Experimental and Theoretical Speedup Prediction for Merge Sort and Quick Sort

Ntuthuko Mthiyane (MTHNTU003)

mthntu003@myuct.ac.za

1. Introduction

The aim of this experiment is to do a performance test between parallel sorting algorithms and their serial version (Merge sort, Quick sort and Bubble sort). The main function of sorting algorithms is to place data elements of a list in a certain order.

A parallel algorithm is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result (ComSIS Vol. 10, No. 3, June 2013). Parallel algorithm are expected to yield improved performance on many different kinds of computers/ architectures, there will be a speedup if an algorithm is paralyzed (Speedup is defined as the ratio of serial execution time to the parallel execution time, it is used to express how many times a parallel program works faster than its serial version used to solve the same problem).

Sequential sorting algorithms are based on comparing the data items to find the correct relative order, this is achieved by comparing two items at a time. Parallelizing sorting algorithms repeatedly divides the original list into sub-lists until the sub-lists and compares these sub-lists simultaneously. These elements are then merged together given the sorted list as in case of merge sort.

The theoretical speedup prediction, for parallel merge sort the estimated complexity in case of using a dual core processor is 2/P*log(n/m) + total overhead and for parallel quick sort the estimated complexity for a dual core processor is 2n/mP*log(n/m) + total overhead (ComSIS Vol. 10, No. 3, June 2013). The complexity of arrays.sort is n*log(n) therefore the expected speed up is the ratio of n*log(n) over the complexity or quick sort and that of merge sort.

2. The proposed methods

The proposed method is summarized in the following steps:

1. Obtain a number of threads in the machine with the code is tested on.
2. Generate a random array with N elements and make a copy of that array.
3. Execute the parallel merge sort algorithm on an M core machine to sort the array generated in 1.
4. Apply the build in “arrays.sort” function to sort the copy of the array in [2].
5. Record the time taken to sort the array with N elements.

6. Record the time taken to sort the copy of the array with N elements for each run.
7. Repeat steps two to six 5 times.
8. Record the minimum time taken to sort the array in the 5 iterations as best time.
9. Calculate the speed up using the best time obtain in [8].
10. Add the results to a text file (Results including array size, optimal number of processors, best time and best speed up).
11. Increase N to N + increment.
12. Repeat the above until N is great or equals to array size max.
13. Repeat steps 2-12 changing the sort in [3] to quick sort then bubble sort.
14. Do the above on three different machine each with a different number of cores.
15. Graph the obtained results as a two dimensional graphs.

The algorithm was validate using a Junit test, the resulting sorted array from a parallel class was compared to the resulting sorted array using the Java arrays.sort function. The time taken to sort an array using these two ways was compared to valid that the parallel algorithm is faster than the serial one. The timing of algorithms were timed in the following ways, the parallel algorithm was timed by recording the time just before it started the sorting process and the time just right after it was done with sorting, the time taken was then calculating the difference between the two times. The serial time was calculated in a similar way, the time just before the arrays.sort function was called was recorded and the time immediately after the sort was also record, the time taken for the sort was calculated as the difference between the two times.

Speedup is the ratio of serial execution time to the parallel execution time, therefore it was calculated by diving the serial execution time by the best parallel execution time obtained in [8]. The algorithms were tested on three different machines each with a different architecture (a 2 core machine, 4 core machine and 8 core machine).

Parallelizing sequential algorithms adds several extra challenges such as application speedup and how can it be affected by the number of cores and/ or the number of the running processes. Problems encountered in this experiment was getting the paralyzed algorithms to be both faster and corrected, getting the program to be faster was the biggest challenge as a result the quick sort was not fast enough, it is a bit slower than parallel merge sort and that should not be the case. Another difficulty was getting the program to run a terminal instead of an IDE and creating JUnit for each class.

