IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 8, 2021, accepted February 21, 2021, date of publication March 2, 2021, date of current version March 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3063205

On the Performance of Mean-Based Sort for Large Data Sets

SHAHRIAR SHIRVANI MOGHADDAM[1], (Senior Member, IEEE), AND KIAKSAR SHIRVANI MOGHADDAM[2], (Student Member, IEEE)

[1]Faculty of Electrical Engineering, Shahid Rajaee Teacher Training University, Tehran 1678815811, Iran
[2]School of Computer Engineering, Iran University of Science and Technology, Tehran 1684613114, Iran

Corresponding author: Shahriar Shirvani Moghaddam (sh_shirvani@sru.ac.ir)

ABSTRACT Computer and communication systems and networks deal with many cases that require rearrangement of data either in descending or ascending order. This operation is called sorting, and the purpose of an efficient sorting algorithm is to reduce the computational complexity and time taken to perform the comparison, swapping, and assignment operations. In this article, we propose an efficient mean-based sorting algorithm that sorts integer/non-integer data by making approximately the same length independent quasi-sorted subarrays. It gradually finds sorted data and checks if the elements are partially sorted or have similar values. The elapsed time, the number of divisions and swaps, and the difference between the locations of the sorted and unsorted data in different samples demonstrate the superiority of the proposed algorithm to the Merge, Quick, Heap, and conventional mean-based sorts for both integer and non-integer large data sets which are random or partially/entirely sorted. Numerical analyses indicate that the mean-based pivot is appropriate for making subarrays with approximately similar lengths. Also, the complexity study shows that the proposed mean-based sorting algorithm offers a memory complexity same as the Quick-sort and a time complexity better than the Merge, Heap, and Quick sorts in the best-case. It is similar to the Merge and Heap sorts in view of the time complexity of the worst-case much better than the Quick-sort while these algorithms experience identical complexity in the average-case. In addition to finding part by part incremental (or decremental) sorted data before reaching the end, it can be implemented by parallel processing the sections running at the same time faster than the other conventional algorithms due to having independent subarrays with similar lengths.

INDEX TERMS Ascending, descending, divide-and-conquer, heap, integer, large data set, merge, non-integer, quick, sorting.

I. INTRODUCTION

Facing a large volume of data in various civil, medical, industrial, and military applications demonstrates that sorting the data in the shortest possible time is the main goal [1]–[4]. The time and power consumed to do the sorting depend on the computational complexity of the sorting algorithm and the length and type of data [5] that may differ from one application to another.

Sorting is the process of ordering unsorted data in a descending (from high to low) or an ascending (from low to high) order. This data can be an integer or non-integer numbers, letters of the alphabet, or specific features or signals [6]. Some practical examples of numerical and non-numerical data used in everyday life are landline and mobile phone numbers, dictionaries, license plates, codes for different colors, student and staff numbers, national code and ID number, passport number, username and password to access e-mail and various portals and systems, bank account number, and bank card number [7], [8].

The Sorting process facilitates queuing, classification, clustering, segmentation, separation, search, and so on. In large networks based on massive multi-input multi-output (massive-MIMO) and ultra-dense networks (UDNs), as well as heterogeneous networks (HetNets) consisting of small cells [9]–[11], dense sensor networks with a large number of sensors, and internet of things (IoT) applications, the number of nodes is very large. It is necessary to efficiently select

The associate editor coordinating the review of this manuscript and approving it for publication was Senthil Kumar.

and assign radio resources to solve large-scale optimization problems.

The main applications of data sorting include ascending or descending sorting, faster data search, speed up the clustering process, speed up the classification process, assist in the process of deleting data below or above a threshold level, recover lost data, find the highest and lowest amount of data, extract the median of data, find similar data, determine the dynamic range of data, and obtain the frequency and histogram of data [6], [12].

Depending on whether the data is numerical or non-numerical, integer or non-integer, random (uniform, Gaussian) or non-random (approximately sorted, somewhat sorted), and in terms of size, small, medium, or large, various algorithms are introduced and used. Also, hardware and software capabilities, allowed processing time, and costs are essential items in determining the type of algorithm [13], [14].

In this regard, evaluation and comparison of algorithms are based on memory space complexity, time complexity, elapsed time to the sorting process, number of comparisons and swaps, number of division levels in methods based on divide-and-conquer, gradual sorting rate, and the sensitivity of the algorithm to the type and length of data. On the other hand, sorting methods can be divided according to the realization (series or parallel) and type of programming (recursive or non-recursive). Specific-purpose or general-purpose, comparative or non-comparative, counting or non-counting, in-place or non- in-place, stable or unstable, adaptive or non-adaptive are the other classifications in the field of sorting algorithms [15], [16].

In the category of comparison-based sorting algorithms, we can name Selection, Bubble, Insertion, Merge, Quick, Heap, Shell, Tim, Comb, Cycle, Strand, Binary Insertion, Tree, Cartesian Tree, Cocktail, and Odd-Even. Among the non-comparison-based algorithms capable of sorting integer data, Counting, Radix, Bucket, Pigeonhole, and Tag are the most important ones. Specific-purpose sorting algorithms include Bitonic, Bogo/Permutation, Pancake, Sleep, Stooge [17]–[19].

In this work, we propose a new sorting algorithm and compare it to the Quick, Merge, and Heap with more detail, and also to the Insertion, Bubble, and Selection in some aspects. Among the reasons why it is necessary to review and improve existing algorithms and introduce new data sorting algorithms, the following can be mentioned:

1) The abundance of databases;
2) Large-scale data sets;
3) High speed of change of phenomena and the need to update them;
4) Complex problems with a large number of parameters and criteria, such as optimization problems in resource allocation and power control of device-to-device (D2D), vehicle-to-vehicle (V2V), and machine-to-machine (M2M) communications [20]–[22].

The divide-and-conquer method widely used in sorting, recursively divides a problem into sub-problems of the same or related type, until these become simple enough to be solved directly. Many algorithms that follow this principle, are structurally recursive and can invoke the typical algorithm itself once or even more times to solve tightly related sub-problems [23]–[25].

