PARALLEL PROGRAMMING

CSC2002S

AUGUST 1, 2016

STEFFI BAUMGART

BMGSTE001

Introduction

The aim of the assignment is to implement two sorting algorithms using the Java Fork/Join Framework and to find out the optimal number of threads for different computer architectures that yield fastest run-time. These algorithms, namely Mergesort and Quicksort, need to be both correct and faster than their sequential counterparts. Other than computer architecture, the factors that affect this include the CPU speed, memory load versus available RAM, available cache size, hard drive speed etc. The choice of programming language impacts the time taken to process as well, but since this assignment was required to be written in Java, no such comparisons can be made in this area (although it is to be expected that Java has a relatively slow run time due to its high-level nature.) It is expected that the speedup gained from parallelizing both algorithms would be enough to warrant the programming effort and time into writing them. Even though they have the same time complexity of $O(n \log n)$ for an array of n items, Quicksort is expected to be slightly faster (almost 2-3 times as fast) if implemented efficiently.

Methods

1. Description of the approach to the solution

The template for Mergesort, which the lecturer Michelle Kuttel provided, was used to structure the MergeSort algorithm. This same structure was applied to Quicksort, where only the sequential method needed to be implemented. Since both sorting methods involve the Divide and Conquer paradigm, recursion could be applied to both by extending the RecursiveAction class (RecursiveAction extends java.util.concurrent.ForkJoinTask and is invoked by ForkJoinPool.) A third class, called DriverSort, was created to take command-line arguments, apply these parameters to the sorting method chosen, calculate the time taken and therefore speedup of the parallel code in comparison to the serial version, and write the results to a file. This information was then used to create data visualizations of the results, in order to gain further insight into the benefits of multi-threading.

2. Algorithm Validation

Java's Regression Testing Framework was used to write Junit tests called TestMergesortParallel and TestQuicksortParallel. The purpose of this was to confirm the correctness of the two algorithms implemented, but didn't specifically test each method written. Trace statements were also inserted at various stages of each class, to check if the integers were actually being sorted in the ascending numerical order.

3. Timing the algorithms with different input

Both sorting algorithms were run with 3 different sets of parameters on 3 different computer architectures. These parameters were the:

I. sorting algorithm chosen (e.g. "Mergesort" or "Quicksort")
II. the minimum size of the array (e.g. 10 000)
III. the maximum size of the array (e.g. 100 000)
IV. the step size for increasing the array size (e.g. 1000)
V. the name of the file which the results would be written to (e.g. "dataFile")

The first test had a minimum array size of ten thousand, and increased in increments of a ten thousand until it reached a hundred thousand.(java DriverSort mergesort 10000 100000 10000 file in the bin folder.) The next test looped from a array size of hundred thousand to a million with a step size of ten thousand (java DriverSort mergesort 100000 1000000 100000 file.) Lastly, the arguments given for 3rd trial were mergesort 1000000 10000000 1000000 file (Increasing in one million increments from one million to ten million.) This ensured that the algorithms were tested in a very wide range of array sizes, all the way from 1000 to 10 000 000.

4. <u>Measuring speedup</u>

Speedup was measured as the time that the Arrays.sort() function took to sequentially sort the array (in nano-seconds), divided by the time that the parallelized code took (in nanoseconds.) This is represented as $Speedup = \frac{Sequential\ Time}{Paralell\ Time}$ and shows how much faster the program runs on multiple threads, compared to the sequential approach of the builtin Java method. Latency was recorded by evaluating the difference in system time exactly before and after the sort was called (the garbage collector was also called before the sorting invocation, so as to ensure that the time recorded would be as accurate, and as small as possible.)

