ISSN (Print): 2476-8316
ISSN (Online): 2635-3490
Dutse Journal of Pure and Applied Sciences (DUJOPAS), *Vol. 4 No. 2 December 2018*

Performance Evaluation of Three Quick sorting Algorithms on a Single and Multi-core Processors

***Abbas Muhammad Rabiu[1], Ahmed Baita Garko[2], Aisha Muhammad Abdullahi[3], Hadiza Ali Umar[4], Mansur Babagana[5]**
[1,2,3] Department of Computer Science, Federal University, Dutse (FUD)
[4,5] Department of Computer Science Bayero University, Kano (BUK)
Email: mambas86@yahoo.com

Abstract
Multi-core processor is an improvement over the Single-core processor architecture and represents the latest developments in microprocessor technology. Performance evaluation of algorithms can be carried out on a computer with different number of processing cores with a view to establish the level of their performances. This research paper carried out performance evaluation studies on three different Sorting algorithms namely: Quick Sort-Sequential, Quick Sort-Parallel Naive, and Quick Sort-Fork Join on a Single-core and Multi-core processors to determine which of the algorithms has better execution time, and to show the effect of Multi-processor machines on their performances. System.nano time() benchmark suits was used to measure the performances of these three algorithms. The overall running time of these algorithms were reported and compared. Results showed that the running time of Quick Sort-Fork Join is about 46.62% faster than Quick Sort-Sequential, and 31.20% faster than Quick Sort-Parallel Naive. Therefore, Quick Sort-Fork Join algorithm exhibits better performance probably due to its divide and conquer approach. Work stealing action is also another reason for this good performance when measured on both single and Multi-core processor. It was also discovered that increase in a number of processing cores in a machine significantly improves the performances of these algorithms.

Keywords: Algorithm, Quick sort, Processor, Running time, Performance.

INTRODUCTION
Sorting is the rearrangement of items in a list into their correct lexicographic order, while algorithm is a step by step procedure of achieving solution for a desired result. A number of sorting algorithms have been developed like quick sort, heap sort, merge sort and selection sort all of which are comparison based sort (Ahmed and Zirra, 2013). There are other classes of sorting algorithms which are non-comparison based sort. It was discovered that the running time increases with increase in the array size. Quick sort is another comparison based sorting algorithm which is better than selection sort and works in a gap sequence (Robert, 2002). The running time and the performance analysis carried out by Thomas, (2004) shows that although its performance is better than the selection sort, there are other various factors that need to be kept in mind.

When we say that one computer is faster than another, what we are actually saying is that a program runs in less time on one computer than the other. Most Computer users are generally interested in reducing response time or execution time and increase in the

**Author for Correspondence*

throughput (the total amount of work done in a given time). In comparing design alternatives, it is often the practice to relate the performances of two different computers, say, X and Y. When computer X is faster than computer Y, this means that the response time or execution time is lower on X than on Y for the given task. Then, computer X is n times faster than computer Y (John and David, 2012).

Therefore, $execution\ time\ of\ Y / Execution\ time\ of\ X = n$ ---------- (1.1)

Since execution time is the reciprocal of performance, therefore the following relationship holds.

$$n = \frac{execution\ time\ of\ Y}{\text{execution time of X}} \ldots\ldots\ldots\ldots\ldots (1.2)$$

taking the reciprocal of equation (1.2) shows that:

$$\text{n}= \frac{execution\ time\ of\ X}{\text{execution time of Y}} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1.3)$$

The phrase "the throughput of X is 1.3 times higher than Y" signifies here that the number of tasks completed per unit time on computer X is 1.3 times the number completed on Y. The only consistent and reliable measure of performance is the execution time of real programs (Patterson and Hennessy, 2007)

STATEMENT OF THE PROBLEM

Sorting is a problem that mostly arises in computer science. Many Sorting algorithms have been developed while existing ones have been improved upon all to make them more efficient. According to Sengupta *et al*. (2007) efficiency of algorithms can be measured in terms of execution time (complexity) and amount of memory required. The improvement in performance gain by the use of multi-core processors depends very much on the nature of algorithms used and their implementations (Cormen *et al*., 2004). Therefore programmers are faced with the challenges of choosing which implementations of the algorithms and also which concurrency tools will give best result when developing and testing their concurrent software as some are faster than others in terms of execution time. Therefore, this paper analyzes these two situations using three different sorting algorithms and some concurrency mechanisms provided by java to measure the performances of all the three algorithms on both single and multi-core machines. By sorting a variety of array size ranging from 1000 to 10,000,000 using the three algorithms we can see how well they perform against each other.

