OPEN ACCESS
Indo-JC Indonesia Journal of Computing
ISSN 2460-9056
socj.telkomuniversity.ac.id/indojc

Ind. Journal on Computing
Vol. 5, Issue. 2, September 2020. pp. 81-90
doi:10.21108/indojc.2020.5.2.487

A Parallel Implementation of Dual-Pivot Quick Sort for Computers with Small Number of Processors

Mohammad F. J. Klaib [#1], Mutaz Rasmi Abu Sara [#2], Masud Hasan [#3]

 # Department of Computer Science, Taibah University, Madina Al Munawarah, Saudi Arabia
[1] mklaib@taibahu.edu.sa
[2] mabusara@taibahu.edu.sa
[3] hmasud@taibahu.edu.sa

Abstract

Sorting algorithms are heavily used by computers. Quicksort is one of the fastest comparison-based sorting algorithms. These days almost all computing devices have multiple processors. There is a strong need of finding efficient parallel versions of the most common algorithms that are widely used. The basic version of quick sort is sequential and uses only one pivot. Recently, Yaroslavsky has proposed a modified version of the quick sort that uses two pivots and runs much faster than the single-pivot quick sort. Since then Java has incorporated this dual-pivot quick sort into its standard library for sorting. Although there are many parallel versions of the original single-pivot quick sort, there is a very few for the dual-pivot. Those few parallel versions of the dual-pivot quick sorts are compared with standard sort functions, rather than the dual-pivot quick sort itself. In this paper, we provide a parallel version of the dual-pivot quick sort algorithm of Yaroslavsky and implement it in Java. For comparison, we run both in small number of parallel processors. The experimental results show that our algorithm runs significantly faster than the Yaroslavsky's algorithm. Moreover, our algorithm performs gradually better as the number of processors and the input size increase.

Keywords: Sorting, Quick sort, Dual-pivot quick sort, Parallel quick sort

Abstrak

Algoritma penyusunan banyak digunakan oleh komputer. Quicksort adalah salah satu algoritma penyusunan berasaskan perbandingan terpantas. Pada masa ini hampir semua peranti pengkomputeran mempunyai banyak pemproses. Terdapat keperluan mendesak untuk mencari versi selari yang berkesan daripada algoritma yang paling umum yang banyak digunakan. Versi asas algoritma penyusunan adalah berurutan dan hanya menggunakan satu pangsi. Baru-baru ini, Yaroslavsky telah mencadangkan algoritma penyusunan diubahsuai yang menggunakan dua pivot dan berlari lebih pantas daripada jenis cepat pivot tunggal. Sejak itu Java telah memasukkan dua pivot penyusunan cepat ke dalam perpustakaan standardnya untuk disusun. Walaupun terdapat banyak versi selari dari pivot tunggal tunggal yang cepat, terdapat sangat sedikit untuk pivot dua. Beberapa versi selari jenis dual-pivot quick dibandingkan dengan fungsi piawai susun, dan bukannya jenis quick-dual pivot itu sendiri. Dalam kertas kerja ini, kami menyediakan versi selari dari algoritma jenis cepat-pivot Yaroslavsky dan mengimplementasinya di Java. Sebagai perbandingan, kami menjalankan kedua-duanya dalam sebilangan kecil pemproses selari. Hasil eksperimen menunjukkan bahawa algoritma kami berjalan jauh lebih cepat daripada algoritma Yaroslavsky. Lebih-lebih lagi, algoritma kami berprestasi lebih baik secara beransur-ansur apabila bilangan pemproses dan saiz input meningkat.

Kata Kunci: Menyusun, Pivot-dua susun cepat, Jenis cepat dua pangsi, cepat selari

Received on Jan 2020. Accepted on Aug 2020

I. Introduction

We start with a very simple observation, which also works as a key motivation behind our work in this paper. These days, almost all computing devices such as PCs, laptops, mobile phones, and tablets have processors with multiple cores. As of today, the number of processors in the abovementioned devices are usually within two to sixteen.

As the number of processors increases in a computer, its performance also increases. A program runs faster on a multi-core processor than on a single-core processor. The advantages of multi-core processors are utilized by an operating system when it automatically schedules the tasks in multi-core processors.