5. <u>Machine architectures that the code was tested on</u>

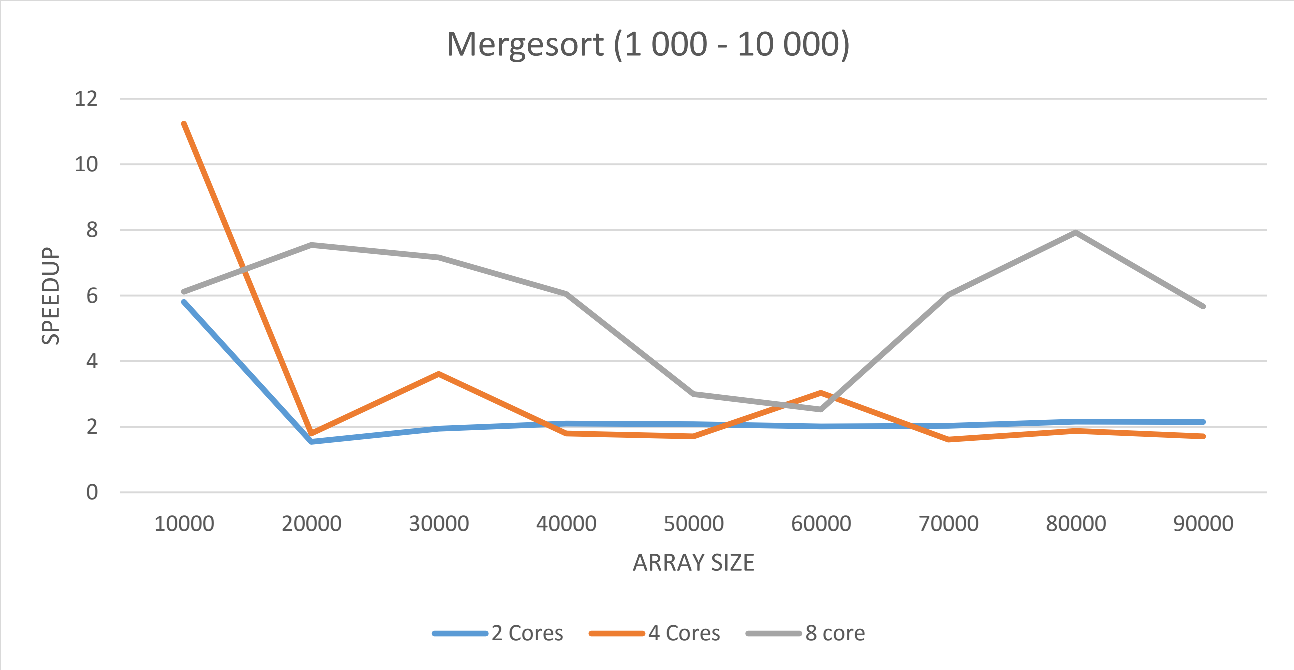
The code was tested on the computers in the Senior Lab at UCT, which had 4 cores. It was also tested on Oracle's *VM VirtualBox* which is a virtual machine that utilized 2 cores. Lastly, the code was executed on 8 cores of the computer science Department's file server *Nightmare.*