RELATED WORK

Quick-sort executes in$O(nlogn)$ on the average, and $O(n^2)$ in the worst-case (Santorn et al, 2007). However, with proper precautions, worst-case behavior is very unlikely. Quicksort is a non-stable sort. It is not an in-place sort as stack space is required. Additionally, Quicksort being such a popular sorting algorithm, there have been a lot of different attempts to create an efficient parallelization of it. The obvious way is to take advantage of its inherent parallelism by just assigning a new processor to each new subsequence. This means, however, that there will be very little parallelization in the beginning, when the sequences are few and large (Macllory *et al*., 1996). Another approach by Philippas (2003) has been to divide each sequence to be sorted into blocks that can then be dynamically assigned to available processors. However, this method requires extensive use of atomic FAA2 which makes it too expensive to use on graphics processors. Moreover, Blelloch (1993) suggested using pre_x sums to implement Quicksort. Similarly, Sengupta *et al* (2007) used this method to make an implementation for Computer Unified Device Architecture (CUDA). The

implementation was done as a demonstration of their segmented scan primitive, but it performed quite poorly and was an order of magnitude slower than their radix-sort implementation in the same paper. (Santorn *et al*., 2007) have presented a radix-sort and quicksort implementation using another approach with insertion sort, thus making the sorting process less efficient. Additionally, (Santorn *et al*., 2007) presented a hybrid sorting algorithm, a modified version of the Radix-sort which splits the data with a bucket sort and then uses merge sort on the resulting blocks thus, making the process more efficient. This paper however uses slightly different approach by taking advantage of multi-core processors to create a number of threads and assigning them to available processing cores in a machine with different number of processing cores to measure the running time of three Quick sorting algorithms namely: Quick Sort-Sequential, Quick Sort-Parallel Naive, and Quick Sort-Fork Join on a Single-core and Dual-core processor machine to determine which of the algorithms has the best execution time, and to determine the effect of multi-processor machines on the performances of these algorithms.

METHODOLOGY

To benchmark the algorithms in this paper, the main method used is the practical measurement of run time. In Java there are currently two built-in functions which let the user retrieve time: System.current Time Millis() and System.nano Time(). System.current Time Millis() is based on computer system's clocks which has some weaknesses. The system clock is not perfect, it may drift off sometimes and occasionally needs to be corrected. Because of this weakness associated with Current Time Millis, System.nano Time() is used as the main method to measure time in this paper. System.nano Time() gives more precise results but it is actually slower than System.current Time Millis(). When this function is called it returns a number in nanoseconds from a fixed but arbitrarily chosen point. This time is then converted to milliseconds and compared for convenience.

TEST DATA STRUCTURE

It is a good idea when benchmarking an algorithm, to do more than one test runs on each dataset. This is mainly because other background processes on the computer may interrupt ongoing computations and may cause a relative huge impact on measurements, especially when sorting smaller dataset.