Many divide-and-conquer sorting algorithms are suitable for large data sets, especially for internet and communication systems and networks [26], [27]. Merge-sort produces a sorted sequence by sorting its two halves and merging them [28]. It is a top-down implementation that starts from the entire array and then splits the original problem into unitary items [24]. Quick-sort picks an element, the first, last, middle, or a random one, as the pivot and partitions the given array around the pivot [29], [30]. Heap-sort based on a binary Heap data structure finds the maximum element, places it at the end, and then repeats the same process for the remaining elements. References [31], [32] introduce a sorting algorithm that divides a single array into two smaller sub-arrays based on the mean value and continues it to reach one-element ones more efficient than Insertion and Selection but less efficient than Merge and Quick sorts.

Merge-sort does not work well for completely unsorted and random non-integer large data set, and we do not have any sorted data until the end. Due to the randomness of the Quick-sort pivot, unequal subarrays are generated, and there is no control when we want to get only a certain number of sorted data. Dealing with low-sorted, somewhat sorted, or similar data in a list, imposes unnecessary comparisons and swaps. The Heap-sort requires first to find the maximum element, put it at the end, and do it for the other elements that are a time-consuming process. In the algorithm proposed in [31], [32], unsorted data remains unsorted until subarrays with similar data are found, or the final level with double-member subarrays is reached. It means that it is not possible to detect the sorted data at any level of sorting. The order of elements changes in the right-hand subarray, which causes unnecessary extra swaps and disrupts the adaptivity. Furthermore, if there is similar data in the list, especially in the right subarray, the stability will be violated. In processing the right and left subarrays, the entire array is updated at each level, which causes a considerable number of swaps that directly increases the elapsed time and complexity.

Although Merge, Heap, and Quick sorts are widely used for sorting large data sets [33], [34], the processing time should be decreased, and gradual sorting is a necessity [35]–[37]. Hence, in this article, we propose an efficient mean-based sorting algorithm that uses the pros of the popular divide-and-conquer algorithms and offers some new useful properties. Also, we compare the proposed algorithm with the Quick, Merge, and Heap sorts, which work much better than the conventional mean-based sorting algorithm.

It is compared to the Quick, Merge, Heap, Insertion, Selection, and Bubble sorts in view of the time and memory complexity and new features that are the main novelties,

especially to the Quick, Merge, and Heap sorting algorithms in the large data sets. The new capabilities such as detection of a sorted part, detection of a part with similar data, Higher ability to sort data gradually, making independent subarrays with approximately the same length, and better stability and adaptivity compared to the Quick-sort, are the main features of the proposed algorithm.

The proposed ideas reported in [38]–[44] are focused on the Merge and Quick sorts in parallel processing and compare them to the conventional Merge, Quick, and Heap sorts by proposing a flexible division of tasks between logical processors to show that this proposition is a valuable method that can find many practical applications in high-performance computing. By separation of concerns, each of the processors works separately. The proposed method was described in a theoretical way, examined in tests, and compared to other methods. The results confirm high efficiency and show that by adding a new processor, sorting becomes faster and more efficient, especially for large data sets. None of these algorithms makes the independent subarrays in parallel processing that the proposed K-S mean-based sorting algorithm makes them with approximately the same length. It means that the proposed algorithm can be examined in the parallel realization by using multicore architectures to find faster and more efficient implementations than those mentioned in [38], [39], [42], [43].

In the following, Section II illustrates the proposed K-S mean-based sort and its *C#* pseudo-code. Section III presents theoretical analyses and formulations. Section IV compares the mostly used conventional algorithms and the new one based on the running time, the number of swaps and divisions, and a new proposed measure, namely normalized mean absolute error, numerically. Section V has an in-depth analysis of the mean-based pivot and its effect on the integer and non-integer data in low and high standard deviations for different division levels. In Section VI, we derive the time complexity order of the proposed algorithm in the best, worst, and average cases. Section VII compares the proposed algorithm to the Quick, Merge, Heap, Insertion, Bubble, and Selection sorting algorithms in different viewpoints. Finally, Section VIII concludes this article and introduces an idea to future works.

II. PROPOSED MEAN-BASED SORTING ALGORITHM

Like the conventional mean-based (relative) sort [31], [32], the proposed K-S mean-based algorithm makes two independent subarrays in each level, one greater (smaller) and another smaller (greater) than the mean value. Unlike the conventional ones, the left subarray goes to the next step up to be divided entirely and sorted while the right one has no change in these steps. Then, it comes back to the previous levels to do that for the right parts, similarly. This algorithm has no additional array and gradually reaches the sorted parts from left to right. Algorithm 1 demonstrates the pseudo-code of the proposed algorithm supporting the following features:

1) In each level, just those parts that are compared to the mean value may be changed if the section involved in the division does not have sorted data or similar elements.
2) After comparing the data with the mean value, if the elements are similar to the main array, they may be sorted. It can be checked by comparing the neighbor elements to each other. If all comparisons are positive (negative), the section is sorted increasingly (decreasingly).
3) If the data in each division is assigned entirely to one subarray means they are similar.
4) The approximate mean value is used for the big data and the first partitioning levels, which does not change the average much, but does take less time to calculate.
5) Achieving a certain number of sorted data is possible using several mean-based divisions or a single-level division made by a threshold based on the data's mean and variance.
6) The left and right parts almost in the same length can be processed by parallel processing, efficiently.

III. THEORETICAL ANALYSES AND FORMULATIONS

The probability of locating positive numbers larger or smaller than a threshold level can be measured from the Markov inequality [45], as (1). In the particular case, which always occurs in the Heap-sort and some instances in the Quick-sort, if the threshold level, α, is the largest (smallest) number, there is no data (total data) more than that, and the complete data (no data) will be smaller than that. In addition to the positive parts of data, this inequality can be applied to the negative part when both positive and negative elements are present in the data set.