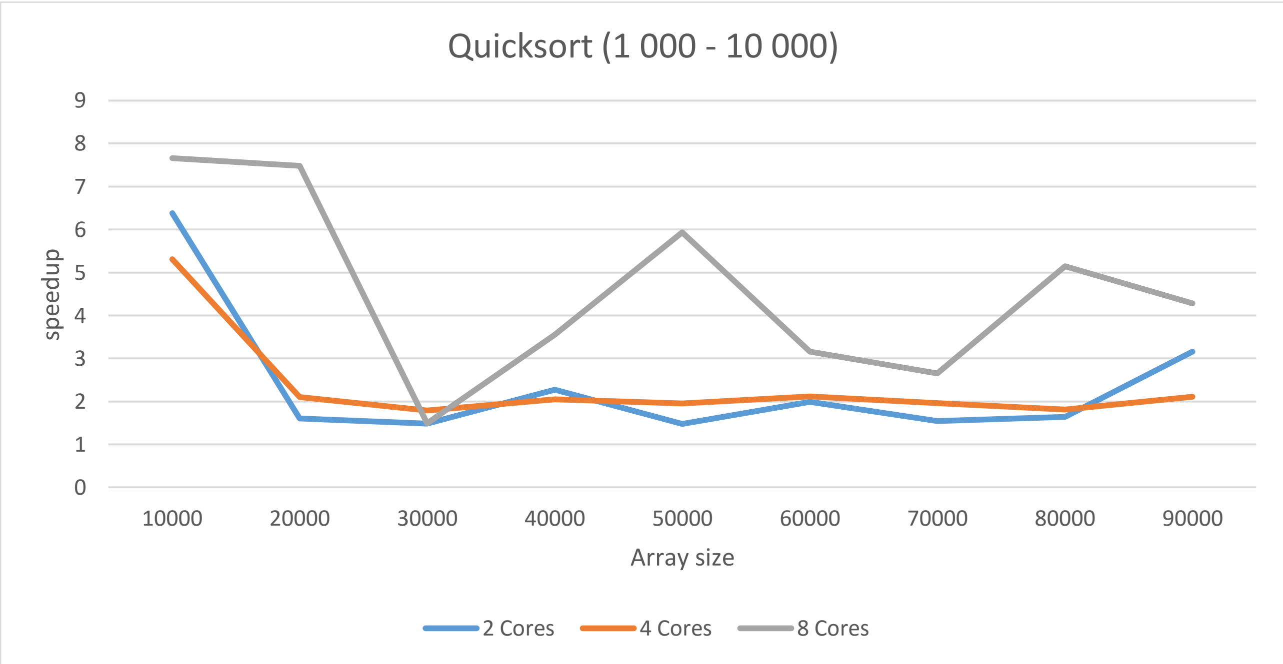
6. <u>Difficulties Encountered</u>

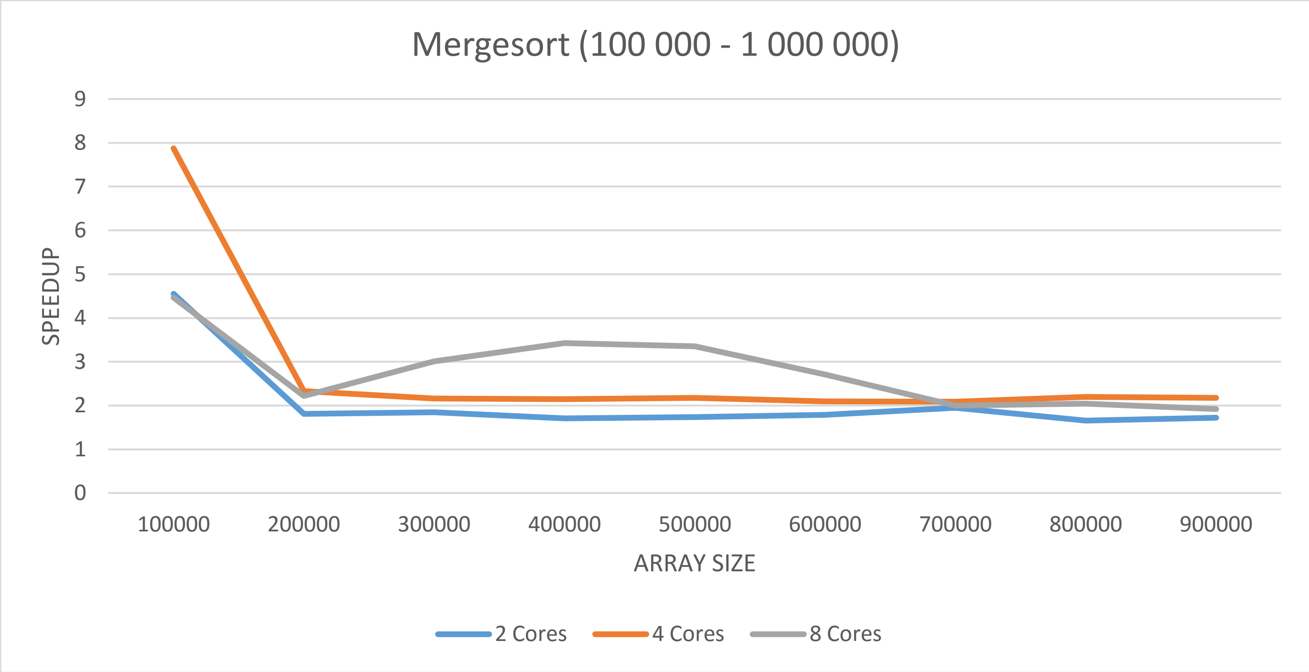
The initial challenge was conceptually understanding the process of multi-threading, as up until this point, all our problems were solved with serial computation, and trying to parallelize code is a harder to both comprehend and plan out. Mergesort proved easier to write, as the pivot chosen was the middle data item, whereas the last element was used in Quicksort. The hardest of all classes was DriverSort, as this class needed to be the driver for the program; taking the command-line arguments and printing to the file were the only parts that were straight-forward. I struggled with the ordering and nesting of the 5 for loops I used, as I started programming immediately and didn't try to first design the logical flow of the class. This also included more debugging than the sorting algorithms, as it was handling the other 2 classes, and produced no results of its 'own.' Trouble-shooting Mergesort and Quicksort mostly involved inserting multiple trace statements to see the stage of the array at different times, while this could not be done for DriverSort.class. I also made my step sizes too small initially, which resulted in almost a hundred data items per test (thus making graphs quite tedious and difficult to analyze.)