The structure in Figure 3.1contains array-sizes ranging from 1000 to 10 million elements and also included the number of times each array is sorted to get a more precise result. The array-sizes are defined using Powers of Ten incremented by $\frac{1}{4}$. ($10n * \frac{1}{4}$), 12<n<32.

```
static int[][]  dataArray = new int[][] {
/*{Array-size, N runs} */
   {      1 000, 2 500}, {      1 778, 2 500}
   {      5 623, 2 500}, {     10 000, 2 500}
   {     31 622, 1 250}, {     56 234, 1 250}
   {    177 827,   500}, {    316 227,   250}
   { 1 000 000,   100}, { 1 778 279,    75}
   { 5 623 413,    50}, {10 000 000,    40}
   {31 622 776,    16}, {56 234 132,     8}
};
```

Figure 3.1: Structure of the test data

PERFORMANCE ANALYSIS OF QUICK SORT

Quick sort is a comparison sort developed by Tony Hoare. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set. Quick sort algorithm according to Ahmed and Zirra (2013) can be depicted as follows:

1: step←m;
2: while step > 0
3: for (i←0 to n with increment 1)
4: do temp←0;
5: do j←i;
6: for (k←j+step to n with increment step)
7: do temp←A[k];
8: do j←k-step;
9: while (j>=0 && A[j]>temp)
10: do A[j+step]=A[j];
11: do j←j-step;
12: do Array[j]+step←temp; 13: do step←step/2;

Run Time of Quick Sort Best case is O ($nlogn$) Average case is O ($nlogn$)and the worst case is $O(n^2)$ The main focus of the Quick sort lies in line 6. Line 6-12 is the algorithm for insertion sort. But insertion sort is performed between some specified gaps. Since the sorted order of elements gets increased with the completion of each step, these lines gives a constant value less than the value of n. Thus, in the best case of this is almost O(n). But there may be the worst case condition for large number of inputs when computational complexity increases and the gap insertion sort can give almost n steps resulting into running time of $O(n^2)$ (Ahmed and Zirra, 2013).

PERFORMACE ANALYSIS OF QUICK SORT SEQUENTIAL

A sequential algorithm has been implemented using inspiration from the book 'Algorithms' by Robert and Wayne (2011), which uses Radix-sort for string sorting. The algorithm does the sorting in what can be defined as five steps:

1. Start of by finding the maximum value in the whole array; and finding how many bits this value consist of in the binary numeral system.
2. Count how many elements it is of each value in the array.
3. Add up values in count, accumulating the values.
4. Distribute the values sorted to temp.
5. Copy back the elements from the temporary array to the original.

Table 3.3 and **Figure 3.3** show the results obtained when sorting different array size ranging from one thousand to ten million sizes. It can be clearly seen that there has been a significant improvement on the performance of this algorithms when measured on a Dual-core as compared with the results obtained earlier on Single-core machine. Generally speaking, the running time continue to decrease throughout the sorting process. At the beginning of the sorting process with the 1000 array size, 0.086ms and 0.074ms were recorded on both Single and Dual-core machine respectively.Thus,0.012ms decrease in the running time has been achieved. At the end of the sorting process when sorting 10,000,000 elements, running time of 2,399ms and 1,270ms were recorded on both Single and Dual-core machine respectively. Therefore, 1,129ms decrease in the running time has been achieved. This is considered to be a very good significant improvement on the running time.

Table 3.3 Running times of Quick Sort Sequential on Single and Dual-core machine

Array Size	Quick Sort Sequential (Single-core) Running time (ms)	Quick Sor tSequential (Dual-core)Running time (ms)
1000	0.086	0.074
31622	4.227	3.526
100000	16.27	12.60
177827	31.25	24.12
316227	56.16	44.18
562341	104.7	81.84
1000000	197.2	105.3
3162277	681.3	389.3
5623413	1212	672.1
10000000	2399	1270

