Optimizing Frameworks for Building More Efficient Concurrent Application in Java

RABIU A. M.[1], GARBA E. J.[2], BAHA B. Y.[3] and MUKHTAR M. I.[4]
[1]Department of Computer Science, Federal University Dutse, Nigeria
[2]Department of Computer Science, 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
mambas86@fud.edu.ng, e.j.garba@mautech.edu.ng, bybaha@yahoo.com, mimukhtar.se@buk.edu.ng

Abstract
Nowadays, there has been an increasing demand for high-speed computation, parallel processing, and efficient algorithms to efficiently manage the rapid increase in the volume of data which gave rise to much research being carried out in concurrent applications and mechanisms to improve performances. In this paper, two parallel concurrency frameworks were developed and tested using the quick-sorting algorithm on a multiprocessor machine to compare their performances. System.nanoTime() method was used for benchmarking the two frameworks. The results show that the ServExecSort framework performs better than NaïveParallelSort on a larger array size while the NaïveParallelSortr exhibits better performance on smaller array elements. It is therefore concluded that a framework that uses separate thread pool and future to keep track of submitted tasks exhibits better performances than a framework that creates separate threads for each task which leads to large overheads and performance degradation.

Keywords: Algorithms, Computation, Efficiency, Framework, Performances, Mechanism.

Introduction

Java as one of the first Object-Oriented Programming (OOP) language to support concurrency and offered multithreading had introduced some low-level primitives in Java 5 (also known as Java 1.5). The low-level mechanisms offered such as synchronized(), notify(), and wait() were difficult to use correctly and some problems such as starvation, livelock, and deadlock could not be avoided (Rabiu, Garko and Abdullahi, 2018; Brad, Paul, and Luke, 2007; Aleksandar, 2014). Additionally, the low-level concurrency primitive could not be used to build a more efficient concurrency framework that can solve sophisticated concurrency related problems (Rabiu et al, 2019; Brad, 2009). With the recent advancement in Operating System (OS) which included multi-threading in its kernel level structure and also the processor architecture, concurrent programming and application have become necessary in real-world applications. A programming language such as C++ and Java provided supports for concurrent application development (Doug, 2015; Hongwei, Yin & Hyoun, 2013). Java programming language which included Java Virtual Machine (JVL) provided the necessary supports for developers to implement concurrency mechanisms and multithreading to improve the performance of their concurrent software (Rabiu et al, 2018). Research carried out by Yaniv, 2016; Pavel & Georgi, 2012) further stated that multithreading enables developers to achieve parallelism and synchronization between threads to protect data, deadlock, and starvation. In this paper two different concurrency frameworks were implemented namely: ServExecSort and NaïveParallelSort to handle the creation and termination of threads using multi-threading and concurrency tools. ServExecSort works by creating ThreadPool for a fixed number of threads. Each thread is then assigned several tasks to be executed in parallel and is being tracked using a 'Future services. When the entire tasks are completely executed, the ThreadPool is shutdown. On the other hand, the NaïveParallelSort framework works by creating threads for each task that needs to be executed during each recursive call. Each processing core in the machine is then assigned available threads to do the computation in parallel. This results in improved performances. Both the two frameworks were tested using the quick-sorting algorithm to measure and compare their performances.

Literature Reviews

To improve the performance of concurrent programs and to minimize programming efforts many software frameworks were developed. Some good examples are: "a library-based framework for heterogeneous multi-core systems" presented by (Michael et al., 2008); "A framework for parallel execution of codes on multiple GPUs" developed by (Zhe, Feng, & Arie, 2018); and another one titled "new programming language for general-purpose computation on the GPU" was also developed by (Qiming, Kun, & Baining, 2018). A framework designed to conform to automatic learning style detection process which comprised of model development and could be integrated into any adaptive learning style was developed

by (Farouk *et al.*, 2020). Another guidelines for tailoring persuasive health interventions to promote individuals' SWB based on their personality was established by (Abdullahi, Orji, Rabiu & Kawu, 2020). Java multithreading mechanism was realized and analyzed together with some concurrency mechanisms that helped in developing concurrent applications. The study further compared some concurrent structures using the different operating systems. The researchers summarized some programming rules that could be used to prevent concurrency issues. It was concluded that efficient synchronization could help developers prevent deadlock and starvation (Clay, 2009; Wuxue, Qi, Zhiming, and Jianfeng, 2013)