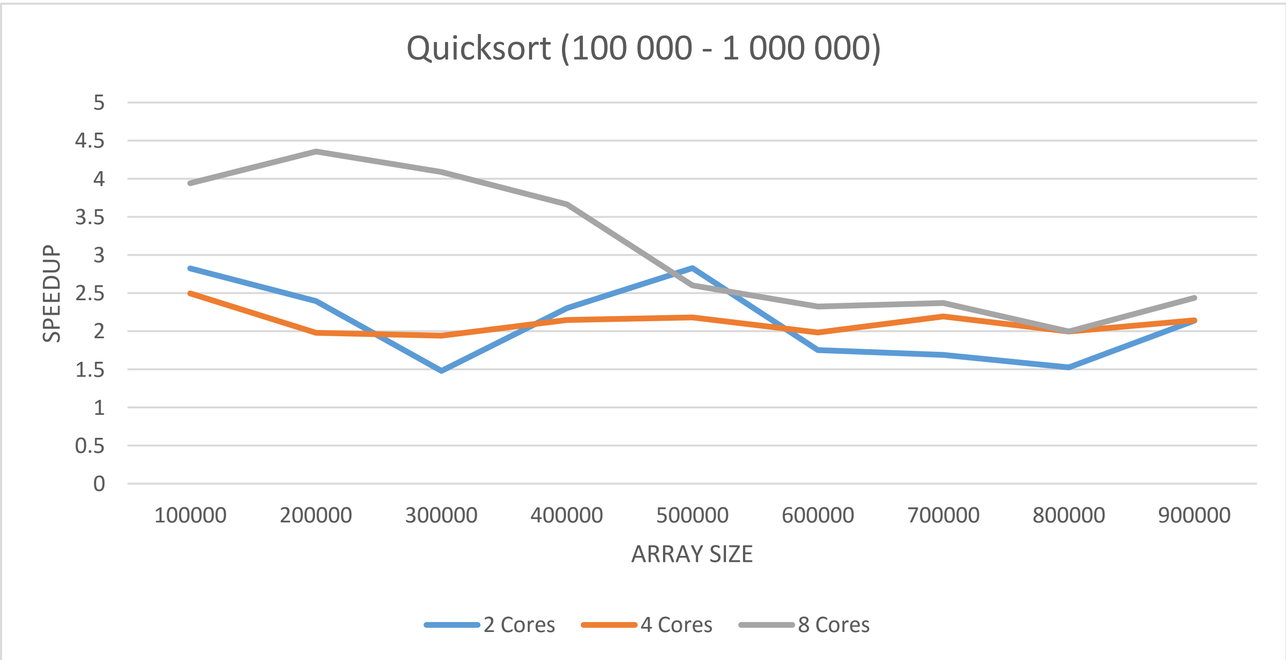
Results

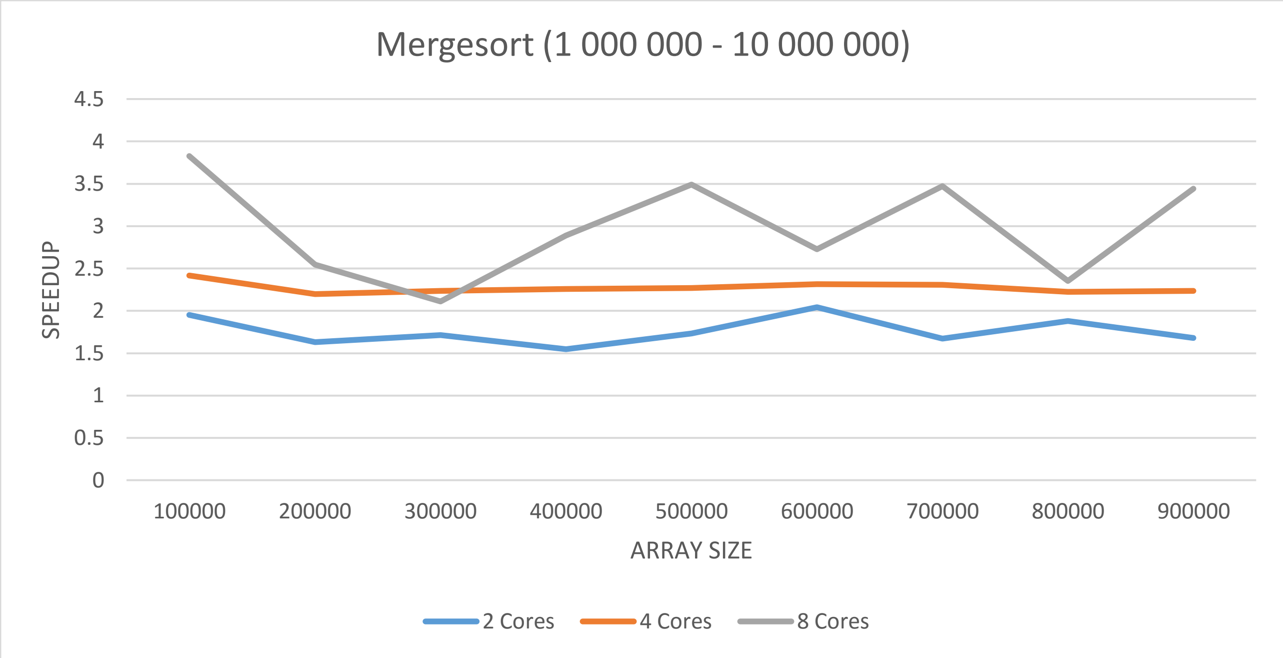
<u>Array Size VS Speedup Results:</u> The following 6 line graphs depict the two sorting algorithms being run on multi-core machines with different array sizes. It should be noted that a thread count of 1 was incorrectly added, thus the first value of the Array Size (x axis) should be ignored. It should also be observed that the some instances of the 2 core graph "surpass" Super Linear Speedup, which is impossible as 2 cores are not able to improve the speed of serial code through parallelisation by more than a factor of 2. This anomaly of unusual speedup fluctuations can be attributed to the fact that the Virtual Machine may use additional processing power of the operating systems it runs on top off.


Mergesort (1 000 - 10 000)
12
10
8
6
4
2
0
SPEEDUP
10000
20000
30000
40000
50000
60000
70000
80000
90000
ARRAY SIZE
2 Cores
4 Cores
8 core

