NJBAS

*Available online at http://www.ajol.info/index.php/njbas/index
Nigerian Journal of Basic and Applied Science (June, 2021), 29(1): 43-48
DOI: http://dx.doi.org/10.4314/njbas.v29i1.5

ISSN 0794-5698

Full Length Research Article

Comparative Analysis between Selection Sort and Merge Sort Algorithms

***[1]A. M. Rabiu, [2]E. J. Garba, [3]B. Y. Baha and [4]M. I. Mukhtar**
[1]Computer Science Department, Federal University, Dutse, Nigeria
[2]Computer Science Department, Modibbo Adama University of Technology, Yola, Nigeria
[3]Department of Information Technology, Modibbo Adama University of Technology, Yola, Nigeria
[4]Department of Software Engineering, Bayero University, Kano, Nigeria
[*Corresponding Author: E-mail: mambas86@fud.edu.ng; ☎: +2348036408434]

ABSTRACT
Sorting and merging are two problems that commonly arise in Computer Science especially in data processing tasks. To solve these problems, several algorithms have been developed. Similarly, existing merge and sorting algorithms have been improved to provide more efficient and accurate results. In this paper, selection and merging algorithms were developed on an octa-core processing machine using System.nanoTime methods in Java in order to compare their running times. The results obtained show that Merge Sort performs far better than selection sort with careful implementations by taking advantage of multiple processing cores in the test machine and some concurrency utility in Java. It was concluded that implementing algorithms using a machine with multiple numbers of cores in their Central Processing Unit (CPU) will result in a significant improvement in the performance of both algorithms.
Keywords: Algorithms, Concurrency, Machine, Merging, Running Time, Sorting.

INTRODUCTION
Two of the common problems in data processing and computer science, in general, are sorting and merging algorithms. Sorting refers to the arrangement of data in statistical order either in increasing, decreasing or lexicographical order while merging on the other hand employs a divide and conquer approach to sort a given array of elements (Rabiu et al., 2018; Robert, 2002). There are many merging and sorting algorithms that have been developed to solve the problems of merging and sorting large data. So also older sorting and merging algorithms have been improved upon to lower their running times and increase their speed to make them more efficient (Rabiu et al., 2018). Some of the sorting algorithms were developed as non-comparison-based sorts and it was further established that quicksort is a better sorting algorithm than selection sort (Robert, 2002).

Performance of quicksort is better than that of selection sort (Thomas *et al.,* 2004) and Shell sort was far better than selection sort (Adhikari and Pooja, 2007; Muhammad et al, 2017; Göetz et al., 2006). Linear search is also known as a sequential searching algorithm while binary search, in contrast, is based on the divide and conquer approach (Knuth, 1997; Zhuoer et al. 2011; Zongli, 2010; Sengupta, 2007). The time complexity of linear search was in the order of O(n) while that of binary search is in the order of O(log n) (Knuth, 1997; Thomas *et al*., 2004; Ankit and Chadha, 2014; Mishra and Garg, 2008). Different parallel algorithms for linear algebra were explained and the results show that performance improvement could be achieved by careful implementation of some parallelization techniques (Aleksandar, 2014).

To show how the performance improvement could be achieved two concurrency frameworks namely: "ServExecSort" and "NaïveParallelSort" were compared on multi-core machines. The results show that the "ServExecSort framework" performs far better than "NaïveParallelSort" (Rabiu *et al*., 2020). Some selected machine algorithms used to predict cardiovascular disease were surveyed and their performances were compared. Investigation on the 18 types of research so far conducted shows that Decision Tree (DT)-J48 NB (Naïve Bayes) NB, and Support Vector Machine (SVM) appeared more frequently with RF having the least frequency. It was concluded that no single algorithm would be generalized to be the best in Cardio Vascular Disease (CVD) prediction (Yahaya *et al.*, 2020). Multicore processors can

be used to improve the performance of concurrent applications (Kaya, 1995; Ganesh and Sondhi, 2018). Most of the literature reviewed focused on measuring the performance of algorithms by considering some factors such as memory space and time complexities only to measure their performances, failing to take full advantage of multi-core processors and newer concurrency mechanisms to develop and improve the performance of these algorithms. To fill in the gap, this paper took advantage of multiple processing cores in an octa-core machine to measure the running times of selection and merging algorithms to compare their running times so as to establish co-relation between the numbers of processing cores and the performance gain.

MATERIAL AND METHOD

Hardware Specifications

The following hardware specifications were used for benchmarking in this paper. Firstly, a single-core computer with 1.5Hz Core, Windows 7 Operating System 32bits (OS) was used to develop and run the program. That gave us the basis for comparison with the results obtained using machines with a higher number of processing cores. Computer with eight processing cores in its CPU was then used on Windows 8, 64 bits Operating System, having a frequency of 1GHz each, 2GB (64bit), Disk space 20GB (64**bit**) to develop the two algorithms of choice.