In addition, the advantages of multi-core processor can also be utilized by users who design the algorithms or software for the computers. An algorithm that runs sequentially in a single-core processor can be designed or modified into a parallel version to take the advantages of multiple cores in the processor. In that case, algorithm instructions or program code are separated and assigned to different cores to run parallelly. This makes a parallel program running much faster on a multi-core processor than on a single-core processor.

So far, almost all basic algorithms have had their corresponding parallel versions, some of them in many. Such algorithms include sorting, searching, graph algorithms, and basic data structure operations [1].

Parallel algorithms are designed, analyzed and experimented by the researchers for different models of computation, different memory architectures, different instruction sets, different parallel programming languages, and different types of parallel processors. Some of these variations include shared memory models (e.g., PRAM models), distributed memory models, task and data parallelism (e.g., MIMD, SIMD), instruction level parallelism, parallel programming languages and libraries (e.g., MPI, OpenMP), parallel graphics processors, and computers with number of processors from two to thousands. See [1] for a comprehensive study.

Sorting is one of the most basic operations performed by computers and their software. On average, a computer spends almost 25% of its total computation for sorting purposes. Knuth in his seminal book [2] mentions that estimated 25% or more of computing time of all computers are spend for sorting. Moreover, there are many computing systems which spend more than 50% of time for sorting [2].

Among all the sorting algorithms, quick sort is considered to be the fastest comparison-based sorting algorithm in practice. Besides that, quick sort is a simple and elegant recursive sorting algorithm. Quick sort is considered as one of the top ten algorithms in twentieth century [3], [4].

After quick sort was first invented by Hoare in 1962 [5], there have not been that many changes in its basic architecture of the algorithm. The original quick sort was proposed with one pivot. After a pivot element is selected by some method, the algorithm partitions the remaining elements (*partition* function) into two parts. One partition contains the smaller values than the pivot and the other one contains the same or bigger values than the pivot. Then the quick sort algorithm is called recursively for the two partitions.

Researchers tried to improve farther the performance of quick sort with modified partitioning functions [6], [7], such as Lumato partitioning [6], which slightly improve the performance of the quick sort in practice.

There are also attempts to use two pivots in order to improve the running time (see [8] for a discussion). For two pivots, the partition function partitions the elements into three parts. The left one contains all elements less than the first pivot, the right one contains all elements that are greater than the second pivot, and the middle partition contains the remaining elements. These versions of the quick sort are called *dual-pivot* quick sort. However, those attempts for dual-pivot quick sort were all unsuccessful and could not improve the running time.

Recently, Yaroslavsky [9]-[13] has proposed a dual-pivot quick sort which in practice shows better performance than the classical single-pivot quick sort. Since then it got much attention, as it is considered as a breakthrough on improving the original quick sort by a noticeable amount. Oracle was quick enough to perform

extensive experiment and eventually accommodated the Yaroslavsky's algorithm in their standard Java.sort() function [8], [9], [10], [11], [12]. Previously, the Java.sort() function used to use the original single pivot quick sort algorithm. The Yaroslavsky's algorithm was published in an informal way [13]. However, the algorithm was later analyzed in detail and was farther investigated by many researchers who published their results in journals and conferences [8], [9], [10], [11], [12].

Quick sort is considered to be the fastest sorting algorithm in practice when it runs sequentially in a single core processor. However, it also got much attention for having its parallel versions [1], [14]-[19]. These parallel quick sort algorithms have been designed and then analyzed both theoretically [19] and experimentally [14]-[19]. Moreover, they have been analyzed for different hardware architecture, different number of processors, different types of processors, and graphics processors. The results of these analysis show that the performance of parallel versions of quick sort are highly specific to the aforementioned factors. It is very difficult to design a single version of parallel quick sort that would run efficiently in all hardware architecture and all models of computation.