Different synchronization mechanisms to reduce the Heisenberg form of a matrix were explained. The results obtained from the experiment showed that dynamic task allocation could be very effective on the machine used for the benchmarking; high efficiency was obtained with careful design and implementations of parallel algorithms even for relatively small matrix (Kaya, 2015; Robert, 2017; Yaniv, 2016). Various parallelization techniques were discussed and various software and hardware tools used for building concurrent applications from the sequential one were also explained. Profiling was the method used to measure the effects of parallelism on two parallel machines. The results show that performance improvement could be achieved by careful implementation of some parallelization techniques (Aleksandar, 2014). A program can be called concurrent if it is divided in such a way that two or more threads can progress at some time. This does not mean that they have to progress simultaneously, but that they can be swapped in and out by the operating system on a single core (Clay, 2009). According to (Mansoor & Alarcon, 2017) the parallel is when two or more threads progress at the same time, requiring multiple cores and where each thread is assigned to a separate processing core. Increasingly, Multi-core processing computers have improved ICT utilization in health care systems due to their ability to carry out multi-tasking activities and to execute tasks concurrently (Rabiu, Mukhtar, Abbas & Yusuf, 2019). According to Rabiu *et al.*, (2019), high performance computers with multiple processors are also suitable for carrying out E-commerce activities efficiently due to their speed and less overheads.

The Java class 'thread' is the whole foundation for all Java concurrency frameworks. This class makes it easy for the developer to create and execute threads. Initially, a thread is identical to normal Java classes, but except for having a runnable task which can be executed concurrently with the main program. When these threads are executed, the Java Virtual Machine (JVM) and the operating system (OS) decide which processor cores will be applied (Intel Corp Desktop Products Group, 2009). Threads are an unavoidable feature of Java programming language. They can be used to simplify the development of some complex programs; this can be done by changing the asynchronous line of codes into some simpler straight line of codes and they are said to be the easiest way to tap the computing power and enjoy the benefit of multi-core processing machines (Göetz *et al.*, 2006; Jeremy *et al.*, 2018; Michael *et al*., 2018). Therefore, as the power of multiprocessor increases, developing concurrent software will become more important as well. The same need for resource utilization convenience and fairness that motivated the development of processes to ensure the efficiency of the program is also the same concern that motivated the development of threads. They allow multiple streams of program control flow to take place within a given process. Threads often share some wide resources of a process such as file handles and memory with each thread having its stack, program counter, and local variable variables. They also provide a natural way of decomposition that enables the exploitation of hardware parallelism on multi-processor machines. To achieve these multiple threads within the same program can be scheduled to run simultaneously on multiple processing cores of CPUs (Göetz *et al.*, 2006; Pavel and Georgi, 2012; Rabiu *et al.*, 2018). Before version 5 of Java was introduced the only way to synchronize threads was through the low-level concurrency control mechanisms such as synchronized, volatile, wait(), notify() and notifyAll(). They are all difficult to use correctly and the potential for common concurrent threats like deadlock and starvation is high (Jeremy *et al.*, 2018). Java 5 which was originally known as Java 1.5 changed this by including new packages named java.util.concurrent which included two new sub-packages called atomic and locks, which allows a programmer to have a high-level synchronization on threads. These packages have been improved and updated by including some new classes and interfaces in versions 6.0, 8.0, and other subsequent versions of Java respectively. These give the developer additional tools to use in developing concurrent software.

With the rapid increase in the number of processing cores in parallel and multi-core computers, developers will need to change their mindsets and focus more on concurrent application development to enable them to achieve the desired performance gain in their programs. Today, different libraries and mechanisms are available for synchronization and parallelization in Java and perhaps other programming languages. Therefore, this paper used some of the concurrency mechanisms provided by Java in the development of two frameworks to optimize their performances. Quick-sorting algorithms were implemented to test the two frameworks by taking advantage of different processor core available in the parallel computer.

Methodology

Array Data Structure

The data structure used in this research is the array data structure. A set of integers was generated to fill the array with data. This structure contained an array size ranging from one thousand to 9 million sizes. This is to enable us to test our frameworks using sorting algorithms to make reasonable comparisons to conclude the results. Different numbers of test runs were used to reduce the effect of background program in our measurement especially at the beginning when the array size is small. As the size of the array increases, the computation time also increases. Therefore, the number of test runs is decreased with the increase in array size. Table 1 contains the array size and the corresponding test runs for each array size.