Software Specifications

Concurrency Tools

All concurrency mechanisms used to develop the algorithms in this paper were those provided by the Java programming language. They included a thread pool for the creation and management of threads, a framework for asynchronous and synchronizations of threads and task executions such as counting semaphores, lock, atomic and condition variables.

Java Development Kit (JDK)

The JDK 10 version was used in the development of the two algorithms in this paper. They were found to be more efficient than the previous version of JDK because of the new features included that could be fully utilized to achieve good results.

Array Data Structure

The data structure used in this research is the array data structure as shown in Table 1. An integer was generated to fill the array with data. This structure contained an array size ranging from 5000 to 70,000. The increase in the number of test runs is to minimize the effects of background programs that can affects measurement thereby minimizing the overheads.

Table 1: Defined array data structure

NO OF RUNS	ARRAY SIZE(N)
1000	5,000
2000	10,000
3000	20,000
4000	30,000
5000	40,000
6000	50,000
7000	60,000
8000	70,000

Data Generation

Having defined the size of the array, it was then filled with the appropriate data type suitable for merging and selection of elements.32-bit integer was used in this paper. This is because int (integer) in Java can contain positive values (2^{15}-1) ranging from 1 to about 2.1 billion and was found to be more appropriate with defined array structure used in this paper.

Algorithm Benchmarking

Two built-in functions are mostly used to measure the start and the end time in java. Namely: System.currentTimeMillis() and System.nanoTime() methods. Since this paper is interested in measuring the running time of algorithms only, System.nanoTime() method was the method used to measure the running times of both the merging and selection

algorithms. System.nanoTime() gives more precise results suitable for comparison.

Selection Sort Algorithms

Selection sort is one of the comparison-based sorting algorithms. It checks an array of elements and tries to find the smallest element in the array. It then exchanges the smallest element with the element in the first position. After finishing this step, this algorithm tries to select the smallest element from the unsorted part of the array after each iteration is carried out. It then exchanges the selected smallest element with the element in the unsorted part of the array. This process continues until all elements in the array are completely sorted (Aliyu and Zirra, 2013; Mishra and Garg, 2008).

Pseudo-code of Selection Sort according to Insertion Sort (2019)

```
i ← 1
while i < length(A)
j ← i
while j > 0 and A[j-1] > A[j]
swap A[j] and A[j-1]
j ← j - 1
end while
i ← i + 1
end while
```

Performance Analysis of Merge Sort

Merge Sort employs the "divide and conquer approach" to sort a given array of elements. "It works by dividing the input array into two halves, and then merge-sort recursively calls itself for the two halves and merges the result of the two sorted halves" (Geeks for Geeks, 2019).

"Merge-Sort (arr [], l, k), If k> l
Step 1: Determine the middle element so as to dividethe array into two halves
Middle M = (l+k)/2
Step2: Call the Merge-sort for the 1st half:
Call the merge-sort (arr, l, M)
Step3: Call the Merge-sort for the 2nd half:
Merge-sort (arr, M+1, k)
Step 4: Merge the two sorted halves in step 2 and step 3:
Call the Merge (arr, l, M, k)"
(Geeks for Geeks, 2019)

RESULTS

Performance Analysis of Selection

Running time of selection sort algorithm using a different number of array sizes and test runs is shown in Table 2. For each array size number of runs was repeated several times. The number of run was varied as the size of the array increases. The reason for doing this is to minimize the effects of the background program in our measurements and to summarize the collections of test runs by a single typical value suitable for comparison.

Table 2: Running times of selection sort algorithm

NO OF RUNS	ARRAY SIZE(N)	RUNNING TIME(SEC)
1000	5,000	0.392
2000	10,000	0.712
3000	20,000	2.140
4000	30,000	4.765
5000	40,000	9.182
6000	50,000	13.311
7000	60,000	19.302
8000	70,000	23.465

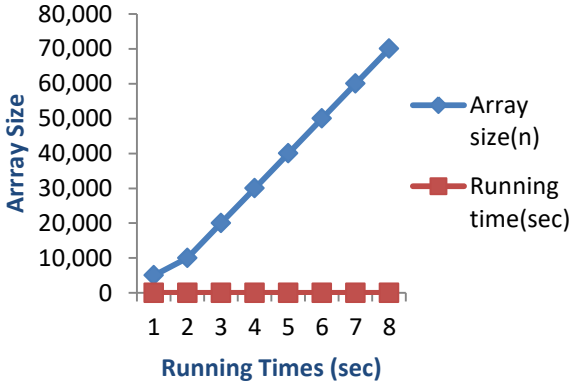

Figure 1: Running times of selection sort algorithm

Table 3: Running times of merge sort algorithms

NO OF RUNS	ARRAY SIZE(N)	RUNNING TIME(SEC)
1000	5,000	0.019
2000	10,000	0.040
3000	20,000	0.073
4000	30,000	0.134
5000	40,000	0.170
6000	50,000	0.192
7000	60,000	0.205
8000	70,000	0.512