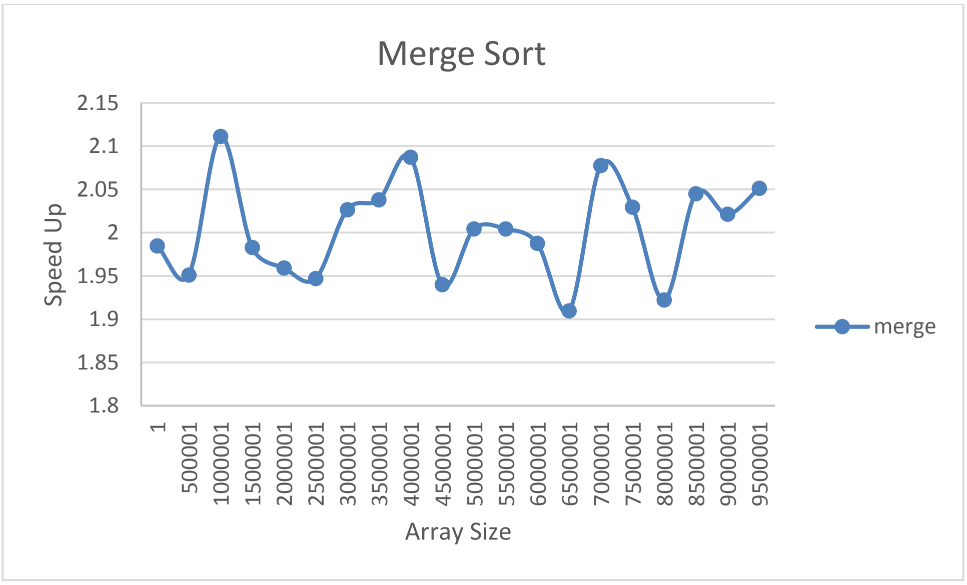
3. Results

The speed up of each algorithms is dependent on numerous factors, the first thing that was investigated was the effect of data size. An experiment was done on both quick and merge sort. The relationships have been graphed below.

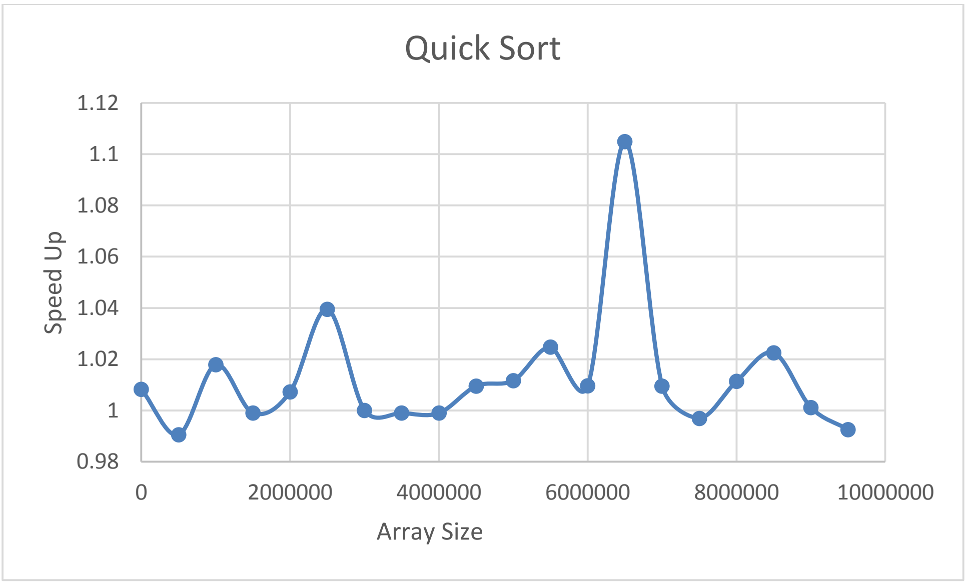
Tabular Results

Data Size	Merge Sort	Quick Sort
1	1.98478	1.00819
500001	1.95117	0.99048
1000001	2.11111	1.01780
1500001	1.98276	0.99897
2000001	1.95923	1.00726
2500001	1.94692	1.03949
3000001	2.02661	1.00001
3500001	2.03803	0.99895
4000001	2.08716	0.99894
4500001	1.93991	1.00949
5000001	2.00431	1.01165
5500001	2.00422	1.02465
6000001	1.98747	1.00956
6500001	1.90964	1.10488
7000001	2.07742	1.00942
7500001	2.02935	0.99691
8000001	2.00122	1.01134
8500001	2.04497	1.02244
9000001	2.02110	1.00109
9500001	2.05117	0.99249