$$P(X \geq \alpha) \leq \frac{E(X)}{\alpha}, \quad X > 0 \tag{1}$$

In the proposed algorithm, in each division, half of the data is in the upper subarray and the next half in the lower subarray, on average. It is assumed that the total number of data is $N = 2^n$, it is entirely random, there are no similar data in a subarray, and the data of each subarray are not sorted until the double-member subarray is reached. Also, the next subarrays have lengths of $2^{n-1}, 2^{n-2}, \ldots, 2$. Therefore, in the worst-case, the number of divisions, N_{dm}, is

$$N_{dm} = n - 1 = log_2^N - 1 \tag{2}$$

In the Quick-sort, the pivot is randomly one of the elements of the data set with equal probability. Therefore, on average, the absolute difference between the data lengths of the upper and lower parts around the pivot, $|\Delta L|$, can be with a probability equal to $\frac{1}{N-1}$ in the range $[0, N-1]$ with an average equal to $\frac{N-1}{2}$. So,

$$|L_U - L_L| = \frac{N-1}{2} \tag{3}$$

and we know that

$$L_U + L_L = N - 1 \tag{4}$$

Algorithm 1 *C#* Pseudo-Code of the Proposed K-S Mean-Based Sorting Algorithm

```
Variables:
arr: Array of data.
si: Starting index.
ei: Ending index.
swapped: Boolean variable for showing if any swap happened in the previous partitioning.
ordered: Boolean variable for showing if this part has descending order.
sum: Variable for storing sum of each part.
mean: Variable for storing mean of each part.
bi: Boundary index.
Function Sort (arr, si, ei) :
    if ei − si > 0 then
        swapped ⟵ False
        Call Partition (arr, si, ei, ref swapped)
        bi ⟵ returned value from partition
        ordered ⟵ False
        if swapped = False then
            ordered ⟵True
            for i ⟵ si to ei − 1 do
                if arr[i] > arr[i + 1] then
                    ordered ⟵ False
                    break
                end
            end
        end
        if ordered = False then
            Call Sort (arr, si, bi)
            Call Sort (srr, bi + 1, ei)
        end
    end
    return void
End Function
Function Partition (arr, si, ei, ref swapped) :
    bi ⟵ si − 1
    sum ⟵ 0
    for i ⟶ si to ei do
        sum ⟵ sum + arr[i]
    end
    mean ⟵ sum/(ei−si+1)
    for i ⟵ si to ei do
        if arr[i] < mean then
            bi ⟵ bi + 1
            if arr[bi] ≠ arr[i] then
                swap arr[bi] with arr[i]
                swapped ⟵true
            end
        end
    end
    return bi
End Function
```

Assuming that the upper subarray is larger than the lower one,

$$\begin{cases} L_U = \frac{3}{4}(N-1) \\ L_L = \frac{1}{4}(N-1) \end{cases} \tag{5}$$

In the next divisions for the larger subarrays, we reach the following lengths:

$$\frac{3}{2^2}(N-1),\ \frac{3^2}{2^4}(N-2),\ \ldots,\ \frac{3^{N_{dq}}}{2^{2N_{dq}}}(N-N_{dq}) \tag{6}$$

Also, for the smaller subarrays, we have

$$\frac{1}{2^2}(N-1),\ \frac{1}{2^4}(N-2),\ \ldots,\ \frac{1}{2^{2N_{dq}}}(N-N_{dq}) \tag{7}$$

In the last division, we have

$$\left(N-N_{dq}\right)\left(\frac{3}{2^2}\right)^{N_{dq}} = 2 \tag{8}$$

or

$$2\left(\frac{4}{3}\right)^{N_{dq}} + N_{dq} = N \tag{9}$$

For large N_{dq}, the first term is much larger than the second one. Therefore, the number of divisions is approximately

$$N_{dq} = \frac{log_2^N - 1}{2 - log_2^3} = 2.41(log_2^N - 1) \tag{10}$$

According to equations 3 and 5, at each division in the Quick-sort, the length of the larger subarray is three times the size of the smaller one, on average, and the difference between the lengths of them is 50% of the total data on which the division is performed. Also, using equations 2 and 10, the ratio of the average number of the divisions of the Quick-sort, N_{dq}, to that for the proposed algorithm, N_{dm}, is equal to

$$\frac{N_{dq}}{N_{dm}} = 2.41 \tag{11}$$

That is, the Quick-sort requires about 140% more divisions than the proposed algorithm. On the other hand, when the data within a subarray are sorted (incremental or decremental) or similar to each other, there is no need for further divisions and swaps in the proposed algorithm. Therefore, the number of divisions obtained is smaller than the upper bound in equation 2. Besides, comparing the average number of data in a larger subarray in the Quick-sort, $\frac{3N}{4}$, to that for each subarray in the proposed algorithm, $\frac{N}{2}$, indicates 50% more comparisons, swaps, and processing time in the parallel realization.

Different algorithms can be compared to each other by defining a new measure based on the difference between the location of the sorted data to that for the unsorted data as

$$e^j(i) = l_u^j(i) - l_s(i), \quad i = 1, 2, \ldots, N \tag{12}$$

In this equation, $l_s(i)$ is the location of the element i after complete sorting, and $l_u^j(i)$ is the location of the element i in

the step j. The location of each element in these two cases is in range $[1, N]$. The difference between these two values in each location is in range $[0, N-1]$. We propose the normalized mean absolute error (NMAE) as

$$NMAE^j = \frac{\sum_{i=1}^{N} |e^j(i))|}{MAE} = \frac{\sum_{i=1}^{N} |l_u^j(i) - l_s(i)|}{MAE} \quad (13)$$

It can be intuitively found that in an N-element array, the maximum absolute error (MAE) between the sorted and unsorted data is equal to

$$MAE = 2 \begin{cases} 2 \times \left(1 + 3 + \ldots + \frac{N-3}{2} + \frac{N-1}{2}\right), \\ \quad N = 2k-1 \\ 1 + 3 + \ldots + (N-3) + (N-1), \\ \quad N = 2k \end{cases} \quad (14)$$

By obtaining these sums, we have

$$MAE = \frac{1}{2} \begin{cases} (N-1)(N+1), & N = 2k-1 \\ N^2, & N = 2k \end{cases} \quad (15)$$

For large N, these two equations are approximately equal to $\frac{N^2}{2}$. Finally, the suggested measure is formulated as

$$NMAE^j = \begin{cases} \frac{2\sum_{i=1}^{N} |l_u^j(i) - l_s(i)|}{(N-1)(N+1)}, & N = 2k-1 \\ \frac{2\sum_{i=1}^{N} |l_u^j(i) - l_s(i)|}{N^2}, & N = 2k \end{cases} \quad (16)$$

IV. NUMERICAL ANALYSES AND COMPARISONS

In [31], [32], it was shown that the Merge, Heap, and Quick sorts work better in both integer and non-integer data than the mean-based (or relative) sort algorithm, especially in the medium and large data sets. Hence, based on the required elapsed time, the number of swaps, the number of divisions, and the difference between the sorted and unsorted data in different samples, the proposed K-S mean-based sorting algorithm is compared to the Merge, Quick, and Heap sorts, indicating the achievement to an efficient and effective mean-based sorting algorithm.