Table 1: Array Data Structure

Array Size	Number of Runs	Array Size	Number of Runs
1000	800	400,000	200
4000	800	500,000	150
6000	800	600,000	140
8000	750	700,000	120
10000	750	800,000	110
20,0000	600	900,000	100
30,0000	600	1,000,000	90
40,0000	550	2,000,000	80
50,000	500	3,000,000	65
60,000	450	3,500,000	70
70,000	400	4,000,000	60
80,0000	440	5,000,000	50
90,000	350	6,000,000	40
100,000	300	7,000,000	30
200,000	250	8,000,000	20
300,000	220	9,000,000	10

Data Generation

Having defined the size and corresponding test runs of our arrays, the next thing to do is to fill the array with the appropriate data type for sorting. The most common data use for sorting and is positive integers for example, [1, 2, 3, 5, 8, 10…n]. In Java short, int, long can be used as variables. 32-bit int is our defined data structure which is considered to be the best for this array size. This is because int in Java can contain positive values (2^{15}-1) ranging from 1 to about 2.1 billion. Having defined our data to be used in the defined array, the next step is to create a function that will take two parameters; the seed for random data generation and the required size of the array. The next step is to fill the index of each array size with random values by creating a "loop". This starts from one to the length of the array. Since the two frameworks were designed to be run on machines with different numbers of processing cores, the same seed will be used to generate random values. This will enable us to have the same starting point and the same values when the two frameworks are run on different machines.

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. System.currentTimeMillis() returns a difference in the measured time between the current time and mid-night of January 1970 UTC while System.nanoTime() will return a difference between the start and the end time in nanoseconds relative to an arbitrary fixed point. This paper focused on testing the efficiency of concurrency frameworks,

systems.nanoTime() method would be a better choice because System.currentTimemillis() has some weaknesses as it depends on the computer system clock which is not very perfect.

Concurrency Frameworks

The two frameworks namely: ServExecSort and NaïveParallelSort are implemented in this section. The two frameworks were used to measure the running times of quick-sorting algorithms to determine their performances.

ServExecSort Framework Implementation

This framework works by creating ThreadPool for a fixed number of threads. The created threads are then submitted into this pool. Each thread is then assigned many tasks to be executed in parallel and is being tracked using a 'Future services'. When the entire tasks are completely executed and the overall result is obtained, the ThreadPool is shutdown. The framework performance analysis is as follows:

Performance Analysis of ServExecSort Framework

Table 2: Running Time of ServExecSort

Array Size	Running Times (ms)	Array Size	Running Times (ms)
1000	0.727	300,000	25.648
4000	0.819	400,000	33.723
6000	0.968	500,000	43.343
8000	1.193	600,000	51.800
10000	1.408	700,000	60.832
20,0000	2.591	800,000	71.750
30,0000	25.354	900,000	85.728
40,0000	5.281	1,000,000	91.383
50,000	6.484	2,000,000	255.5
60,000	5.420	3,000,000	205.609
70,000	6.050	4,000,000	315.127
80,0000	7.705	5,000,000	433.124
90,000	8.822	6,000,000	754.483
100,000	9.809	7,000,000	827.990
200,000	17.648	8,000,000	827.990
		9,000,000	1014.497