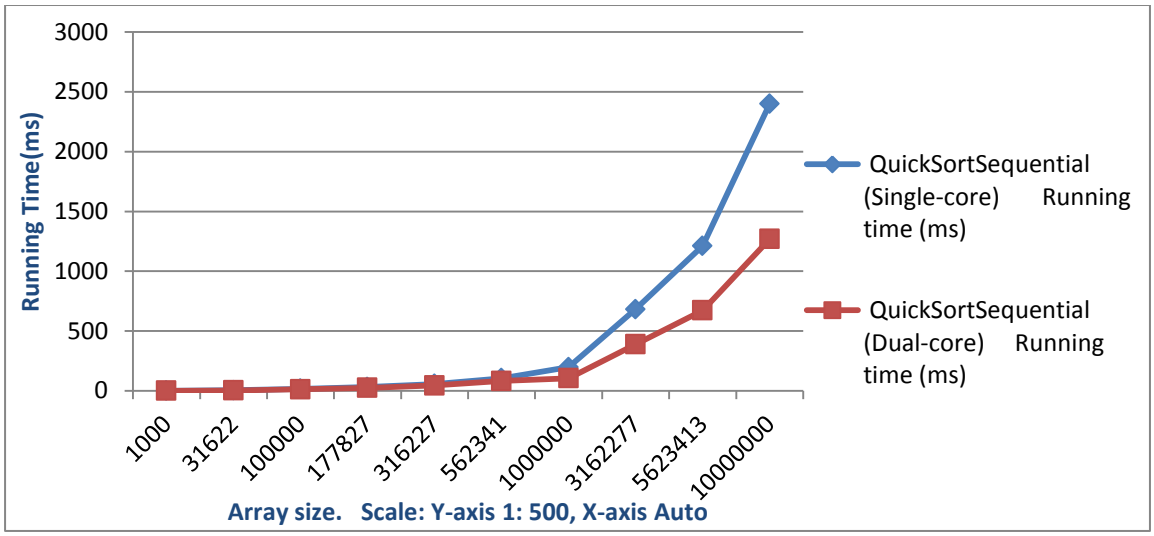

Figure 3.3: Running time of Quick Sort Sequential on Single and Dual-core machines.

PERFORMANCE ANALYSIS OF QUICKSORT-PARALLELNAIVE:

The naive attempt differs from the sequential Quick-sort in a way that each time Quick-sort recursively calls itself; it instead creates new threads which are then called. And by creating these new threads, each core (included Hyper-Threads) on the current system can then be assigned the available threads and do the computation in parallel, which then should result in increasing the overall performance.

Table 3.4 and **Figure 3.4** show the results obtained when sorting different array size ranging from one thousand to ten million sizes. It can be clearly seen from the results that there has been a significant improvement on the performance of this algorithm when measured on a Dual-core as compared with the results obtained earlier on Single-core machine. Similarly, the running time continues to decrease throughout the sorting process. At the beginning of the sorting process with the 1000 array size, 20.30ms and 0.344ms were recorded on both Single and Dual-core machine respectively. Thus, 19.956ms decrease in the running time has been achieved. This is a very significant improvement on the running time compared with one obtained earlier with Quick-sort Sequential on the same array size. This is probability due to the parallel nature of the naïve. At the end of the sorting process when the array elements increased to 10,000,000 sizes, running time of 2,920ms and 1,270ms were recorded on both Single and Dual-core machines respectively. Therefore, 1650ms decrease in the running time has been obtained. Unlike Quick-sort Sequential, in Parallel Naive, There has been significant improvement on the running time right from the beginning to end of the sorting process.

Table 3.4: Running times of Quick Sort Parallel Naïve on Single and Dual-core machines

Array Size	Quick Sort Parallel Naive (Single-core) Running time(ms)	Quick Sort Parallel Naïve (Dual-core) Running time(ms)
1000	20.30	0.344
31622	4.965	3.876
100000	20.73	10.89
177827	40.43	19.16
316227	70.85	32.77
562341	128	57.60
1000000	240.6	105.3
3162277	806.2	389.3
5623413	1480	672.1
10000000	2920	1270