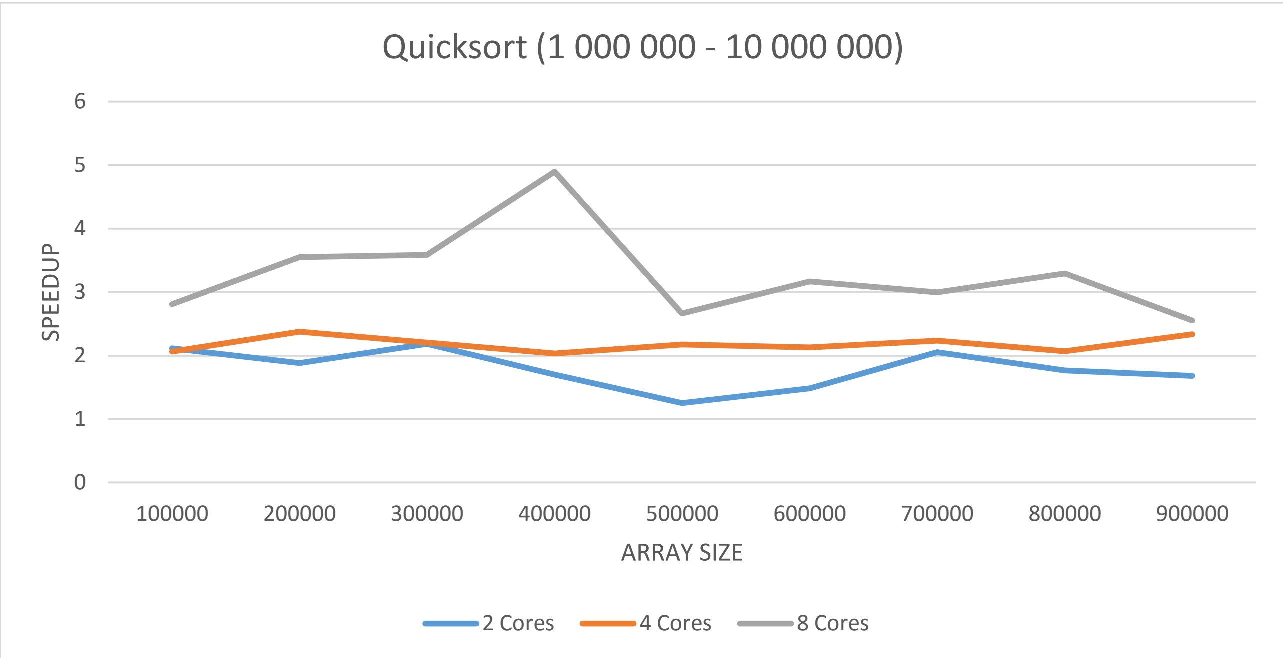

Quicksort (1 000 - 10 000)
9
8
7
6
5
4
3
2
1
0
speedup
10000
20000
30000
40000
50000
60000
70000
80000
90000
Array size
2 Cores
4 Cores
8 Cores


Mergesort (100 000 - 1 000 000)
SPEEDUP
9
8
7
6
5
4
3
2
1
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
ARRAY SIZE
2 Cores
4 Cores
8 Cores


Quicksort (100 000 - 1 000 000)
SPEEDUP
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
ARRAY SIZE
2 Cores
4 Cores
8 Cores

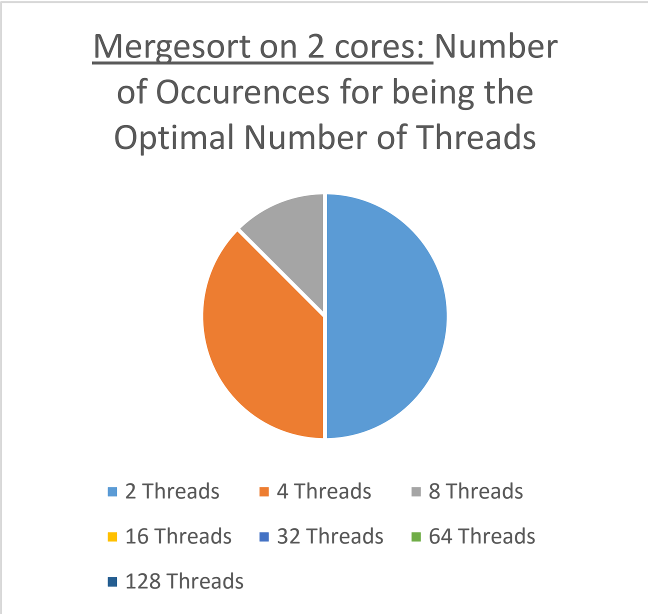


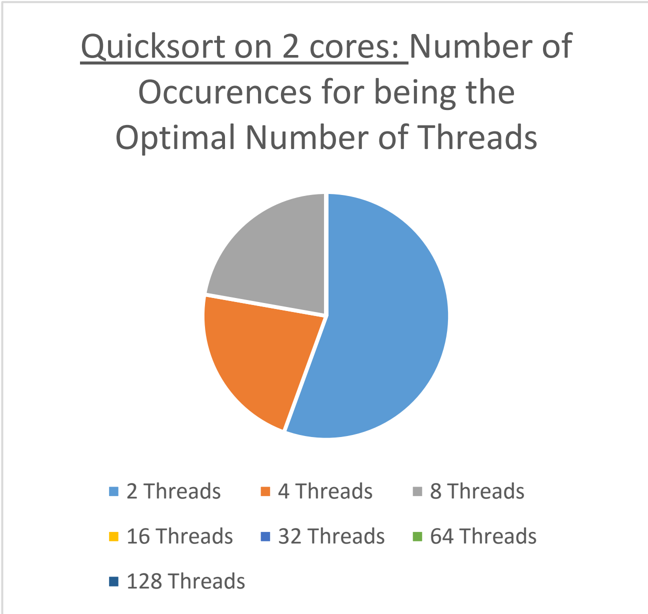
Mergesort (1 000 000 - 10 000 000)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
SPEEDUP
100000
200000
300000
400000
500000
600000
700000
800000
900000
ARRAY SIZE
2 Cores
4 Cores
8 Cores