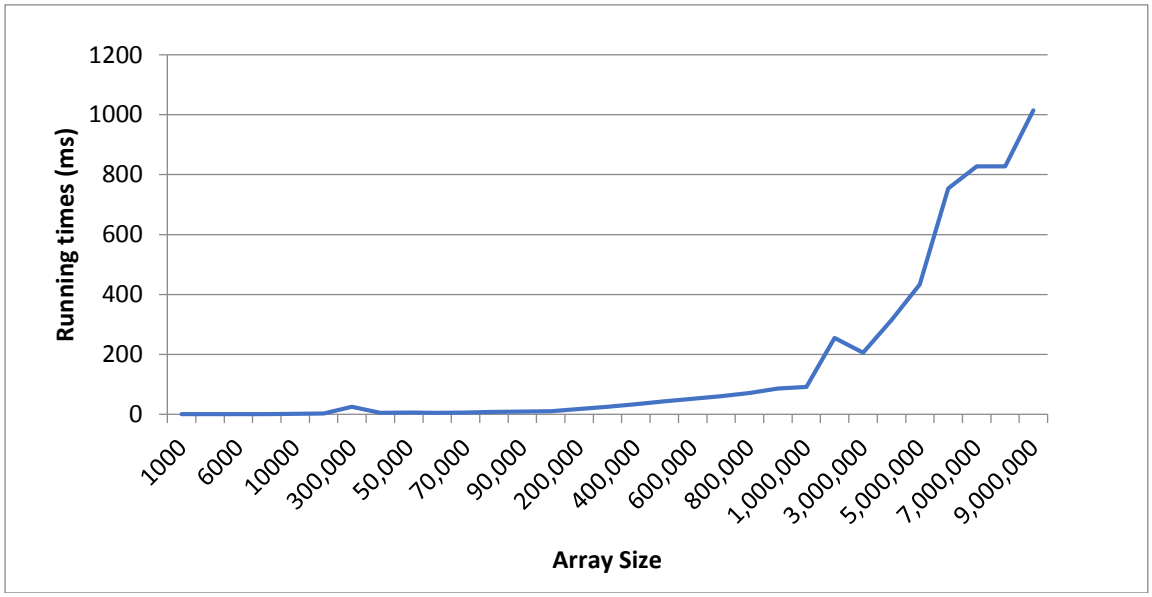

Figure 1: Showing the Running Times of ServExecSort

From Table 2 and Figure 1 above, it can be observed that the running times of quick-sort measured using ServExecSort framework increases with the increase in the size of the array. There was a sudden increase in the running time when the array size reached 90,000 with the corresponding running times of 8.822ms. The running times slowly increase until the array size reached about1000,000 elements with corresponding running times of 91.383ms. There was a significant increase in the running time when the array size increased from 1000,000 to 9,000,000 elements with the corresponding running times of 91.383ms and 1014.497ms respectively.

NaïveParallelSort Framework Implementation

On the other hand, NaïveParallelSort works by creating threads for each submitted task that needs to be executed during each recursive call. Each core on the machine is then assigned the available threads to do the computation in parallel. The results generated by each processing core are then combined to get the final result. This leads to performance improvement. The performance analysis of NaïveParallelSort is as follows:

Performance Analysis of NaïveParallelSort Framework

Table 3: Running times of NaïveParallelSort

Array Size	Running Times (ms)	Array Size	Running Times (ms)
1000	0.675	400,000	44.23
4000	0.758	500,000	54.50
6000	0.916	600,000	64.25
8000	0.916	700,000	77.57
10000	1.153	800,000	97.43
20,0000	2.561	900,000	109.2
30,0000	31.09	1,000,000	137.9
40,0000	5.261	2,000,000	255.5
50,000	6.484	3,000,000	393.9
60,000	5.500	4,000,000	533.2
70,000	6.572	5,000,000	738.7
80,0000	8.505	6,000,000	973.2
90,000	9.662	7,000,000	1194
100,000	10.64	8,000,000	1194
200,000	20.56	9,000,000	1271
300,000	31.80		