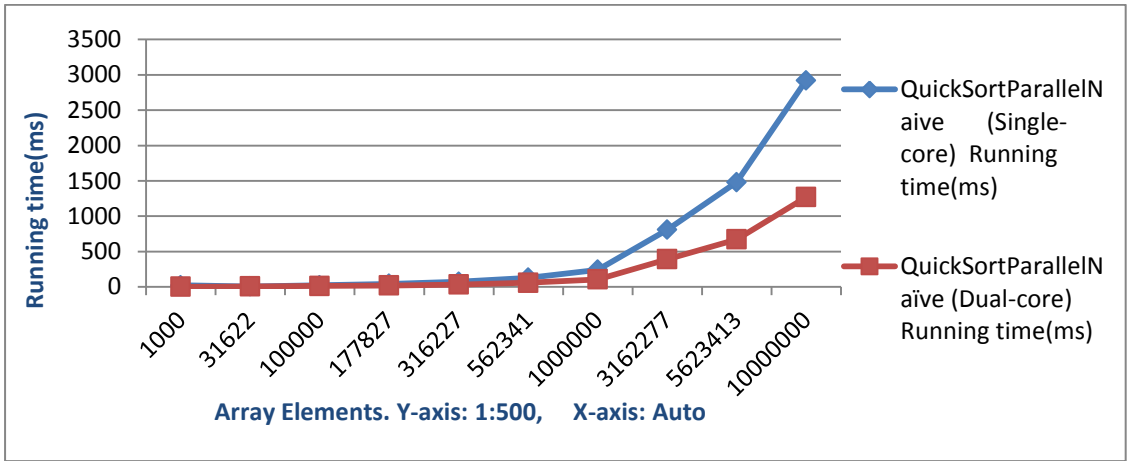

Figure 3.4: Running time of Quick Sort Parallel Naïve on Single and Dual-core machines.

PERFORMANCE ANALYSIS OF QUICK SORT - FORK JOIN:
Unlike Parallel Naive, In Quick Sort - Fork Join, a fixed pool is created to contain the available worker threads, and then submit tasks to this pool. It then uses invoke on tasks, this performs the given task and returning its result upon completion. This means that there is no need to use Future to keep track of the tasks. Creating the Fork/Join-framework only requires specifying the number of desired threads for the pool.

Table 3.5 and **Figure 3.5**show the results obtained when sorting different array size ranging from one thousand to ten million sizes using Quick Sort – Fork Join frame work. It is noticeable from the results that there has been a significant improvement on the performance of this algorithm when measured on a Dual-core as compared with the results obtained earlier on Single-core machine. Similarly, the running time continues to decrease throughout the sorting process. At the beginning of the sorting process with the 1000 array size, 0.106ms and 0.104ms were recorded on both Single and Dual-core machine respectively. Thus, 0.002ms decrease in the running time has been achieved. There is no significant difference between the running times obtained on both the two machines. This is due to the divide and conquers ability of this algorithm. At the end of the sorting process when the array elements increased to 10,000,000 sizes, running time of 2,495ms and 1,008ms were recorded on both Single and Dual-core machines respectively. Therefore, 1487ms decrease in the running time has been obtained. Unlike in Quick-sort Sequential and Parallel Naive, here in Fork Join, the running time has been significantly improved even from the beginning of the sorting process but the difference in the running time becomes more noticeable when the array elements increases to 5,000,000on a Dual-core machine.

Table 3.5 Running time of Quick Sort – Fork Join on both Single and Dual-core machines.

Array Size	Quick Sort Fork/Join (Single-core) Running time(ms)	Fork/Join (Dual-core) Run time (ms)
1000	0.106	0.104
31622	4.392	3.583
100000	16.20	8.934
177827	35.43	15.36
316227	57.84	26.687
562341	108.2	46.16
1000000	204.6	84.95
3162277	699.9	307.8
5623413	1248	543.6
10000000	2495	1008

