of arrays using the Java ForkJoinPool framework. The analysis was done by comparing the performance of the multithreaded sorting, against the Java's native sequential Arrays.sort().
In this task, two divide-and-conquer sorting algorithms were used and compared: Mergesort and Quicksort. A short description of the two algorithms follows:

Mergesort:
1. Divide the input array into two.
2. Recursively sort the left and right half of the input
3. By recursively moving the smallest elements, merge the sorted results into a new sorted array.

Quicksort:
1. Pick a pivot element
2. Split the input data into three groups: elements less than the pivot, the pivot itself, elements greater than the pivot.
3. Recursively sort each group.

The sequential mergesort and quicksort both have an average time-complexity of O(nlogn). The intended multithreading approach would result in an expected parallelism of O(logn).

Method

Approaching the solution

The solution was approached by first figuring out the structure of the parallelism to be implemented for the two sorting algorithms. Due to the divide-and-conquer strategy of both, step 3 in the algorithm above was modified to do the recursive sorting in parallel. Each thread creates a helper thread, splitting the work between the threads.

The following shows the parallel recursive step as implemented in the merge sort.

```
int mid = start + (end-start) / 2;
MergesortParallel left = new MergesortParallel(arr, start, mid, threshold);
MergesortParallel right = new MergesortParallel(arr, mid, end, threshold);
left.fork();
right.compute();
left.join();

sequentialMerge(mid);
```

It can be seen that the partition point was selected to be the middle of the array created by each thread. The sequentialMerge() method is called on each thread to perform the final merging step. It is worth noting that initially, I used the in-built Java method, invokeall() to start the threads, divide, and wait for all threads to finish executing. However, using the approach shown in the code above ensured that the main thread would work with the helper thread, and not just create helper threads and wait for them to execute. This proved to optimize performance by a significant factor.

To prevent too many threads being created, which are unnecessary for a relatively small array size, a sequential cut-off was used.. This is the point at which the process would switch from parallel to sequential. For this project, the sequential cut-off point was decided by the number of threads, where cut-off = (array size)/(number of threads).

Quicksort was implemented in a similar manner, with an exception of the partition point (pivot) which was determined by a separate method called partition(). This method would pick the median element, and move all the elements greater and less than this point to the left and right respectively.

Junit tests (passed) were carried out against the native Arrays.sort() to validate the accuracy of the parallel sorting.

Analysis

The actual timing and analysis was done by the Driversort class. A couple of steps were implemented in the code to ensure an efficient running and comparison of the two sorting methods.

Firstly, a main array of size <ArrayMaxSize> was created and populated with random integers to hold the maximum number of elements needed.
Both the sequential and parallel sorts do 5 timed runs for array sizes as specified by the user, with the number of threads for each run increasing in powers of two from 2 to 1024 (achieved using three for loops).

For consistency, the same array of random integers is sorted at each run, with separate copies being created at the specified array size.
The timing was done using System.nanoTime() and was started just before the execution of the sorting line of code, and stopped right after that. The difference in the two times is then inserted into an array list for both sequential and parallel runs, making it possible to select the best run times using the Collections.min() method.

From the two array lists holding the sequential and parallel run times, the best is selected for both. Consequently, the speed up is calculated as follows:
Speed up = (Best sequential run time) / (Best parallel run time).
The maximum speed up is selected and displayed as output.

Tested machine architectures

Machine 1:

- CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
- CPU cores: 2
- Number of processors: 4
- Cache size: 3072 KB

Machine 2 (Nightmare):

- CPU: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- CPU cores: 4
- Number of processors: 8
- Cache size: 12288 KB

Interesting encounters/problems

An interesting problem I encountered was that of multithreaded sorting of an already-sorted array at each run. Initially, I was not copying the elements from the main array containing random integers, into the array to be sorted at each run. In other words, the elements were not being copied within the for loop. This resulted in longer run times for the parallel sorts as compared to the sequential ones, leading to no speedups.