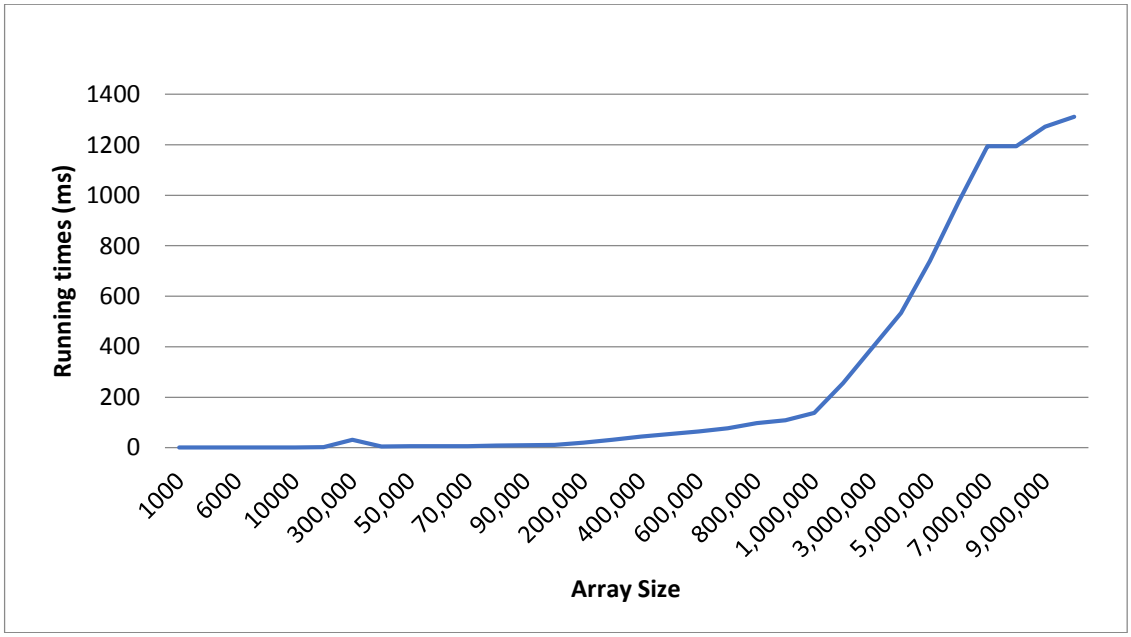

Figure 2: Showing the Running times of NaiveParrallelSort

Similarly, Form Table 3 and Figure 2, it can be seen that the running times of quick-sort measured using the NaiveParrallelSort framework also increases with the increase in the size of the array elements. From 1,000 to 300,000 elements the running time slowly rises. The running times start to increase significantly when the array size reaches 200,000 elements. The rapid growth continues to be recorded until the size of the array gets to 9,000,000 elements with corresponding running times of 1271ms.

Comparing the performance of ServExecSort and NaiveParrallelSort Frameworks

Table 4: Running Times of NaiveParallelSort and ServExecSort

Array Size	Running times of NaiveParallelSort (ms)	Running times ServExecSort (ms)	Array Size	Running times of NaiveParallelSort (ms)	Running times ServExecSort (ms)
1000	0.675	0.727	400,000	44.23	33.723
4000	0.758	0.819	500,000	54.5	43.343
6000	0.916	0.968	600,000	64.25	51.8
8000	0.916	1.193	700,000	77.57	60.832
10000	1.153	1.408	800,000	97.43	71.75
200,000	2.561	2.591	900,000	109.2	85.728
300,000	31.09	25.354	1,000,000	137.9	91.383
400,000	5.261	5.281	2,000,000	255.5	255.5
50,000	6.484	6.484	3,000,000	393.9	205.609
60,000	5.5	5.42	4,000,000	533.2	315.127
70,000	6.572	6.05	5,000,000	738.7	433.124
800,000	8.505	7.705	6,000,000	973.2	754.483
90,000	9.662	8.822	7,000,000	1194	827.99
100,000	10.64	9.809	8,000,000	1194	827.99
200,000	20.56	17.648	9,000,000	1271	1014.497
300,000	31.8	25.648			