For the case of the recent dual-pivot quick sort algorithm of Yaroslavsky, so far there is a very few attempts [20], [21], [22] to parallelize it. Taotiamton and Kittitornkun presented two parallel implementations [21], [22] and Rattanatranura presented a parallel implementation [20] of the Yaroslavsky's dual-pivot quick sort. All their implementations are in C++. Moreover, they compared their algorithms with the C++ Standard Template Library (STL) Sort() function, rather than comparing with the original dual-pivot quick sort of Yaroslavsky. They claim that their implementations achieve speedup of 3 to 6. Therefore, a comparison between Yaroslavsky's dual-pivot algorithm and its parallel counterpart is still not available in the literature.

Our contribution. In this paper, we take a simplistic and basic approach. We provide a parallel version of the Yaroslavsky's dual-pivot quick sort algorithm. We implement in Java our algorithm as well as the Yaroslavsky's one. We run these two algorithms in the same computer with multiple cores. We run the Yaroslavsky's algorithm in a single-core processor of the computer by keeping all other cores inactive. Then we run our parallel algorithm in the same computer with two, four or eight cores active. All our experiments show that our algorithm outperforms Yaroslavsky's algorithm by a great amount and the speedup achieved by our algorithm is about 3. Moreover, when we compare our algorithm with itself for different number of cores, we see that as the number of cores increases, our algorithm performs better.

We organize the rest of the paper as follows. In Section II, we present our parallel version of dual-pivot quick sort. In Section III, we present the experimental results. Section IV concludes the paper with some future work.

II. PARALLEL DUAL-PIVOT QUICK SORT ALGORITHM

We first describe the dual-pivot quick sort algorithm of Yaroslavsky [9]-[13]. Then we shall describe the modification that we do for making it parallel.

Yaroslavsky's dual-pivot algorithm: We assume that the elements that are to be sorted are in an array A and that they are indexed from 0 to n-1. For the rest of the paper, we call the Yaroslavsky's dual-pivot algorithm as algorithm 1 and our algorithm as algorithm 2. The two pivots in algorithm 1 are called P1 and P2. P1 and P2 are selected as the first and last elements of the array, which are A[0] and A[n-1], respectively. If P2 is smaller than P1, then we swap them so that the first pivot is always same or smaller than the second pivot.

Algorithm 1 works as a quick sort if the number of elements in A is bigger than some threshold. Otherwise, it works as an insertion sort. The threshold can be set by the user. The threshold set by the author in his original version of the algorithm is twenty-seven. If n is smaller than this threshold, then the insertion sort is used. Otherwise, if n is same or bigger than this threshold, then algorithm 1 partitions the elements into three parts. This is called *partitioning.* Elements that are less than P1 go to the first (or left) part. Elements that are greater

than or equal to P1 and less than or equal to P2, go to the second (or middle) part. Elements that are greater than P2 go to the third (or last) part. For each partition, quick sort is recursively called, and the partition becomes the input array A. These three parts are denoted as *Part I*, *Part II*, and *Part III* respectively.

During the partition process, elements of A are scanned from left to right and gradually putted into the three partitions. At any time, the unscanned elements remain together in a contiguous block, which is called *Unscanned Block*. Similarly, Part I, Part II, and Part III remain as contiguous blocks. Five indices are used to track the boundaries of the three partitions as well as the Unscanned Block. These pointers are *left*, *L*, *K*, *G* and *right*. Left points to the first element of A. This element is also at the left of P1 when Part I is non-empty. Right points to the last element of A. This element is also at the right of P2 when Part III is non-empty. L points to the first element of Part II. K and G point to the first and last elements of the Unscanned Block respectively. See Figure 1. Once the partitioning is done, the algorithm calls itself recursively for Part I, Part II, and Part III. See Algorithm 1 for the pseudo code of the algorithm and the partition function.

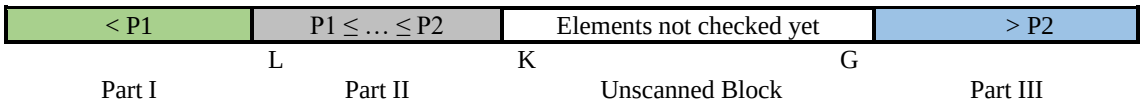

Fig. 1. A generic view of partitioning in algorithm 1. Green, grey and blue colors depict three partitions Parts I, II, III respectively. White color depicts elements that have not been checked yet (Unscanned Block).