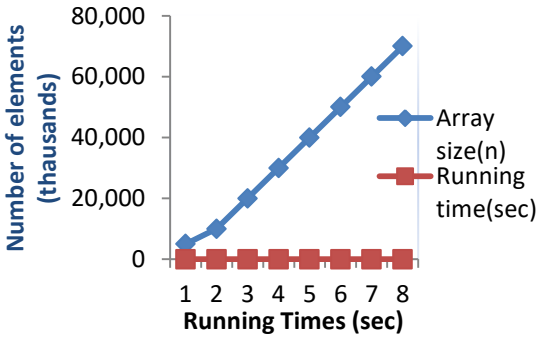

Figure 2: Running times of merge sort algorithm

Table 4: Performance comparisons between selection sort and merge sort algorithms

ARRAY SIZE	RUNNING TIME OF SELECTION SORT (sec)	RUNNING TIME OF MERGE SORT (sec)
5,000	0.392	0.019
10,000	0.712	0.040
20,000	2.140	0.073
30,000	4.765	0.134
40,000	9.182	0.170
50,000	13.311	0.192
60,000	19.302	0.205
70,000	23.465	0.512

DISCUSSION

From Table 2, and Figure 1, it is noticeable that the running times of selection sort increase as the size of the array. This is in accordance with the findings of Rabiu *et al*. (2018). Therefore, it can be observed that the times it takes to sort a given element in the array are dependent upon the number of elements within the array using selection sort

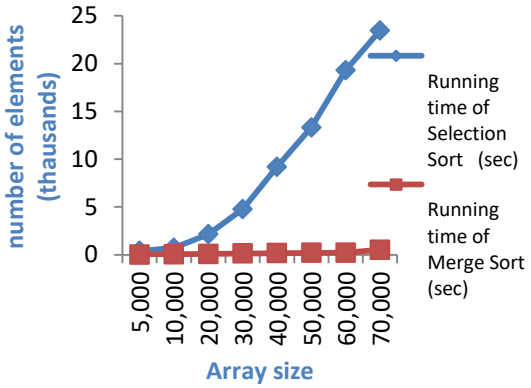

Figure 3: Performance of selection sort and merge sort algorithms

When the array sizes sorted were around 10,000 the time taken is 0.712s and when the array size increases to 20,000 the time taken to sort the given array increases to 2.140s. The difference between the two sorting times is 1.428s which is almost 1.5 times. It can also be observed that when the array size increases from 30,000 to 40,000 elements with the corresponding running time of 4.765s and 9.182s respectively, the difference between the two running times is approximately four times. Unfortunately, when the size of the array is doubled, the time required for sorting it with selection sort increases four times, making it less effective.

Therefore, increasing the size of the array by a factor of 2 will lead to the corresponding increase in the sorting times by a factor of 200 using selection sort. These can be observed from the results obtained in Table 2 and Figure 1 respectively. From Table 3, Figure 2, Table 4, and Figure 3 it can be similarly observed that the merging sort algorithm exhibits better performances on both smaller and larger array sizes as compared to the Selection sort algorithm throughout the sorting process. This is in agreement with the findings made by Rabiu et al. (2020) and Aliyu and Zirra, (2013) that the performance of algorithms is a factor of the input size of the array. It can be seen from Figure 4 that algorithms whose "time complexity" is in the order of O (n log(n)) exhibit better performances when compared with those algorithms whose time complexity is in the order of O(n^2) as in the case of selection sort.

This is in line with the findings of Aliyu and Zirra, (2013); Rabiu et al. (2018); Thomas *et al*., 2004; Ankit and Chadha, 2014; Mishra and Garg, 2008) that the order in which complexity of a given algorithm is defined determines the efficiency of that particular algorithm. Based on the results obtained, and the comparisons made so far, from Table 2, Table 3, Table 4, Figure 1, Figure 2, and Figure 3, suppose that both merge sort and selection sort algorithms take 0.712s to sort an array of 10,000 elements on the same octa-core machine. Then, it would take about 0.712 × 150, which equals 106.8s which is less than two minutes to sort a million array sizes. However, it would take selection sort more than two hours to sort the same array elements. Hence, it is worth time and effort to spend several hours learning about a better algorithm no matter its complexity than using a simpler one which could be learned in less time.