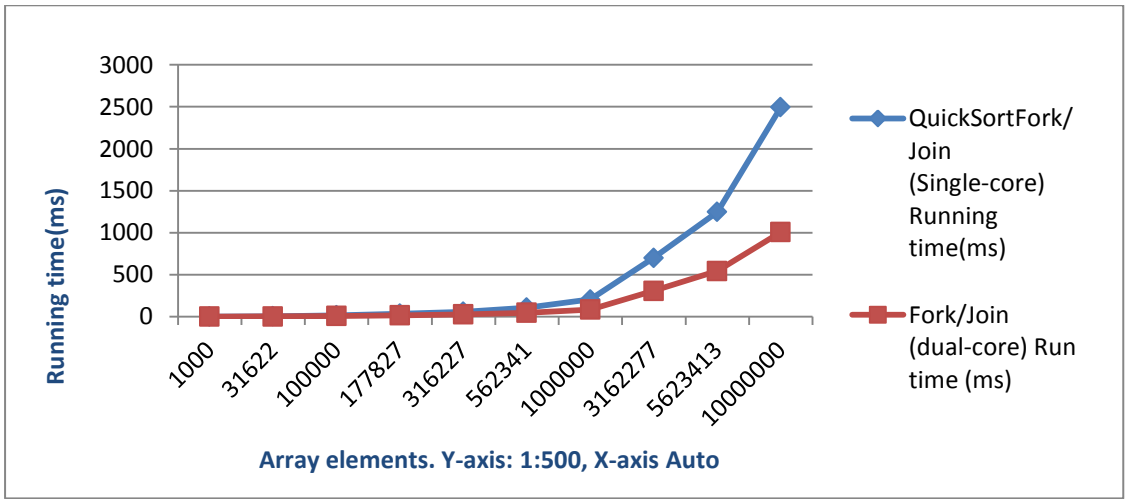

Figure 3.5: Running time of Quick Sort - Fork Join on both Single and Dual-core machines

HARDWARE AND SOFTWARE SPECIFICATIONS

For the development and benchmarking of these three Quick sorting algorithms used in this paper, the hardware specifications shown below in Table 3.6, Table 3.7were used.

Table 3.6: Single-Core Processor Specification

CPU	Mobile AMD Sempron™ Single Core Processor 3200+1.60GHz
RAM	1.50 GB
OS	Windows 7 64-bit

Table 3.7: Dual-Core-Core Processor Specification

CPU	Intel(R) Pentium® Dual Core CPU T2390 @ 1.86GHz, 1.87GHz
RAM	1.00 GB
OS	Windows 7 64-bit

PERFORMANCE COMPARISON OF ALL THE THREE ALGORITHMS ON BOTH SINGLE-CORE AND DUAL-CORE MACHINES.

Table 4.1: Running time of all the three Sorting algorithms on a single-core machine

Array Size	Quick Sort Sequential (Single-core) Running time(ms)	Quick Sort Parallel Naive (Single-core) Running time(ms)	Quick Sort Fork/Join (Single-core) Running time(ms)
1000	0.086	20.30	0.106
31622	4.227	4.965	4.392
100000	16.27	20.73	16.2
177827	31.25	40.43	35.43
316227	56.16	70.85	57.84
562341	104.7	128	108.2
1000000	197.2	240.6	204.6
3162277	681.3	806.2	699.9
5623413	1212	1480	1248
10000000	2399	2920	2495