Function Partition (Array A)

1. Select two pivots P1 and P2 as A[left] and A[right] respectively
2. If P2 is less than P1, then swap them
3. Divide A into the following three Parts I, II, and III. Initially, Parts I, II and III are empty and all elements of A are in Unscanned Block
    - Part I, A[left+1, …, L–1]: All elements less than P1
    - Part II, A[L, …, K–1]: All elements greater or equal to P1 and less or equal to P2
    - Part III, A[G+1, …, right–1]: All elements greater than P2 goes to Part III with indices
4. In Unscanned Block, the next element A[K] from left to right is compared with the two pivots P1 and P2 and placed into the corresponding Parts I, II or III
5. Then the pointers L, K, and G are adjusted accordingly
6. Steps 4-5 is repeated as long as K ≤ G
7. The last element of Part I is swapped with P1, and the first element of Part III is swapped with P2

End of Function

Algorithm 1 (Array A)

1. If (size of A ≤ 27), then use Insertion sort and return
2. Call Partition (A)
3. Call Algorithm 1 (Part I)
4. Call Algorithm 1 (Part II)
5. Call Algorithm 1 (Part III)

End of Algorithm

Algorithm 1. Pseudo code of algorithm 1 and partition function

Our parallel algorithm: Our algorithm for parallelization works in Step 3-5 of the algorithm 1. The first two steps remain same. In steps 3-5, instead of executing the algorithm recursively for Parts I, II and III in sequence, our algorithm executes them in parallel if their size is bigger than a threshold and if there are available threads. The threshold is a value that can be chosen by a user. We choose this value to be 8192. This value is not an arbitrary value. Java uses this value as its own threshold to determine whether its sort function should run sequentially or parallelly [23].

As the algorithm 2 runs recursively, many parts of different size will be generated, and the available threads will be occupied. Therefore, at some time it may happen that for a recursive call of the algorithm 2, there may not be enough available threads for allocating all three Parts I, II and III. In that case, priority is given to the parts by their size. A part with bigger size will get higher priority. See Algorithm 2 for the pseudo code of the algorithm.

Algorithm 2 (Array A)

1. If (size of A ≤ 27), then use Insertion sort and return } Steps 1-2 remain same as in algorithm 1
2. Call Partition (A)
3. If (Part I size ≤ 8192), then Call Algorithm 1 (Part I) in the current thread
4. If (Part II size ≤ 8192), then Call Algorithm 1 (Part II) in the current thread
5. If (Part III size ≤ 8192), then Call Algorithm 1 (Part III) in the current thread
6. If (Part I size or Part II size or Part III size > 8192) then
7. Call Algorithm 2 (largest part among Parts I, II, III with size > 8192) in a new thread (if thread available)
8. Call Algorithm 2 (second largest part among Parts I, II, III with size > 8192) in a new thread (if thread available)
9. Call Algorithm 2 (smallest part among Parts I, II, III with size > 8192) in a new thread (if thread available)

End of Algorithm

Algorithm 2. Pseudo code of algorithm 2

III. EXPERIMENTAL RESULTS

We have implemented our algorithm in Java (JDK 8.2). We have run our algorithm in a computer having eight cores. The computer consists of a processor Intel Core i7-6700 @ 3.40 GHz with 8 cores and 8GB RAM. The operating system is Windows 10 Pro 64-bit.

We have executed algorithm 1 in that computer with only one core by deactivating the remaining seven cores. We measure the running time of algorithm 1 with that setup. Then we have executed algorithm 2 in that computer with two cores (by deactivating six other cores), four cores (by deactivating four other cores), and eight cores. We have measured the running time of algorithm 2 in those setups. See Tables I-VI.

We have generated random input of six different settings. In all settings, we take three different input sets of size 2^{28}, 2^{29} and 2^{30} respectively. The domain of the input values varies in different settings. In the first settings, the value of the inputs is randomly taken from 0 to 10. In the next two settings, the value domains are 0 to 100 and 0 to 1000 respectively. Therefore, in these three settings, there are a lot of repeated values in the input. In the last three settings, the input values are all unique. In fourth, fifth and sixth settings, the input values are sorted in ascending order, sorted in descending order, and unsorted respectively. See Tables I-VI.