All simulation parameters and performance metrics used in this article are summarized in Table 1. Simulations are performed for integer and non-integer Gaussian and uniform distributions for partially to entirely sorted scenarios, on a 64-bit system with the specifications summarized in Table 2.

Simulations show that in the case of non-integer data, a part of the data may be sorted, but the probability of similar data is low. On the contrary, in the case of integer data, similar data is more likely to occur, but for high variances, it will be decreased. The lower the data variance, the greater the possibility of similarity of data causes unnecessary divisions required in the conventional sorts. Herein, it is investigated that the Quick-sort has a big problem, namely stack overflow, when there is a huge number of similarity in data. It happens mostly for low-variance data and/or integer data.

TABLE 1. Simulation parameters and performance metrics.

Parameter/metric	Value/description
Type of data	Integer and non-integer
Data distribution	Gaussian and uniform
Data's standard deviation	5, 13, 1000
Data length	1000000
Percentage of the sorted parts	10-100
Number of iterations	1000
Swap	The number of swaps
Division	The number of divisions
Elapsed time	The running time to sort data
NSI	Normalized similarity index
NMAE	Normalized mean absolute error

TABLE 2. Software and computer system specifications.

Specification	Description
Software language	*C#*
Software package	MATLAB
Laptop model	Ubuntu 20.04.1 LTS
Computer system	64-bit, Intel Core i7-9750H
CPU max. speed	2.6GHz
RAM space	16GB
Cache space	12MB

In these cases, although the Quick-sort experiences a large number of unnecessary swaps, they never occur in the proposed mean-based algorithm. Hence, to have the simulation results in an acceptable time, the minimum value of the data's standard deviation is set 13 to overcome the stack overflow in the Quick-sort.

The elapsed time for the proposed algorithm in the case of integer and non-integer data sets are close to each other while they are significantly different for Quick-sort. For different mean and variance values, it is shown that the number of elements of the two subarrays in the Quick-sort differs by as much as 50% of the total data. In comparison, the lengths of the two subarrays are almost equal in the proposed algorithm. These simulation results are valid for different sizes, variances, and types of data.

Figs. 1, 2, and 3 respectively depict the required elapsed time, the number of swaps, and the number of divisions, in partially to entirely sorted data scenarios in both integer and non-integer data. The proposed algorithm offers higher performance in terms of the metrics mentioned above than the others in both integer and non-integer data sets.

In the integer data, the elapsed times of the Heap and Quick algorithms do not depend on the percentage of the sorted parts. Still, in the Merge and the proposed sorting algorithms, we have a slight decrease because similar data is possible, which in the proposed one is no longer divided and swapped, and we have no swap in the Merge-sort. In the non-integer data, increasing the percentage of the sorted parts causes a reduction in the elapsed time of the Heap, Merge, and the proposed algorithms, because the number of swaps reduces. The similar data decreases the need for additional swaps, which causes lower running time of the Heap and

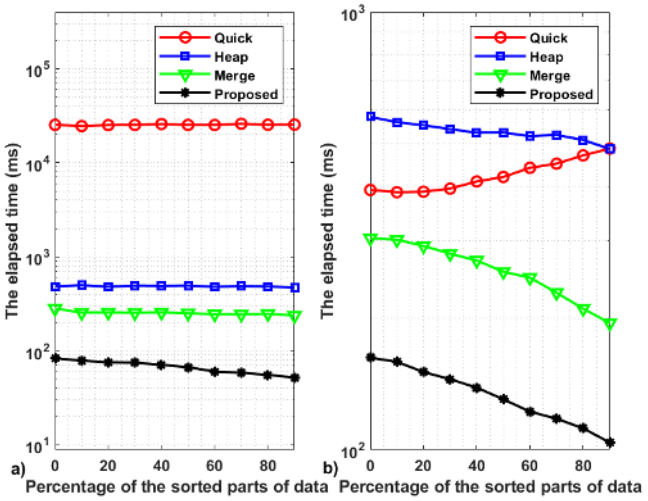

FIGURE 1. The required elapsed time for partially to entirely sorted data with length 1000000, a) integer, b) non-integer.

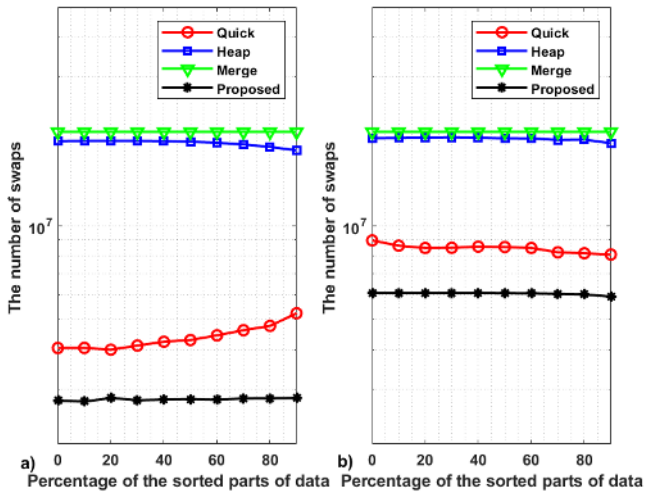

FIGURE 2. The number of swaps for partially to entirely sorted data with length 1000000, a) integer, b) non-integer.

Merge in the integer data sorting compared to that for the non-integer data. Because the probability of similar data in each level is high, the higher running time of the Quick-sort is in the integer data compared to that for the non-integer one. Therefore, by comparing the data to the pivot, one subarray will have much higher elements than another subarray which consequently causes more unnecessary divisions. When all the elements of a K-element subarray are equal, one of the subarrays at the next division has no element and the second one has $(K - 1)$ elements. Also, in the non-integer data, increasing the sorted parts increases the elapsed time for the Quick-sort because unnecessary swaps experience longer distances.