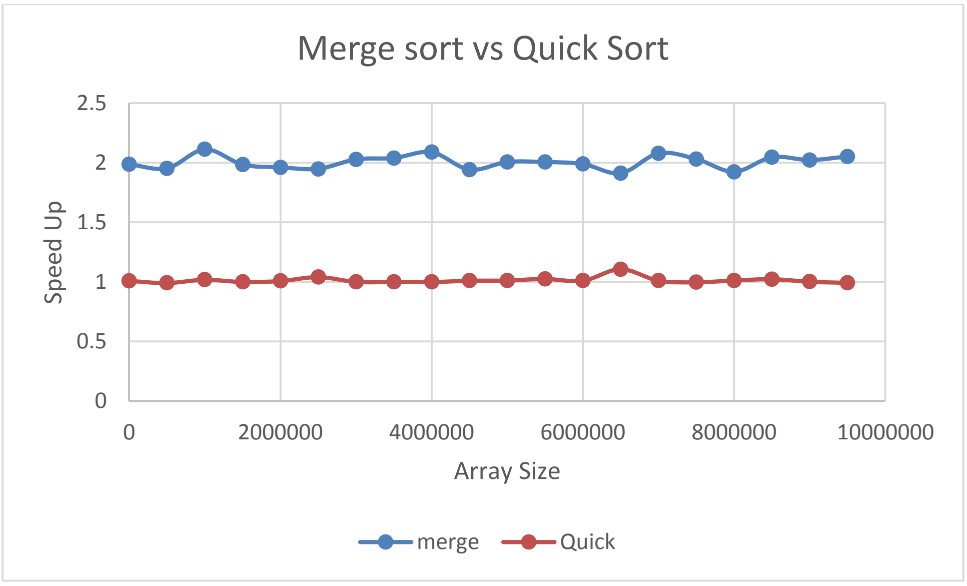
The speed up for merge sort



The speed up for quick sort



The speed up for quick sort vs merge sort



Merge sort seems to be faster than quick sort as the data size increases, the speed up of changes with data size, with merge sort the increase is linear and with quick there is no straight pattern (There is no relationship).

a) Merge sort performs well as the array size gets big, when the array size is a million and above that is when a higher speed up is observed. For quick sort the speed up did not change even with an increase in array size.

b) Maximum speed up obtained for merge sort is 2.05117 and the maximum speed up for quick sort is 1.02 The speed up for merge sort is a bit close to the ideal speed up and for quick sort it is not close enough, it is a lower than the expected speed up.

c) Merge sort is somehow a few split seconds faster than quick sort, even though this should not be the case.

d) Speed up increases with an increase in cores, a machine with fewer number of cores produces a lower speed up.

e) The optimal number of cores were 2, 4 and 8.

f) No. The difference in time is a few split seconds, you can hardly observe the difference.

4. Conclusions

From the experiment it can be concluded that parallel algorithms do improve the performance, an algorithm can be 2 or more times faster when if paralyzed. This depends on the number of processors (architecture) and how well the algorithm is implemented.

From the results it was drawn that merge sort is faster than quick sort (even though that is not what was predicted). The implementation of quick sort was not effective as it did not perform how it was expected.

The results are reliable as the algorithm was tested on difference machines and the testing was done more than 5 tests and the same results were produced.

The theoretical predicted speedup has been compared with the experimental speedup for the merge and unfortunately not for quick sort. Our theoretical prediction of speedup was very close to the experimental results, this gives a good indication about the scalability of the proposed method.