TABLE I
EXPERIMENTAL RESULTS WITH RANDOM INPUT. INPUT VALUE DOMAIN 0-10

Value domain: random 0-10						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	3281	-	7812	-	15000	-
2	3500	0.90	6422	1.22	13125	1.14
4	2922	1.12	5812	1.34	10928	1.37
8	2704	1.21	6656	1.17	10344	1.45

TABLE II
EXPERIMENTAL RESULTS WITH RANDOM INPUT. INPUT VALUE DOMAIN $0\text{-}10^2$

Value domain: random $0\text{-}10^2$						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	7283	-	12734	-	27445	-
2	6468	1.13	12594	1.01	22443	1.22
4	4578	1.59	10828	1.18	21219	1.29
8	4125	1.77	6875	1.85	12453	1.20

TABLE III
EXPERIMENTAL RESULTS WITH RANDOM INPUT. INPUT VALUE DOMAIN $0\text{-}10^3$

Value domain: random $0\text{-}10^3$						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	11063	-	19547	-	38109	-
2	7921	1.40	15465	1.26	31391	1.21
4	6719	1.65	12071	1.62	22078	1.73
8	5485	2.02	10828	1.81	17938	2.12

TABLE IV
EXPERIMENTAL RESULTS WITH UNIQUE VALUES IN ASCENDING ORDER

Value domain: Unique ascending order						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	3681	-	7453	-	15438	-
2	3484	1.06	7422	1.00	14437	1.07
4	2297	1.60	4141	1.80	7922	1.95
8	1562	2.36	3172	2.35	6144	2.51

TABLE V
EXPERIMENTAL RESULTS WITH UNIQUE VALUES IN DESCENDING ORDER

Value domain: Unique descending order						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	4156	-	8640	-	17344	-
2	3813	1.09	8141	1.06	16140	1.07
4	2297	1.81	4703	1.84	9140	1.90
8	2085	1.99	4015	2.15	7656	2.27

TABLE VI
EXPERIMENTAL RESULTS WITH RANDOM UNIQUE VALUES

Value domain: Random unique						
Input size →	**2^{28}**		**2^{29}**		**2^{30}**	
#Processor ↓	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
1	26188	-	52875	-	112471	-
2	25486	1.03	48620	1.09	85996	1.31
4	20803	1.26	43116	1.23	58750	1.91
8	11781	2.22	26172	2.02	36651	3.07

From the experimental results, it can be seen that in most of the cases, the running time improves substantially as the number of processors is increased. Moreover, the speedup gets higher as the input size is increased. The speedup is computed by the following standard function [1]:

speedup = (time of algorithm 1 in one core)/(time of algorithm 2 in multi core)

For input in ascending order (Table IV) the speedup for higher number of processors is better than the input in descending order (Table V). This is due to the fact that the output will be in ascending order. So, the input in descending order will need some swapping to become ascending. Whereas, the input in ascending order are already sorted and need no swapping. Another observation is that the results in Tables IV, V and VI all assume that the input values are unique. So, the speedup in these three cases are similar, but they are higher than the previous three tables as they have repetition in input values.

Among the six settings that we considered, the final setting is the most significant, because its input values are random and have no repetition. Here, the speedup is as much as 3.07. See Table VI. Figures 4 and 5 illustrate more about the improvement of running time and the speedup in this setting. These two figures illustrate that in the last setting the running time and the speedup is improved more for eight processors as well as for the highest input size.