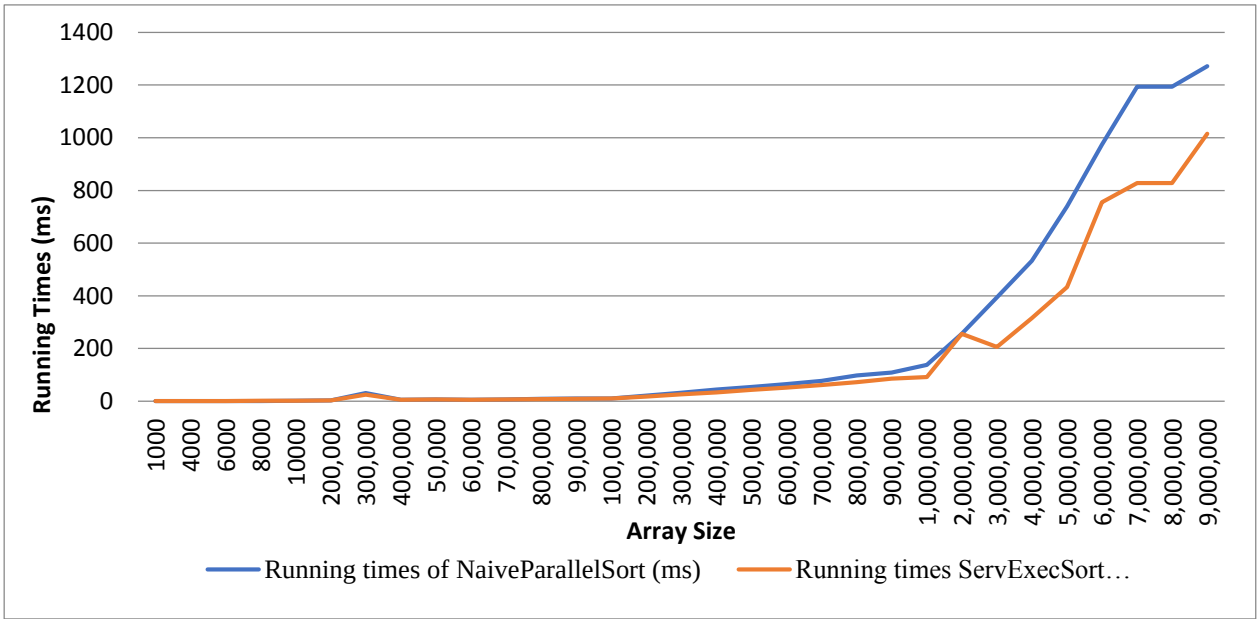

Figure 3: Depicting the Running Times of NaiveParallelSort and ServExecSort

Discussion of Results

From Table 4 and Figure 3, it can be seen that the running times of quick-sort measured using both the two frameworks increases with the increase in the size of the array. On the other hand, performance gain using the two frameworks differs. This can be seen from the running times of the two implementations obtained at a different point in time. For example, when sorting the smallest array elements, say 1000, the running times obtained using NaiveParallelSort and ServExecSort were 0.675ms and 0.727ms respectively. Therefore, the NaiveParallelSort framework exhibits better performance by recording less running times. Hence, NaiveParallelSort was found to be 0.052ms faster than ServExecSort when the smallest array was sorted. NaiveParallelSort continues to exhibit better performances until the array reaches 300,000 elements. At this point, the running time recorded by both NaiveParallelSort and ServExecSort was found to be 31.09ms and 25.354ms respectively. Thus, that makes the NaiveParallelSort framework to be 5.736ms slower than the ServExecSort framework. The good performance benefit of the ServExecSort framework continues to become more noticeable when the array size increases from 400,000 to 9,000,000 elements. At the end of the sorting process, the running times recorded by both NaiveParallelSort and ServExecSort frameworks were found to be 1271ms and 1041.497ms respectively. The difference between the two running times was 229.503ms. This shows that the NaiveParallelSort framework is almost 230ms slower than the Service executor framework when sorting the largest array elements.

Conclusions

This paper compares the performance of two concurrency frameworks. The two frameworks were used to sort array elements containing 1,000 to 9,000,000 sizes and the running times were measured and compared. From the results obtained, it can be concluded that the performance of the two frameworks was interwoven. NaiveParallelSort framework exhibits better performance with small array sizes ranging from 1,000 to 300,000 sizes while the ServExecSort framework performs better on larger array elements ranging from 300,000 to 9,000,000 sizes. It can be concluded that the performance improvements recorded by the ServExecSort framework are because the ThreadPool executor improves its performances by creating and terminating threads automatically without any effort from the developer's side. It is further concluded that the poor performance recorded by the NaiveParallelSort framework on the larger array is because creation and termination of threads are mainly done by the programmer and that excessively limit fractioning of the framework.

Recommendations

The two concurrency frameworks presented in this paper can be further designed, experimented and tested on other popular algorithms such as merge sort, selection sort, binary search and linear search on machines with different number of processors/ processing core to see if the same results can be obtained by using similar approach. The authors of this paper recommend that this deserves further research.

References

Aleksandar, V. (2014). Manual parallelization versus state-of-the-art parallelization techniques. Netherlands: Elsevier. DOI: https://doi.org/10.1016/B978-0-12-420232-0.09990-6, 203-251.

Brad, L., Paul, S., & Luke Wildman. (2007). A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience 19*(3), 281-294.

Brad, L. (2009). Framework for model checking concurrent Java components. *Journal of Software 4*(8), 867- 874.