The highest number of swaps is related to the Merge, because for this algorithm, the swaps take place first in the two-member sets and then in the adjacent parts. The second one is the Heap-sort, because it does a lot of swaps in each step to find the maximum element until it eventually reaches a single-member set. The Quick-sort has a smaller number

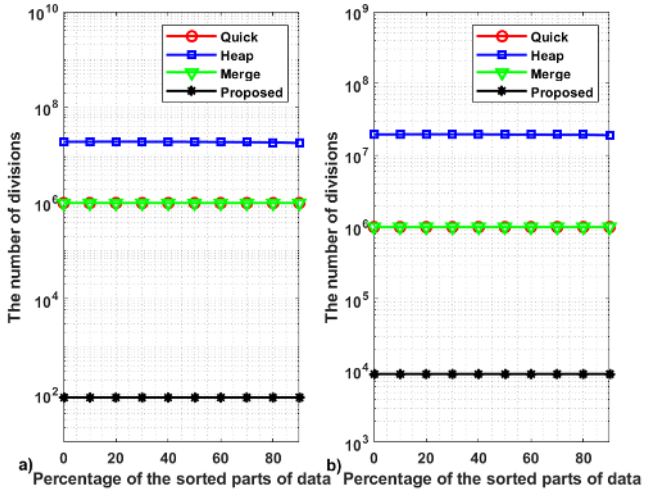

FIGURE 3. The number of divisions for partially to entirely sorted data with length 1000000, a) integer, b) non-integer.

of swaps than these two algorithms, because in each step, the data is divided into two subarrays. Also, in each step, the data is swapped only in that subarray. Although the number of swaps is less than the Merge and Heap algorithms, the running time of the Quick-sort is increased because each element is compared to a random pivot. It is decreased in the proposed algorithm because the array elements are compared to the mean-value locating almost in the middle of the data.

The Heap-sort requires the highest number of divisions because it finds one element at each step and divides over the rest to the end. The number of divisions in the proposed algorithm is reduced because, at each division, two subarrays of approximately the same length are made. If the data in each subarray is similar or sorted, there will be no further divisions or swaps. On the contrary, the Quick-sort does not care about the sorted data or the similarity rate of the data, and pushes the data to a single-member subarray. On the other hand, if they are sorted or similar in a level, some sorted data in one subarray and some in another subarray fall out of order, which causes them be sorted later.

Figs. 4, 5, and 6 show the performance of the proposed K-S mean-based algorithm compared to the Quick, Merge, and Heap algorithms for low and medium sorted primary data in the view of the NMAE when the data length is 1000000.

For the Merge-sort, especially in a significant portion of the processing time, little changes are made to the NMAE because there is no swap and/or the swaps occur in close locations, mostly in a subarray. Sometimes we have a decrease that may increase again in a few samples. In cases where data from one subarray is merged with another, there may be upward jumps. In the last steps of the algorithm, there is a sudden decrease in the value of this criterion, which is due to the nature of the Merge algorithm, i.e., until the last steps of the sorting process, we cannot guarantee the data is sorted.

In the Heap algorithm, we first see an increase in the NMAE measure, which reaches about 90%. By finding the maximum element and placing it in the exact final location,

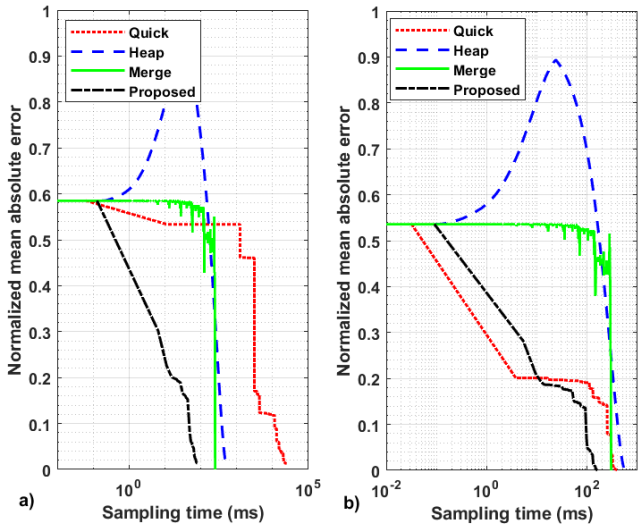

FIGURE 4. The normalized mean absolute error for 10% sorted data with length 1000000 in different sampling times, a) integer, b) non-integer.

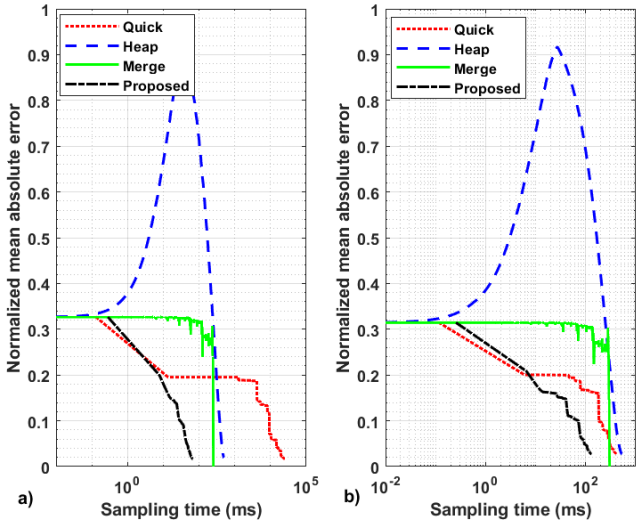

FIGURE 5. The normalized mean absolute error for 50% sorted data with length 1000000 in different sampling times, a) integer, b) non-integer.

the locations of all other elements change, which means increasing the degree of disordering. Moreover, when a new maximum element is found, this effect will be increased. This increase in the NMAE continues until the sorting result due to locating the maximum elements overcomes the disorder caused by the Heap algorithm. Then, the NMAE measure will be decreased monotonically to reach a non-zero minimum value because it is non-stable. This value is higher for integer data because similar data are more than the non-integer data. By comparing the results of the Heap algorithm for two types of data studied in Figs. 4, 5, and 6, it can be seen that the higher the percentage of the sorted primary data, the lower the peak of the NMAE curve. The peaks reach 0.88 in 90%, 0.92 in 50%, and 0.94 in 10% sorted primary data, while the data are entirely sorted after 1045, 1260, and 1400 milliseconds, respectively. In the case of non-integer data, for high, medium, and low sorted primary data, the peaks of 0.87, 0.92,