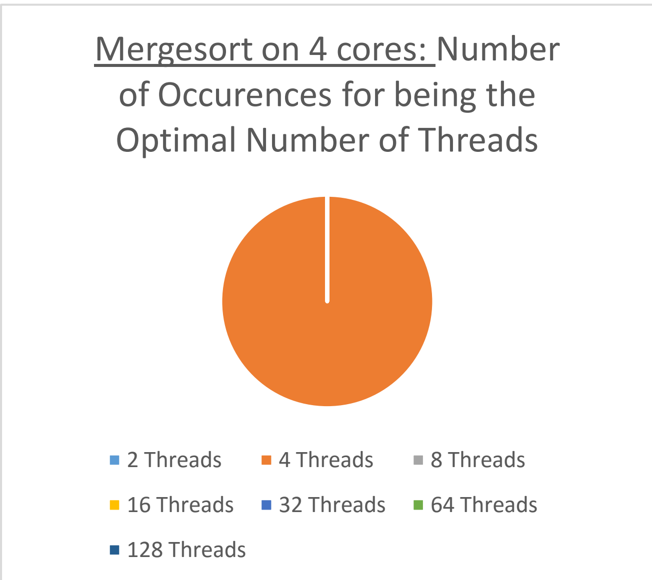


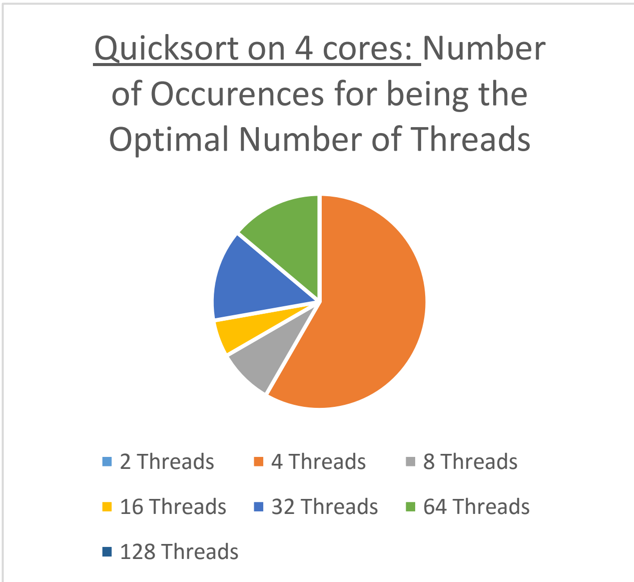
Quicksort (1 000 000 - 10 000 000)
6
5
4
3
2
1
0
SPEEDUP
100000
200000
300000
400000
500000
600000
700000
800000
900000
ARRAY SIZE
2 Cores
4 Cores
8 Cores

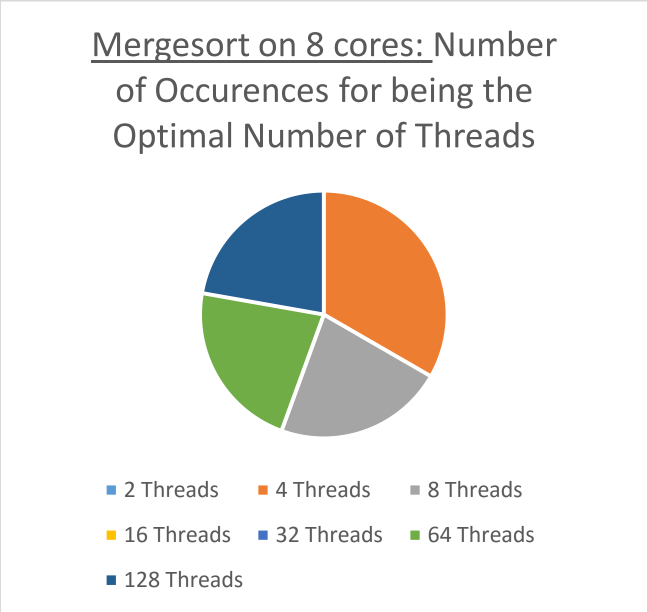
Optimal Number of Thread Results: The following 6 Pie Charts depict the number of occurrences a specific thread count was the optimal number of threads for a particular algorithm on a particular machine. It is important to note that only thread counts that occurred *more than once* for a particular range of array sizes were recorded to improve the readability of the visualizations.