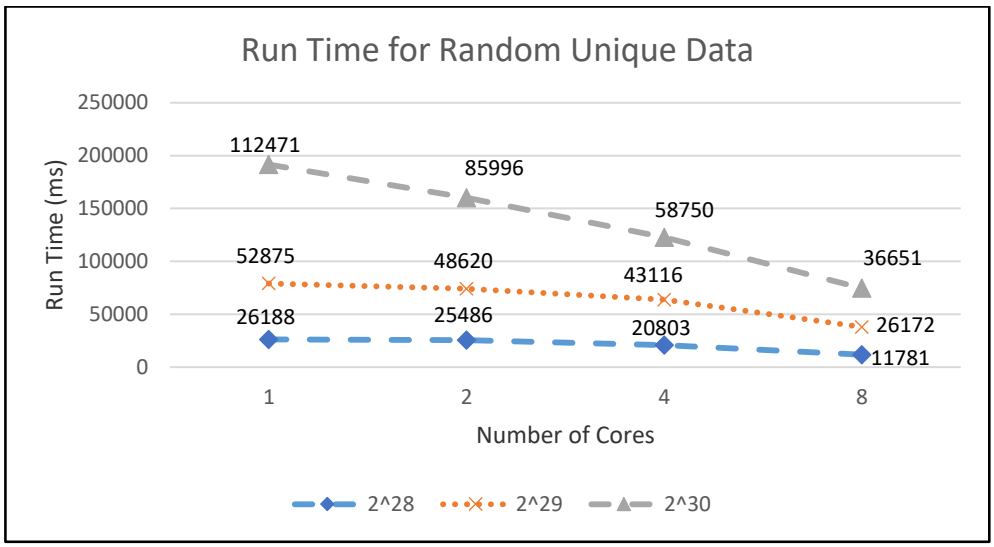

Fig. 2. Run time improvement for random unique data

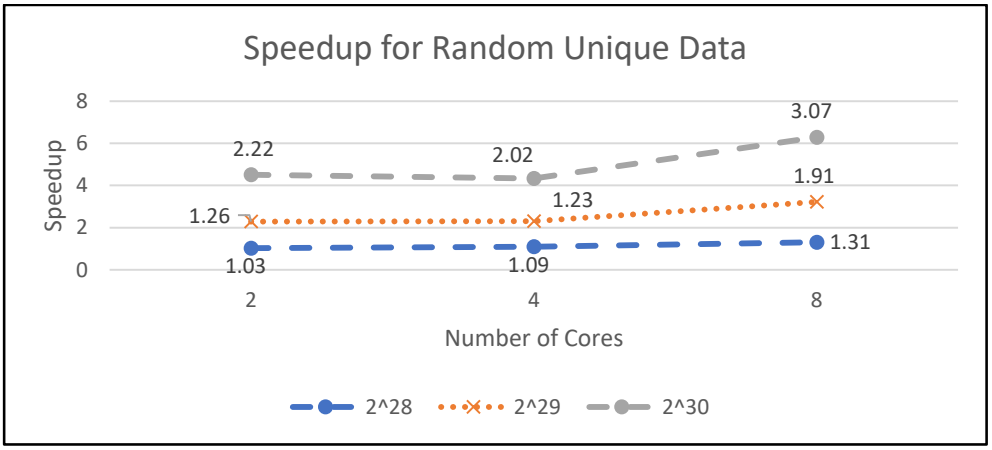

Fig. 3. Speedup improvement for random unique data

IV. DISCUSSION AND CONCLUSION

From the experimental results it can be seen that as the input size and the number of processors are increased, our algorithm gets better performance. This suggests that the simple modification that we do in our algorithm to make the algorithm 1 parallel works fine for the recent days' personal computers that have about two to eight cores. Not only that, in the near future, when the personal computers, laptops, mobile devices, and similar devices will have more processors, our algorithm can continue performing better and better.

In this paper, we have presented a parallel version of the Yaroslavsky's dual-pivot quick sort algorithm that has got much attention as a breakthrough to the classical quick sort algorithm. Our modification to make the algorithm parallel is simple. Our experiment is also done in recent days' computers (PCs, laptops, etc.) that have small number of processors. The experimental results show that our algorithm runs much faster than the original Yaroslavsky's quick sort algorithm.

There are many works that can be done in future. We have not concentrated to parallelize the partition function. If partition functions can be carefully parallelized, then more efficiency could be achieved. However, that may lose the simplicity of the parallel algorithm. There is also a scope to parallelize the quick sort algorithm that has more than two pivots [12].

ACKNOWLEDGMENT

The authors would like to thank Dr. Noraziah Ahmad of University Malaysia Pahang for helping us in translating the abstract and keywords in Bahasa Melayu.