Correct code:

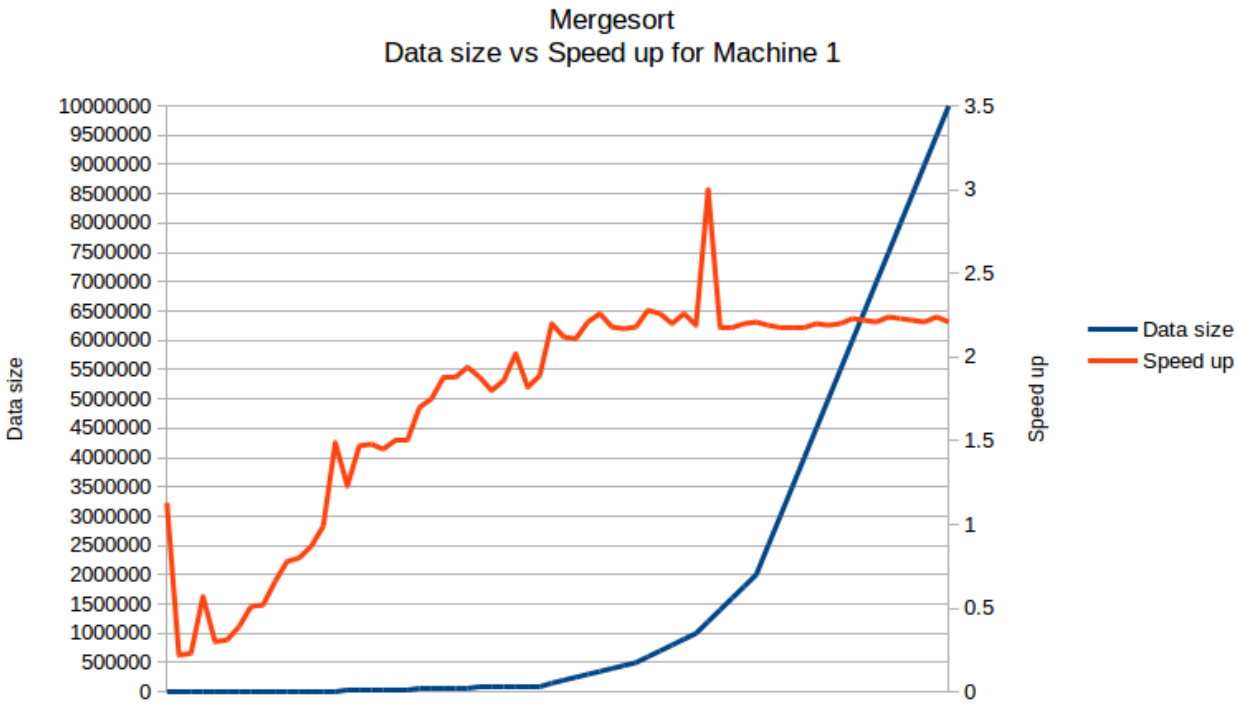
```
for(int i = arraySizeMin; i<=arraySizeMax; i+=arraySizeIncr){
        ……
        for(int t=2; t<=1024; t*=2){
        ……
                for(int j = 0; j<5; j++){
                 System.arraycopy(array, 0, copy, 0, i);  //missing line
```