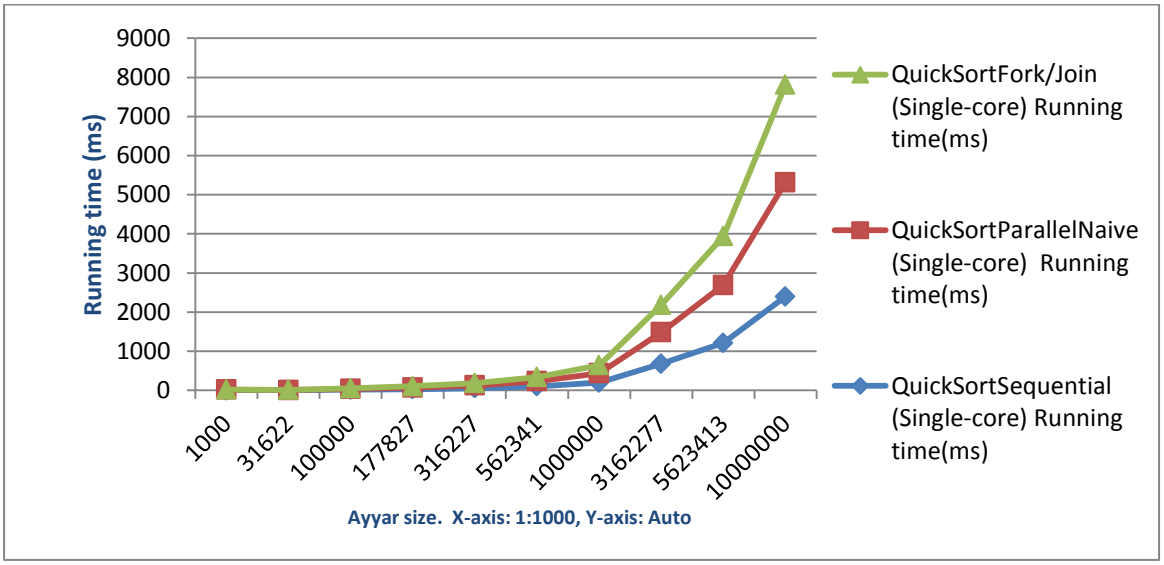

Figure 4.1 Running Times of all three algorithms on Single-core machine

Table 4.2: Running times of all the three Sorting algorithms on a dual-core machine

Array Size	Quick Sort Sequential Run time(ms)	Quick Sort Parallel Naïve Run time(ms)	Fork/Join Run time (ms)
1000	0.074	0.344	0.104
31622	3.526	3.876	3.583
100000	12.6	10.89	8.934
177827	24.12	19.16	15.36
316227	44.18	32.77	26.687
5623413	81.84	57.6	46.16
1000000	151.9	105.3	84.95
3162277	537.4	389.3	307.8
562341	977.2	672.1	543.6
10000000	1808	1270	1008

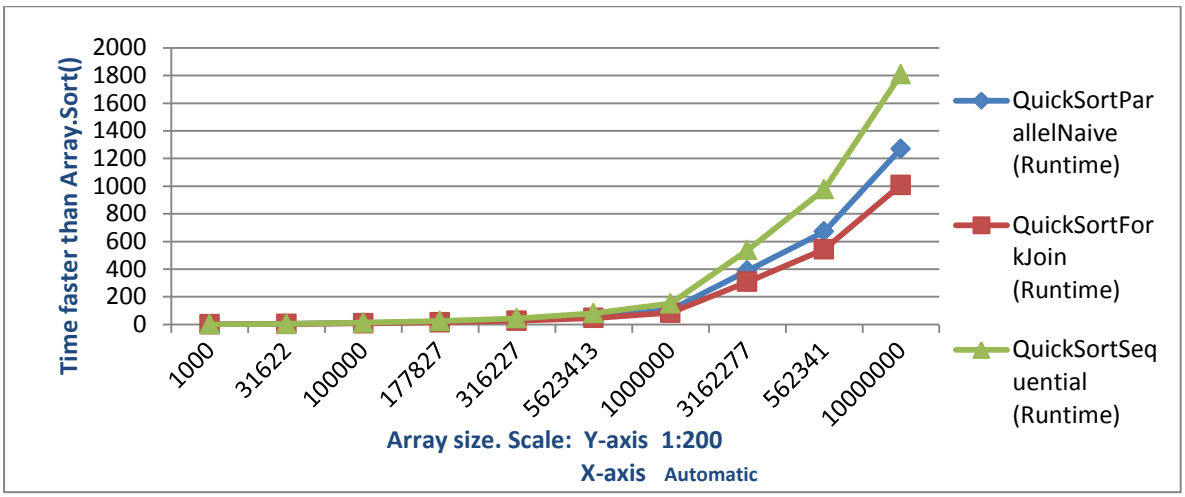

Figure 4.2: Running time of all the three sorting algorithms on a dual-core machine.