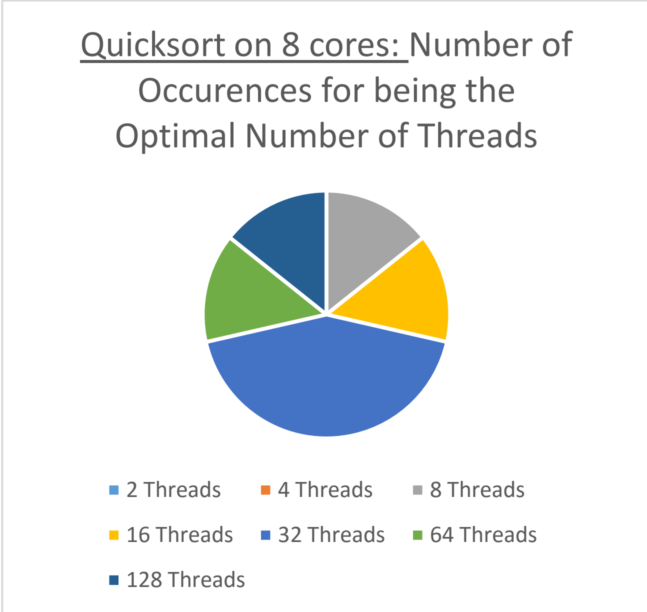












Discussion

1. <u>For what range of data set sizes does each parallel sort perform well?</u>
The parallelized sorts perform better for larger array sizes (testing from a million to ten million consistently yields greater speedups than the smaller array sizes.)

2. <u>What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?</u>
The best speedup result was 7.871281 for Mergesort (array range: 100000-1000000) for the 8 core Nightmare server. This is very close to the 8 speedup ideal (Super-linear speedup.)

3. <u>How do the parallel sorting algorithms compare with each other?</u>
In most cases Quicksort slightly outperforms Mergesort, but they mostly provide similar speedups. However, Mergesort has a more consistent performance whereas Quicksort has proven to have more speedup spikes and dips.

4. <u>How do different architectures influence speedup?</u>
Speedup and the number of cores used have a directly proportional relationship, meaning that in general it can be said that the more processors one uses, the higher the speedup is (but this limited by Amdahl's Law.) The 8 core Nightmare server proved faster than the 4 core computer, which was faster than the 2 core virtual machine.

5. <u>What is an optimal number of threads on each architecture?</u>
2 threads and 4 threads are the optimal number of threads for a 2 core machine. 4 Threads is best thread count for 4 core architecture, and the 8 core results provided show that a wide variety of thread numbers gave the best speedup for different array sizes (most notable are 4,8, and 32 threads.) It can also be observed that dividing tasks among 4 threads is the most beneficial for Mergesort in general.

6. Is it worth using parallelization (multithreading) to tackle this problem in Java?
From the results it can be concluded that the speedup gained from parallelizing code is worth investing time into, as parallelized code appears to run anywhere from 2-7 times faster depending on what computer architecture is used.

Conclusions

The hypothesis stated in the Introduction is correct, as both sorting algorithms produce good results, but Quicksort is slightly faster.

In most cases, the more cores the computer has, the greater the optimal thread count for speeding up computation (thus speeding up processing time.) However, according to Amdahl's Law, as the number of process approach infinity, the theoretical speedup becomes limited by the part of the task that cannot benefit from the improvement (i.e. the sequential code.)

Also, the analysis drawn from the 2 core virtual machine is not the most reliable, owing to the false fluctuations of speedup.

Appendix

My Git repository is divided into 2 folders, namely Git Repo 1 and Git Repo 2. The reason for this is that I was working on my laptop, and halfway through I transferred to one of the desktops at UCT. After many unsuccessful attempts at trying to merge the two repositories, I decided to leave them as separate.

Result Data

java DriverSort mergesort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	94.9115	5.809575
20000	32	77.5747	1.5416104
30000	4	109.8595	1.9480631
40000	4	138.4058	2.0977044
50000	4	190.493	2.0768564
60000	4	230.8801	2.0144672
70000	8	253.4009	2.0349195
80000	4	275.8186	2.1622922
90000	4	334.6996	2.1492262

java DriverSort quicksort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	195.8208	6.383065
20000	2	221.9303	1.6012726
30000	8	259.2773	1.4823893
40000	2	527.2305	2.276527
50000	16	440.9791	1.4787725
60000	32	588.4577	1.98854

70000	2	662.2727	1.5407866
80000	8	703.358	1.641018
90000	1	842.5486	3.1556385

java DriverSort mergesort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	1	2167.7483	4.556106
200000	4	2533.5662	1.8091528
300000	8	4223.5586	1.8474808
400000	2	5989.609	1.7102461
500000	1	7262.337	1.7396632
600000	32	8210.237	1.7877481
700000	4	10093.033	1.9454727
800000	2	10619.112	1.6534718
900000	2	9128.498	1.7200438

java DriverSort quicksort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	2	1246.5427	2.8244205
200000	1	1199.704	2.3942769
300000	2	2684.316	1.4819477
400000	1	4514.8306	2.3035235
500000	256	3804.7776	2.8266835
600000	4	4099.269	1.7550794
700000	4	5346.359	1.6904826
800000	32	7005.108	1.5244137
900000	128	6245.1494	2.1391485

java DriverSort mergesort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	1	11479.457	1.9519835
2000000	2	21489.346	1.6307894
3000000	4	31496.281	1.7156831
4000000	4	46287.844	1.5482295
5000000	4	57044.68	1.7343128
6000000	2	63211.832	2.0444021
7000000	256	83737.85	1.6728152
8000000	4	85736.27	1.8788611
9000000	8	99802.54	1.679911