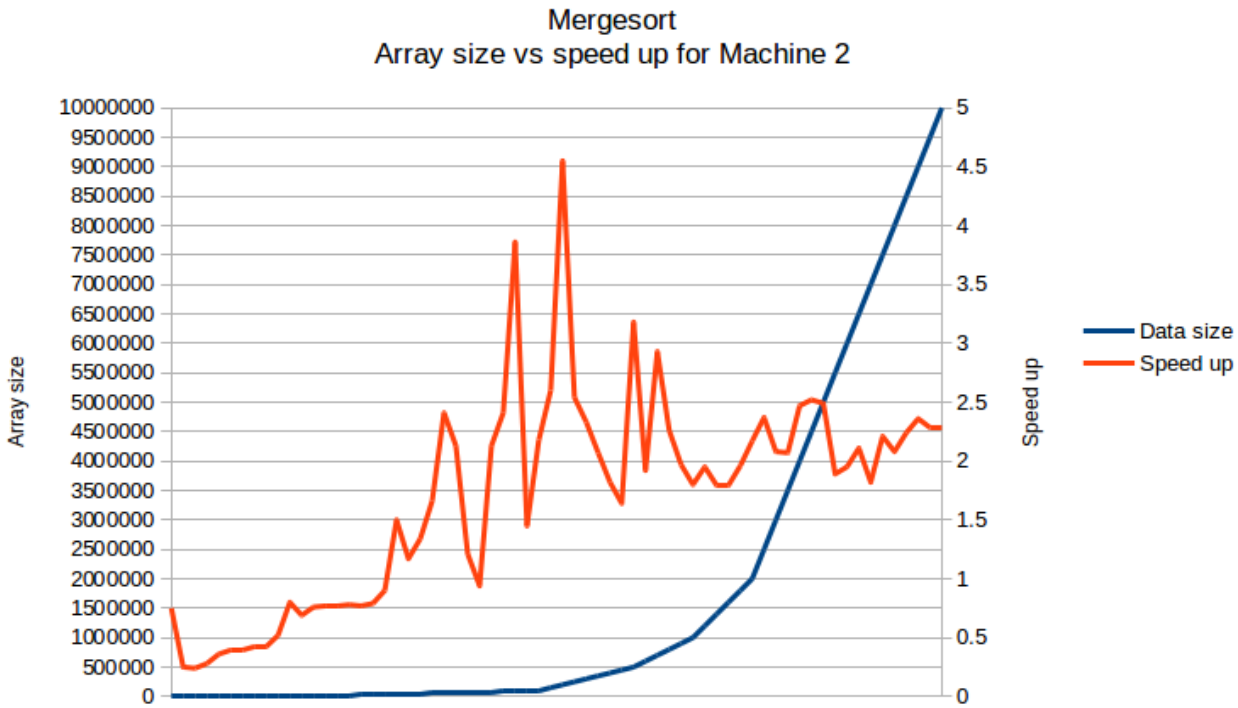
Secondly, since each run was conducted on number of threads increasing from 2 to 1024 in powers of 2, the creation of too many threads caused an error while sorting array sizes less than 3000. This was a thought provoking situation, which was fixed by inserting an if statement to limit the number of threads for array sizes less than 3000.