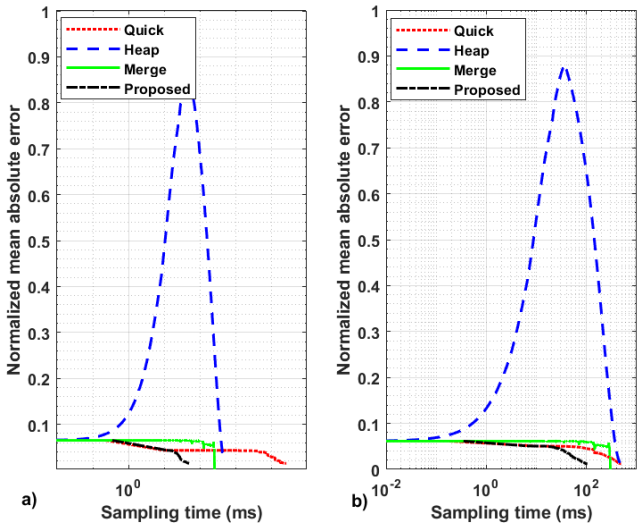

FIGURE 6. The normalized mean absolute error for 90% sorted data with length 1000000 in different sampling times, a) integer, b) non-integer.

and 0.94 and the total processing times of 1200, 2000, and 2400 are obtained, respectively.

In both Quick and K-S mean-based sorts, the NMAE measure is initially reduced exponentially on a linear scale because smaller data sets are made by dividing the data into two subarrays around a pivot which are ordered to each other, and their data locations are at most about the length of the new subarray far from their final locations. In the later steps, because the data is somewhat sorted and smaller subarrays are obtained, the changes of NMAE are less. The proposed algorithm offers a higher number of jumps because, unlike the Quick-sort that the pivot is random, it is the mean value of each array that makes two subarrays with a higher probability to be sorted. Although the new algorithm is non-stable same as the Quick and Heap sorts, it has a slightly lower NMAE floor due to the selection of a mean-based pivot located approximately in the middle of each array and also two conditions that cause the sorted parts and the parts with similar data would be extracted before reaching the next divisions.

The numerical analyses show that different algorithms experience different NMAE curves because they have different sorting procedures. Furthermore, each algorithm introduces similar trends for changes of NMAE measure in different data sets. Averaging the values obtained from 1000 times of simulation shows that the best performance belongs to K-S mean-based algorithm. Also, the proposed algorithm offers the highest decreasing slope of NMAE and the least processing time. To see more details about the changes in NMAE, the simulation results of Figs. 4, 5, and 6, are not averaged over several experiments.

V. WHY THE MEAN-BASED PIVOT IS APPROPRIATE?

Here is a criterion defined (equation 17) that shows the degree of similarity between the two subarrays in each division. We expect the most similarity if the data's median is selected as the pivot and the least resemblance if the random pivot is

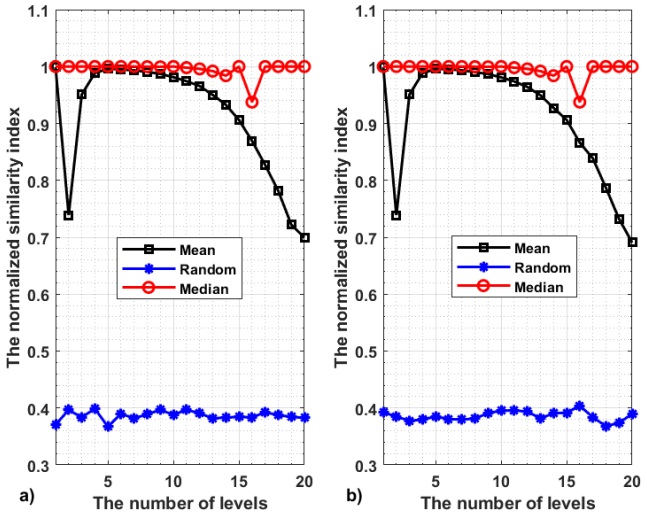

FIGURE 7. Normalized similarity index for non-integer Gaussian data using the proposed algorithm, a) low standard deviation (5), b) high standard deviation (1000).

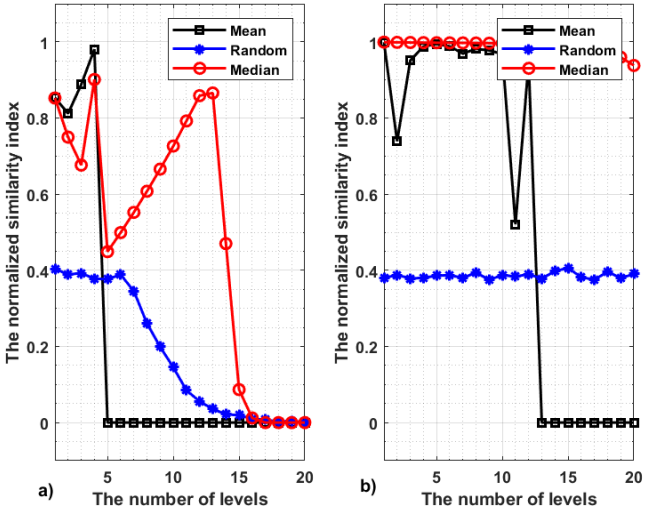

FIGURE 8. Normalized similarity index for integer Gaussian data using the proposed algorithm without the capability to detect the subarray with similar data, a) low standard deviation (5), b) high standard deviation (1000).

selected.

$$NSI(i) = \frac{|L_U(i) - L_L(i)|}{max(L_U(i), L_L(i))} \tag{17}$$

Fig. 7 shows that the closer the normalized similarity index is to 1, the fewer differences there are in the subarrays' length, and the fewer levels and divisions we get to the end of the sort. If there is no similar data, the data's length is almost halved each time we divide, so after n_d layers, we get to the single-member subarrays. For the non-integer data in the length of 1000000, we reach the divisions' end after 20 levels. It shows that in the integer data, even the use of the median does not necessarily give the exact value of 1 for NSI, because the same data in an array or when the length of an array is odd, they cause the length inequality of the subarrays.

As shown in Fig. 8, for integer data, after dividing data into several levels, the similarity tends to zero because the larger subarray that is considered as the main array in the next levels has similar data. In this case, after each division, one subarray has no data. In contrast, the next subarray has the total number of elements minus the pivot, which indicates the maximum difference in length. This action also occurs in subsequent divisions in the following levels. The number of layers and divisions it imposes is equal to the number of similar data in the array minus 1. For example, if the number of similar data in the array is 100, 99 levels with 99 times division are required. In data set with 1000000 members, 20 levels of division would suffice if there were no similar data. Therefore, to use the existence of similar elements in integer data set, the condition of having similar data in the proposed algorithm improves, and by using it, not only we do not get caught in additional layers like the Quick-sort, but also we reach sorted data in fewer levels than non-integer data.