java DriverSort quicksort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	1	10302.028	2.1126032
2000000	32	21224.145	1.883275
3000000	128	23677.773	2.1847277
4000000	8	32773.555	1.7017943
5000000	2	60569.344	1.2498902
6000000	4	61025.824	1.4849873
7000000	8	63698.434	2.0543718
8000000	1024	74192.24	1.7659857
9000000	32	88782.24	2.552984

java DriverSort mergesort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	196.6125	11.242547
20000	2	207.1429	1.8025204
30000	8	326.062	3.6143384
40000	4	410.2487	1.8002301
50000	2	527.3622	1.7080543
60000	16	640.354	3.0322597
70000	2	750.3705	1.6079289
80000	2	867.3288	1.8797703
90000	8	1269.8363	1.7056675

java DriverSort quicksort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	51.9492	5.311408
20000	1024	63.0427	2.1007125
30000	16	91.397	1.7881353
40000	32	110.213	2.0522795
50000	8	143.4761	1.9502662
60000	8	168.5063	2.1147141
70000	16	209.5552	1.9600285
80000	8	261.9612	1.8089217
90000	32	258.6407	2.1103106

java DriverSort mergesort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	1	511.4955	7.871281
200000	2	772.3125	2.325302
300000	4	1213.3624	2.1605012
400000	4	1568.0651	2.1469529
500000	4	1980.0236	2.1782696

600000	4	2522.6145	2.097935
700000	4	2953.8726	2.0871177
800000	4	3271.7102	2.1937032
900000	4	3754.1587	2.1749127

java DriverSort quicksort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	1	317.4351	2.4985785
200000	64	648.2276	1.9808213
300000	4	1090.9164	1.9408166
400000	32	1326.873	2.147535
500000	16	1607.0919	2.1828527
600000	32	2172.8237	1.9826161
700000	8	2293.56	2.1940284
800000	64	2905.6318	1.9963104
900000	32	3060.2715	2.145665

java DriverSort mergesort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	1	4174.322	2.4184687
2000000	4	8614.06	2.1978083
3000000	4	12950.432	2.2368755
4000000	4	16895.354	2.2591245
5000000	4	22151.986	2.2701797
6000000	4	25427.79	2.3160949
7000000	4	31992.572	2.3087742
8000000	4	35843.434	2.2252626
9000000	4	31754.234	2.2361833

java DriverSort quicksort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	4	3398.7036	2.0637252
2000000	64	6310.1733	2.3742642
3000000	64	11561.912	2.203404
4000000	64	16868.553	2.0346906
5000000	64	17853.338	2.1744022
6000000	64	21675.555	2.1303895
7000000	512	25191.816	2.235642
8000000	128	30301.8	2.0664937
9000000	128	31754.234	2.3361833

java DriverSort mergesort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	459.1455	6.1198087
20000	2	416.4298	7.539743
30000	128	962.0283	7.1588664
40000	32	318.1833	6.046818
50000	4	716.3114	2.9985914
60000	4	474.4962	2.526188
70000	4	625.5074	6.017147
80000	512	1500.9229	7.925475
90000	128	2589.3892	5.6692038

java DriverSort quicksort 10000 100000 10000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
10000	1	107.8444	7.6596236
20000	128	434.507	7.4857464
30000	4	287.0284	1.4943861
40000	8	565.786	3.5509367
50000	512	740.232	5.933151
60000	256	1013.4824	3.1619418
70000	8	653.0505	2.650259
80000	16	1136.2971	5.1432576
90000	16	1654.6926	4.2791934

java DriverSort mergesort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	16	865.2018	4.4624352
200000	1	3052.0657	2.2168345
300000	8	4632.269	3.0081277
400000	64	6040.976	3.4282663
500000	8	8359.834	3.3512597
600000	4	8148.4326	2.7116373
700000	32	7929.0024	1.9961952
800000	64	12081.545	2.0435164
900000	16	13451.659	1.9177115

java DriverSort quicksort 100000 1000000 100000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
100000	32	2670.981	3.9415252
200000	128	1654.4756	4.3575597
300000	1024	2274.047	4.0876102
400000	16	4962.1177	3.6645045

500000	8	5226.014	2.60253
600000	128	7762.51	2.324037
700000	4	8420.539	2.3696065
800000	512	12053.002	1.9976411
900000	32	12919.95	2.438518

java DriverSort mergesort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	1	9532.47	3.8265934
2000000	16	23301.598	2.547542
3000000	64	37068.406	2.1105363
4000000	32	37577.348	2.88918
5000000	4	60849.453	3.48952
6000000	16	56523.355	2.7267118
7000000	4	66372.3	3.472027
8000000	32	82005.46	2.35464
9000000	4	99721.164	3.4429827

java DriverSort quicksort 1000000 10000000 1000000 file

Array Size	Optimal Num of Threads	Best Time	Best Speedup
1000000	128	10817.44	2.8092499
2000000	32	19806.387	3.5519598
3000000	8	17495.475	3.588231
4000000	1024	16478.527	4.894003
5000000	32	50966.586	2.665486
6000000	32	59962.453	3.1678824
7000000	16	60949.51	2.997327
8000000	64	62328.434	3.2927246
9000000	32	88782.24	2.552984