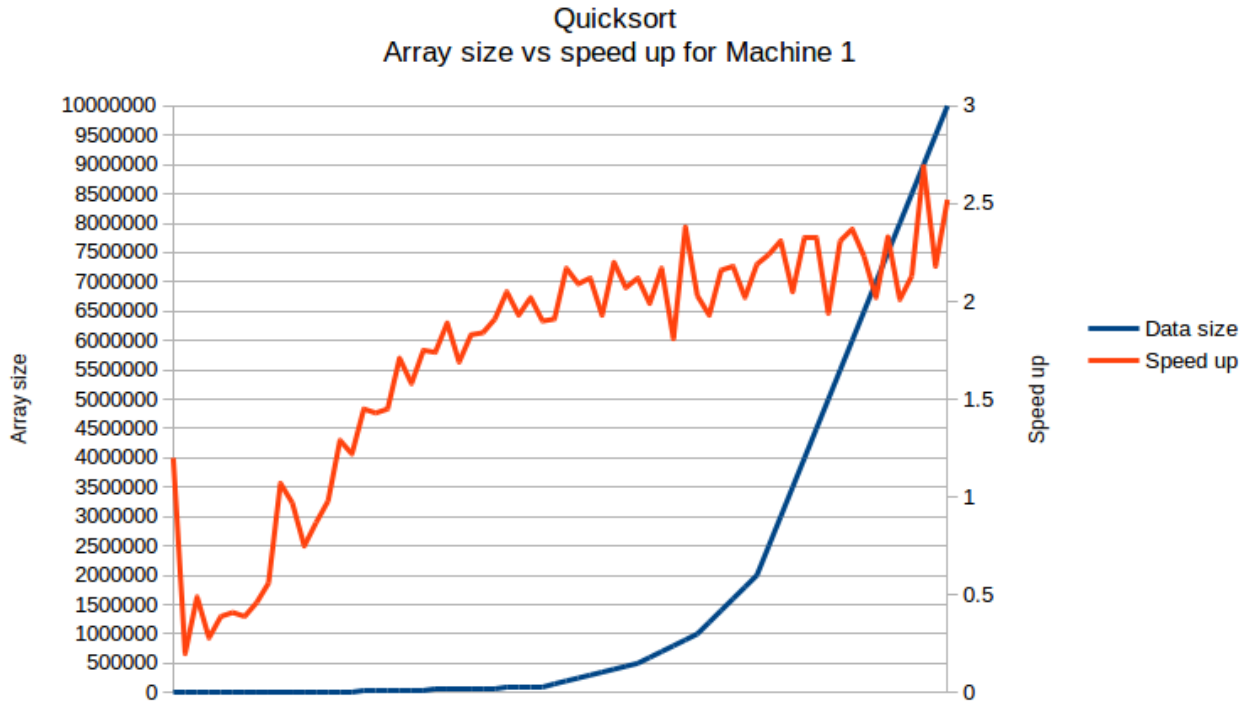
Results and Discussion

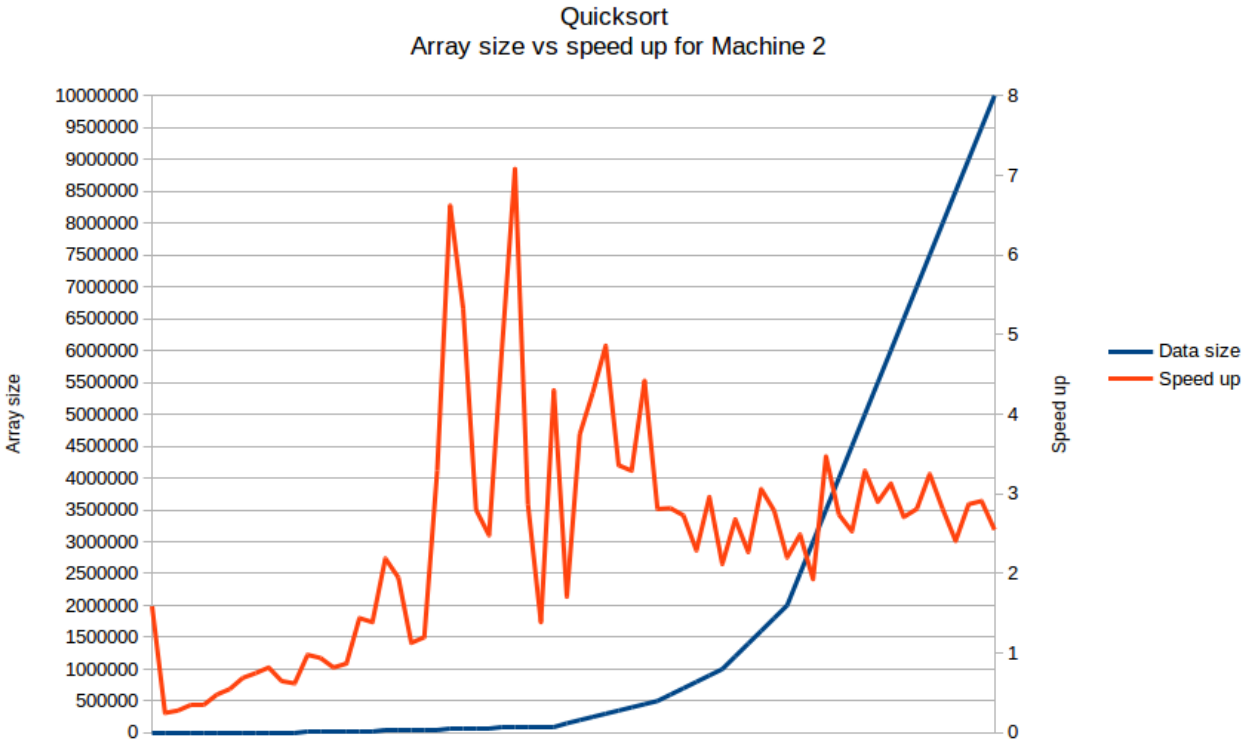
The two sorts were tested and compared with array sizes ranging from 1000 to 10000000. The graphs on the following pages display the results.