As shown in Fig. 9, 15 levels of division in low standard deviation (5) and about 20 levels in data with high standard deviation (1000) are required. Interestingly, using the second

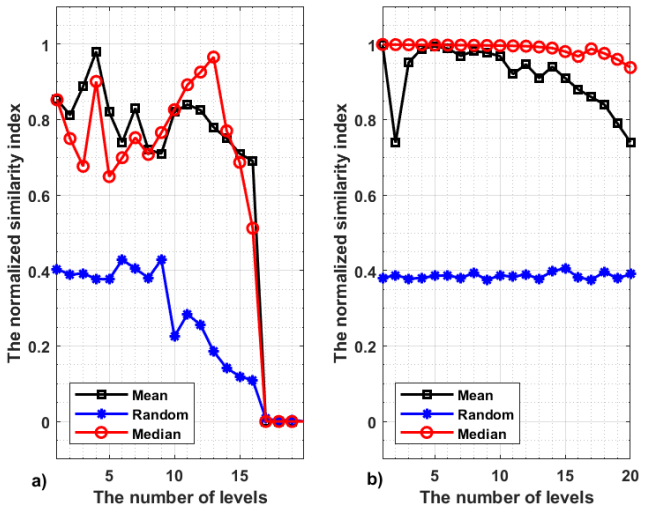

FIGURE 9. Normalized similarity index for integer Gaussian data using the proposed algorithm with the capability to detect the subarray with similar data, a) low standard deviation (5), b) high standard deviation (1000).

feature of the proposed algorithm, if the data is sorted in higher levels, even in large standard deviations, 20 levels of division may not be required. Using the proposed algorithm, if the data are similar in the larger subarray, the processing time of the smaller subarray is considered in the parallel processing, and there is no need to sort the larger subarray with similar data.

Briefly, we can find the following results:

1) In the integer data, where the possibility of similarity of the data in one subarray is high, especially in low variances, the number of divisions in the mean-based or median-based methods has decreased.
2) The proposed algorithm for high variances has also improved because in more divisions, subarrays with similar elements are encountered, which are left out.

In this way, in addition to reducing the processing time in serial processing, the similarity of the two subarrays is closer to each other, which also reduces the parallel processing time.

3) In the sorting algorithm based on the median and mean values, especially for low variances, the similarity index is improved. On the contrary, when the pivot is random, also there is a slight improvement. This is because subarrays with similar data are discarded without subsequent additional divisions, and the divisions are made into subarrays, which similar data are less likely. Therefore, the length of the subarrays becomes closer to each other.
4) In the integer data with low variance, considering the similarity of data in each subarray, after 15 divisions, they reach the sorted state. But in data with higher variance, on average, 20 divisions are needed where fluctuations are less.
5) If the data is sorted, we will still have an improvement, which will improve both the value of the similarity index and the number of the layers of division.
6) In non-integer data in low-variance and high-variance cases, there is almost the same result in the view of the similarity index because the possibility of similarity of data in different variances is approximately the same. For uniform distribution, the NSI is close to 1 in higher divisions, but in the Gaussian distribution, the NSI is always greater than 0.6.
7) The reason that the similarity index in the integer data with low variance in the median-based method is lower than 1 is due to the existence of a high number of similar data. By obtaining the median as the pivot to divide the array into two subarrays, there are identical numbers that fall into the larger subarray, reducing the value of the index. But in high variance, because the probability of similar data decreases, at each level, the likelihood of having data similar to median also decreases.
8) Downward jumps indicate duplicate data similar to pivot. It causes more data to be placed in a subarray, and the similarity index becomes smaller than 1.
9) In non-integer data, the possibility of reaching similar data is very low, so the similarity index in large subarrays with the mean-based algorithm is close to 1. In the next subarrays, the difference in length between the two subarrays is 0 or 1. As the subarrays get smaller, the difference is divided into a smaller number, which indicates the NSI is closer to 0.

For the integer data, the Quick-sort is slower than the case of non-integer data because there is a lot of similar data. It is possible to make the subarrays with different lengths that are not obtained for non-integer data. But in the proposed algorithm, this issue is prevented, so the difference that exists for the processing time in the Quick-sort between the integer and non-integer data has been eliminated in the proposed algorithm. It is noteworthy that in the random pivot, there are cases where the difference between the two subarrays reaches 100%. Interestingly, in the median-based and mean-based pivots, the difference between the two subarrays will never exceed 40%. It is an essential advantage over the Quick-sort.

The median value is the best pivot to make the subarrays with similar lengths. It needs a time-consuming process to be found, which is not acceptable. The mean value is mainly the same as the median value for symmetric distributions, while it is close to the median value for non-symmetric distributions. Natively, the mean-value involves summing up the N numbers and then dividing by N. Hence, it has linear complexity.

Herein, selection the median value as the pivot in the sorting of integer and non-integer data is just to show the upper bound of the normalized similarity index. In practice, the median cannot be used because it is necessary to have the data almost sorted, which creates a paradox. After all, our goal is to sort the data and not use a pivot that must first be found based on a sorting algorithm. Finding the median requires sorting the data, which involves the complexity of $O(NlogN)$. In this research, the mean value is used instead, which requires $O(N)$ time complexity if the actual mean of N-element data is obtained.

Another idea given in this article is that in large data, the approximate mean can be obtained using a limited number of data (M) that experiences lower complexity. For example, in this study, 100 points with a random position were used for different layers whose data length is more significant than 100. Thus, in the first average, which is related to data with a length of 1000000, we only need to average 100 numbers; that, with a factor of 10000, we will reduce the calculations and improve the processing time. Figs. 10, 11 show the NSI of the exact and 100-point approximate means in both integer and non-integer data. It can be seen that for the data with a length of 1000000, it is sufficient to select 100 samples in each level, and the mean and median values are close to each other.