REFERENCES

[1] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*, Springer, 2019.
[2] D. E. Knuth, *The Art of Computer Programming volume 3: Sorting and Searching, 2nd ed*. Addison-Wesley. 1998.
[3] B. A. Cipra. (2000, May). The best of the 20th century: Editors name top 10 algorithms. *SIAM News. volume(33),* pp. 1-2.
[4] J. Dongarra and F. Sullivan. (2000, January/February). Introduction: the top 10 algorithms, computing in science & engineering. *IEEE. volume 2*, pp. 22-23.
[5] C. A. R. Hoare. (1962). Quicksort. *Computer Journal*. *volume(*5), pp. 10–16.
[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms, 3rd ed*. Cambridge, Massachusetts, USA, MIT Press, 2009.
[7] R. Sedgewick. (1977). The analysis of quicksort programs, *Acta Informatica*. *volume(7),* pp. 327–355.
[8] S. Wild and M. E. Nebel. (2012). Average case analysis of Java 7's dual pivot quicksort. *ESA, LNCS, Springer, Berlin. volume(7501),* pp. 825–836.
[9] M. E. Nebel, S. Wild and C. Martínez. (2016). Analysis of pivot sampling in dual-pivot quicksort: a holistic analysis of Yaroslavskiy's partitioning scheme. *Algorithmica, Springer. volume (75),* pp. 632-683.
[10] S. Wild, M. E Nebel, and R. Neininger. (2015, January). Average case and distributional analysis of dual-pivot quicksort. *ACM Transactions on Algorithms. volume (11),* pp. 22.
[11] S. Wild, M. E. Nebel and H. Mahmoud. (2016). Analysis of quickselect under Yaroslavskiy's dual-pivoting algorithm. *Algorithmica*. *volume (74),* pp. 485–506.
[12] S. Kushagra, A. López-Ortiz, J. I. Munro, and A. Qiao. (2014, January). Multi-pivot quicksort: theory and experiments. *Proceedings of the Meeting on Algorithm Engineering & Expermiments (ALENEX), SIAM*, pp. 47–60.
[13] V. Yaroslavsky. (2009, September). Dual-Pivot Quicksort algorithm. Available: https://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf.
[14] S. S. M. Al-Dabbagh and N. H. Barnouti. (2016, June). Parallel quicksort algorithm using OpenMP. *International Journal of Computer Science and Mobile Computing. volume (5)*, pp.372–382.
[15] P. Sanders and T. Hansch. (1997). Efficient massively parallel quicksort. *International Symposium on Solving Irregularly Structured Problems in Parallel IRREGULAR 1997, Springer LNCS. volume (1253),* pp. 13-24.
[16] P. Tsigas and Y. Zhang. (2003, February). A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, Genova, Italy, IEEE, volume(5-7),* pp. 372-381.
[17] D. Cederman and P. Tsigas, (2010, January). GPU-Quicksort: a practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, pp. 4.
[18] D. Pasetto and A. Akhriev, (2011, October). A comparative study of parallel sort algorithms. *proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications, Portland, Oregon, USA*, pp. 203-204.
[19] B. A. Mahafzah. (2013). Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *Journal of Supercomputing, Springer. volume (66),* pp. 339–363.
[20] A. Rattanatranurak, (2018, June). Dual parallel partition sorting algorithm. *Proceedings of 2018 the 8th International Workshop on Computer Science and Engineering (WCSE 2018), Bangkok*, pp. 685 -690.
[21] S. Taotiamton and S. Kittitornkun. (2017, November). A parallel dual-pivot quicksort algorithm with lomuto partition. *21st International Computer Science and Engineering Conference (ICSEC), Bangkok, Thailand, IEEE*, pp. 15-18.
[22] S. Taotiamton and S. Kittitornkun. (2017, June). Parallel hybrid dual pivot sorting algorithm. *14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), Phuket, Thailand, IEEE*, pp. 27-30.
[23] GeeksforGeeks. (2020). Serial Sort v/s Parallel Sort in Java. Available: https://www.geeksforgeeks.org/serial-sort-vs-parallel-sort-java/