It is evident from the general pattern that the speedup increases with the increase in array sizes. Although not clearly seen on the graphs due to the size limitations, the tests showed that speedups were > 1 for array sizes >= 10000 for Machine 1 and array sizes > 20000 for Machine 2. This was the case for both mergesort and quicksort. Therefore, multithreading would only be feasible for these array sizes, below which the sequential sorts would be faster.


Mergesort
Data size vs Speed up for Machine 1
Data size
Speed up
Data size
Speed up
10000000
9500000
9000000
8500000
8000000
7500000
7000000
6500000
6000000
5500000
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
3.5
3
2.5
2
1.5
1
0.5
0


Mergesort
Array size vs speed up for Machine 2
Array size
Speed up
Data size
Speed up
10000000
9500000
9000000
8500000
8000000
7500000
7000000
6500000
6000000
5500000
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0


Quicksort
Array size vs speed up for Machine 1
Array size
Speed up
Data size
Speed up
10000000
9500000
9000000
8500000
8000000
7500000
7000000
6500000
6000000
5500000
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
3
2.5
2
1.5
1
0.5
0


Quicksort
Array size vs speed up for Machine 2
Array size
Speed up
Data size
Speed up
10000000
9500000
9000000
8500000
8000000
7500000
7000000
6500000
6000000
5500000
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
8
7
6
5
4
3
2
1
0

On Machine 1, the parallel mergesort was seen to obtain fairly constant speedups of 2.1-2.4 for array sizes greater than 100,000. One anomaly was a speedup of 3.0 for array size 1200000, but this could be caused by unstable processes. On Machine 2, the speedups for mergesort were more significant, ranging from 2 to 4. Array sizes between 100,000 and 500,000 seemed to perform best on both the architectures, although not too significant.

Quicksort outperformed mergesort in both architectures, but performed the best in Machine 2 by attaining a maximum speedup of 7.08. This did not seem authentic on the first run, but similar results were obtained in the second one. For array sizes 55000 – 450000 and 200000 – 5000000, quicksort attained significantly good speedups.

The maximum speedup for the parallel implementation of mergesort was 3.86, obtained when run on Machine 2 with 4 CPU cores. Multithreaded quicksort obtained a significantly higher maximum speed up of 7.08 on the same Machine. Consequently, this provides evidence that the parallel sorts performed better on the architecture with a higher number of cores and processors.
Using Amdahl's law, the ideal parallel processing case is limited to the number of added processors. Parallel mergesort did not seem to get close to the ideal case of 8 on Machine 2, as compared to parallel quicksort which obtained a maximum speed up of 7.08.

The results of the optimal number of threads depended not only on the architecture, but also the type of parallel sorting. Mergesort showed to have an average optimal number of threads equal to the number of processors in each architecture. In Machine 1, the optimal number of threads was 2 for array sizes < 50000, but held a constant of 4 threads for greater array sizes. Machine 2 had a few scattered optimal threads, but the average was still equal to the number of processors, 8. Interestingly, multithreaded quicksort required a greater number of threads of between 8 and 1024 (scattered) for optimum results in both the architectures.

Conclusions

Clearly, the architecture with the higher number of cores performed better than the other. More interesting would have been to test it on other architectures with even greater number of cores, and validating the theorem of diminishing returns with added processors (speedup per core decreases as the number of cores increases). This is one of the positive outcomes of this project, which instilled in me the interest for further investigation of multithreading. It gave me an opportunity to witness the speedups practically through timing experiments.

The parallel sorting approach proved to obtain average speedups mostly in the range of 2 and 4, but a higher maximum. In this experiment, the only aspect parallelized was the recursive sorting using divide-and-conquer mechanism which was not too expensive to implement, apart from the debugging difficulties. In my opinion, the algorithm performance was definitely optimized, although not too significantly, and was worth the parallelization.