DISCUSSION OF RESULTS

From the analysis of the algorithms and the results obtained in Table 3.1, Figure 3.1, Table 3.2 and Figure 3.2, Table 3.3 and Figure 3.3e.t.c. It has been shown that Quicksort Sequential performed poorly with larger array elements as compared with the two other algorithms. However, its performance has been improved with the increase in the number of processing cores. Similarly, Quick sort Parallel Naive, has better performance than Quicksort Sequential right from the begging to the end of the sorting processes, probably due to its parallel nature and threads calls ability. Quick sort Fork/Join has the best performances on both the two machines probably due to its work stealing actions and divides and conquers approach. The overall results showed that running time of Quick Sort – Fork Join is about 46.62% faster than that of Quick Sort - Sequential, and 31.20% faster than Quick Sort – Parallel Naive. Therefore, Quick Sort – Fork Join algorithm exhibits better performance probably due to its divide and conquer approach. Work stealing action is also another reason for this good performance when measured on both single and Multi-core machines.

CONCLUSION

This paper discusses three comparison based sorting algorithms. It analyses the performance of these algorithms for the same number of elements on two different machines. It then concludes that Quick sort Fork/Join has better performances when sorting large number of array elements than Quick sort Parallel Naive and Quick sort Sequential. However, Quick sort Sequential performs better than Quick sort Parallel Naive and Quick sort Fork/Join on smaller array size on both single and Dual-core machines. It can be concluded however, that Quick sort Fork/Join is the best sorting algorithms than Quick sort Sequential and Quick sort Parallel naïve when sorting larger array elements on both the two machines. It is also concluded that the increase in a number of processing cores in a machine significantly improves on the performance of these algorithms.

REFERENCES

Ahmed, M. and Zirra, P.(2013). A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays. *International Journal of Engineering and Science* Vol. 2, PP. 25-30.

Ananth, G., Anshul G., George K. &Vipin K.(2007).Introduction to Parallel Computing.2nd Ed.,Addison-Wesley.pp.123-145.

Guy, E. and Blelloch (1993). *Pre_x Sums and Their Applications*.3rd Ed., New York. Addison-Wesley.pp.45-56.

John, H. and David, P. (2012). *Computer Architecture: A Quantitative Approach* [pdf], [online], UK: Elsevier. Available at: https://www.amazon.com/Computer-Architecture-Quantitative-John-Hennessy/dp/012383872X. [Accessed on 5th October 2016].

John, P. (2003). *A Simple, Fast and Parallel Implementation of Quicksort and its Performance Evaluation*. 3rded., [pdf]. New York: A. Wasley. Available at: http://www.ieeeexplorer.ieee.org.com/ [Accessed 1st June. 2017].

Knuth, D. (1997). *The art of computer programming, sorting and searching.* Third edition. New York, Addison Wasley, PP.395-409.

Macllory, H., Norton, A., & John, T. (1996). *Parallel Quicksort Using Fetch-And Add.* IEEE Trans. Comput., pp.133-138.

Philippas, and Zhang, Y. (2003). Evaluation on SUN Enterprise 10000.*Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*.pp.372-374

Patterson, D. and Hennessy J. (1996).*Computer Architecture: A Quantitative Approach*, 2nd Edition. San Francisco California, Morgan Kaufmann, USA.PP.32-78.

Robert, L. (2002). *Data Structures and Algorithms in Java*,[online], [2nd Ed. New York: A. Wasley. PP. 24-28. Available at: http://www.quora.com/ [Accessed 1st Feb. 2015].

Randal, E. Bryant, David R. and Hallaron, O. (2010).*Computer Systems*: A *Programmer's Perspective.*[online], [US. Wasley.PP.1-15. Available at: http://csapp.cs.cmu.edu/2e/ch4-preview.[pdf], [Accessed 5th October 2017].

Robert, S., and Kevin, W. (2011). *Algorithms*. [online] 4th edition. Princeton University. pp.23-67.Available at: http://www.cs.bu.edu [Accessed 1st Feb. 2014].

Santorinin, K. and Thira O. (2007). *Adapting Radix Sort to the Memory*.3rd ed. [pdf], [online], New York: A. Wasley Available at: http://www.quora.com/ [Accessed 1st Feb. 2015].

Sengupta, p. (2007). *Algorithms in Java, Parts 1-4*,3rd ed. Addison-Wesley New York: Professional Publisher.pp.36-76.