Clay, B. (2009). The art of concurrency (3rd Ed.). Graham: O'Reilly Media. Retrieved from: https://www.amazon.com/Art-Concurrency-Monkeys-Parallel Applications/dp/0596521537, 20-43.

Doug, L. (2015). The java.util.concurrent synchronizer framework. *Science of Computer Programming 58*(3), 293-309.

Gambo, F.L., Wajiga, G.M., Garba, E.J. & Abdullahi, A.A. (2020). Prediction of learning style using facial expressions with convolution neural network. *International Journal of Information Processing and Communication (IJIPC) 9*(1&2), 389-399.

Intel Corp Desktop Products Group (2009). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 2*(3), 4-17.

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

Hongwei, L., Yin, W., & Hyoun, K. (2013). Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Discrete Event Dynamic Systems: Theory and Applications 23*(2), 157-195.

Jeremy, S. Bradbury, James, R. C., & Juergen, D. (2018). Mutation operators for concurrent Java (J2SE 5.0). *School of Computing, Queen's University Kingston, Ontario, Canada*. Retrieved from: http://research.cs.queensu.ca/home/cordy/Papers/BCD_ConcOps_Mutation06.pdf, 1-10.

Kaya, D. (2015). 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.

Mansoor, N., & Alarcon, S.L. (2017). *Advances in GPU research and practice*. Retrievedfrom: https://www.sciencedirect.com/topics/computer-science/multicore.

Michael, D., Linderman, J. D., Collins, H. W. & Teresa, H. M. (2018) Merge A programming model for heterogeneous multi-core systems. *SIGPLAN* 43-287.

Pavel, G. Z. & Georgi, K. (2012). Multithreading on configurable hardware: An architectural approach. *Microprocessors and Microsystems 36*(8), 695-704.

Qiming, H., Kun, Z., & Baining, G. (2018). BGP: Bulk synchronous GPU programming. *ACM Trans. Graph., 27*(3), 12-29.

Rabiu, A.M., Garba, E.J., Baha, Y. B. & Mukhtar, M.I. (2019). Comparative analysis between merge and selection sorts algorithms. *Paper Presented at the 6th Big Data Analytics & Innovation Conference Held at Baze University Abuja, Nigeria.*

Rabiu, A.M., Garko, A.B. & Abdullahi, A.M. (2018). Effects of multi-core processors onlinear and binary sorting algorithms. *Dutse Journal of Pure and Applied Sciences 4*(2). Available at http://fud.edu.ng/journals.php, 130-140.

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 4*(2). Available at http://fud.edu.ng/journals.php, 254-263.

Rabiu, A.M., Ibrahim, A., M., Dauda, Mukhtar, M.I., Abdullahi, A. M & Abdul'azeez, Y. (2019). Adoption of E-Commerce in Nigeria challenges and future prospects. *2019 2nd International Conference of Institute of Electrical Electronics (IEEE) Nigeria Computer Chapter (NigeriaComptConf), Zaria, Nigeria, 1-8.* DOI: 10.1109/NigeriaComputConf45974.2019.8949641

Rabiu, A.M., Mukhtar, M.I., Abbas, F.M. & Abdul'azeez, Y. (2019). ICT utilization and its barriers in Jigawa State primary health care centers. *Information Technologist (The): International Journal of Information and Communication Technology 16*(2), 141-156. Retrieved from: https://www.ajol.info/index.php/ict/issue/view/18700

Abdullahi, A.M, Orji, R., Rabiu, A.M. & Kawu, A.M. (2020). Personality and subjective well-being: Towards personalized persuasive intervention for health and well-being. *Online Journal of Public Health Informatics, 12*(1), 1-24. Retrieved from: https://doi.org/10.5210/ojphi.v12i1.10335

Robert, L. (2017). *Data structures and algorithms in java* (3rd Ed). Retrieved from: http://www.resaechgate.net.

Wuxue, J., Qi, L., Zhiming, W. & Jianfeng, L. (2013). A framework of concurrent mechanism based on Java multithread. *TELKOMNIKA,11*(9), 5395-5401.

Yaniv, E. (2016). Concurrent Java test generation as a search problem. *Electronic Notes in Theoretical Computer Science 144*(4), 57-72.

Zhe, F., Feng, Q. & Arie, E. K. (2018). Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum, 27*(2), 341-350.