CONCLUSION

From the results obtained so far, it can be concluded that merge sort is the better sorting algorithm considering the size of the data sets used throughout the experiment. Secondly, increasing the processing core also increases the performance of both the selection and merge sort algorithms. It was also concluded further that the order of complexity of an algorithm determines its efficiency. Other popular algorithms such as quick-sorts, heap-sort, and insertion-sort have the potentials to exhibit similar behaviour and performances using the same approaches on different test machines with a different number of processing cores in their CPUs. Hence, these algorithms and other popular ones deserve further research to see if they can give the same results when measured on different machines.

ACKNOWLEDGEMENTS

We must acknowledge the efforts and contributions of the former Director of Information and Communication Technology (ICT) Federal University, Dutse, Jigawa State, Nigeria, in the person of Prof. Ahmed Baita Garko who despite his tight schedules took his time to go through this, and other related manuscripts to make some recommendations and suggestions on how to improve on their contents.

REFERENCES

Adhikari, P. & Pooja, A. (2007). *Review of sorting algorithms: A comparative study of two sorting algorithms.* Mississippi State, Mississippi Press, Pp. 20-24.

Aleksandar, V. (2014). Manual parallelization versus state-of-the-art: parallelization techniques.

Ankit, R. & Chadha, (2014). Modified binary search algorithm. *International Journal of Applied Information Systems, (IJAIS),* **7**(13): 1-10.

Aliyu, A. M. & Zirra, P. B. (2013). A comparative analysis of sorting algorithms on integer and character arrays. *The International Journal of Engineering and Science, .***2**(7): 25-30.

Ganesh, A. & Sondhi, G. (2018). An overview of multi-core. *International Journal of Innovative Science and Research Technology,* **3**(4): 261-263.

Geeks for Geeks. (2019*). Merge sorting algorithms*. Pp.1-10. Retrieved from: https://www.geeksforgeeks.org/merge-sort/. Accessed 3rd Jan., 2020.

Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., David, H., & Lea, D. (2006). *Java concurrency in practice.* Addison, Wesley Professional. Pp. 1-10.

Insertion Sort (2019). *Insertion Sort.* Retrived from: https://en.wikipedia.org/wiki/Insertion_sort.Accessed 23rd Mar. 2018.

Kaya, D. (1995). Parallel algorithms for numerical linear algebra on a shared memory multiprocessor [Doctorate Thesis].*The University of Newcastle Upon Tyne Department of Computing* Science, 1-123.

Knuth, D.E. (1997). *The art of computer programming, sorting and searching* (3rd ed.). New York, Addison Wasley, Pp. 395-409.

Mishra, A. D., & Garg, D. (2008). Selection of the best sorting algorithm. *International Journal of Intelligent Information Processing,* **2**(2): 363-368.

Muhammad R. A., Harith Z., Farouk S., Dauda B. (2017). *Comparison of bubble sort and selection sort with their enhanced versions.* Department of Electrical Engineering Univeristy of Lahore, Pakistan.1-10.

Rabiu, A.M., Garko, A.B., &Abdullahi, A.M. (2018).Effects of multi-core processors on linearand binary sorting algorithms.*Dutse Journal of Pure and Applied Sciences,* **2**(4):130-140

Rabiu, A.M., Garba, E.G., Baha, B.Y. & Mukhtar, M.I. (2020). Optimizing frameworks for building more efficient concurrent application in Java.*Islamic University Multidisciplinary Journal (IUMJ),* **7**(2):348-355.

Rabiu, A.M., Garko, A.B., Abdullahi, A.M, Umar, H.A., & Babagana, M. (2018). Performance evaluation of three quick-sorting algorithms on single and multi-core processors. *Dutse Journal of Pure and Applied Sciences* **2***(*4), 254-263.

Robert, L. (2002). *Data structures and algorithms in java* (3rd Ed).Retrieved from:http://www.resaechgate.net.Accessed 2nd Oct. 2018.

Sengupta, D. L. (2007). *Algorithms in java* (3rded.). New York, A. Wasley, Pp. 5-23.

Suleiman, A. K. (2013). Review on sorting algorithms: A comparative study. *International Journal of Computer Science and Security* (IJCSS), **3**(7):120-126.

Thomas, H. C., Charles, E. L., Ronald, L. R. & Clifford, S. (2004). *Introduction to Algorithms* (4th Ed.).NY: Addison-Wesley Professional, 50-51.

Yahaya, L., Hassan, I. & Rabiu, A.M. (2020). A survey of performance of some selected machine learning algorithms for cardiovascular disease predictions. *BIMA Journal of Science and Technology,* **4**(1): 165-180.

Zhuoer, L., Chenghong, Z., & Yunfa, H. (2011). Backwards search algorithm of double-sorted inter-relevant successive trees. *Fifth International Conference on Fuzzy Systems and Knowledge Discovery,* **3**(23)*:*2-12.

Zongli, J. (2010). A tag feedback based sorting algorithms for social search. *International Conference on System and Informatics, (ICSAI2012),***3**(2): 12-32.