As shown in Figs. 10 and 11, first the mean, approximate mean, and median values are almost zero. They experience a large jump to a positive (shown in Fig. 10) or negative value (shown in Fig. 11) and gradually return to a number close to zero. It is due to this fact that it first changes from a symmetric Gaussian distribution to an asymmetric one-sided Gaussian distribution. Then, in subsequent divisions, it gradually tends to form a more uniform histogram, which is shown in Figs. 12 and 13 for four layers of division. In these figures, the data probability density function curves are plotted in the first, second, third, and fourth layers, respectively, for the larger length subarrays.

Fig. 12 shows a case where the number of data above zero is slightly more significant than the number of data below zero. Therefore, in the second layer, the upper half is chosen. In subsequent divisions, the subarray is selected close to zero because the frequency of data around zero is higher. This is why in the figure for the mean value of different layers, the mean value first suddenly jumps from zero to a

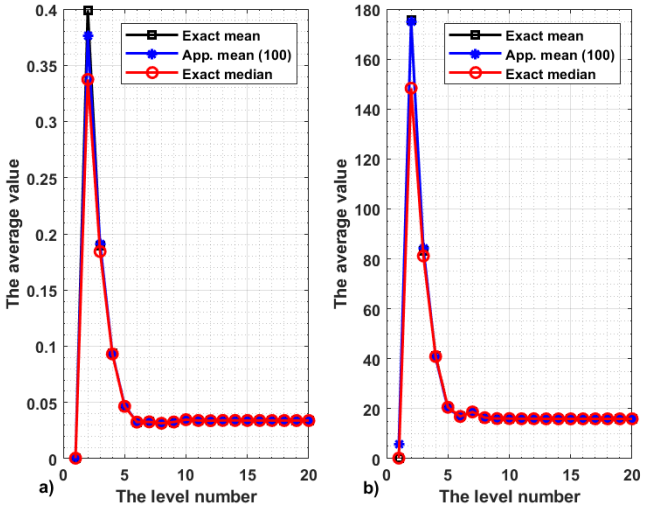

FIGURE 10. Median, exact mean, and 100-element approximate mean values for non-integer Gaussian 1000000-element data in different levels of division, a) low-standard deviation, b) high-standard deviation.

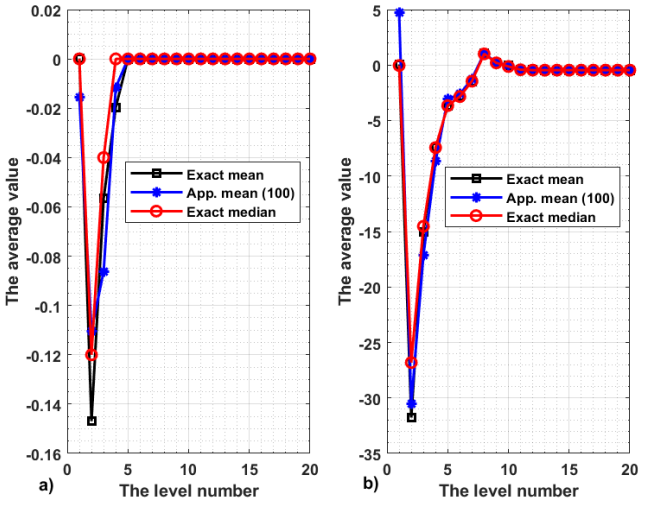

FIGURE 11. Median, exact mean, and 100-element approximate mean values for integer Gaussian 1000000-element data in different levels of division, a) low-standard deviation, b) high-standard deviation.

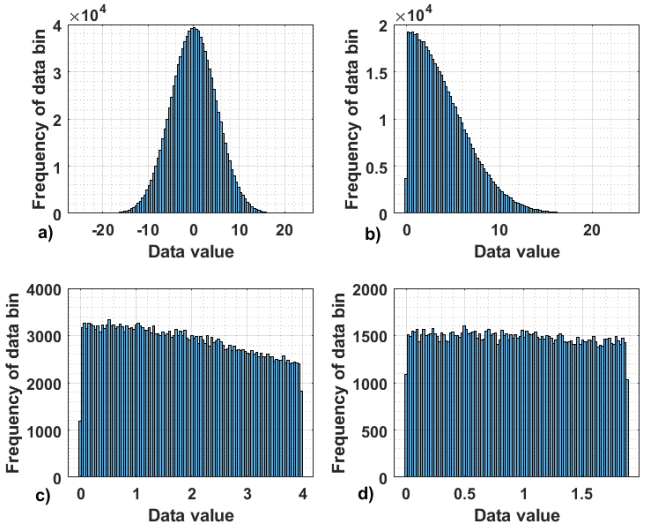

FIGURE 12. Gaussian data with low standard deviation (5), a) the first level, b) second level, c) third level, d) fourth level.

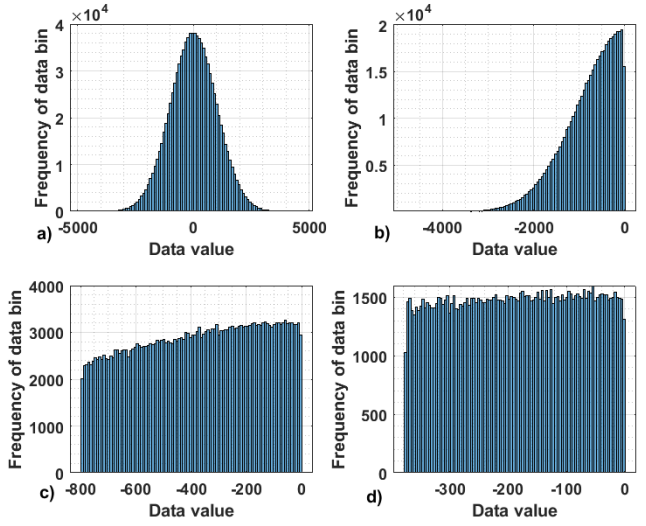

FIGURE 13. Gaussian data with high standard deviation (1000), a) the first level, b) second level, c) third level, d) fourth level.

considerable value (positive or negative) and then tends to a number around zero. Fig. 13 shows the case where the number of data less than zero is slightly higher. Therefore, in the second layer, the data smaller than zero are in the larger subarray. In both figures, the probability density function of the data of different subarrays of each layer is more uniform than the previous layer. The results obtained in Figs. 12 and 13 are valid for both Gaussian and uniform data distributions with different standard deviations.

VI. COMPLEXITY ANALYSIS OF THE PROPOSED ALGORITHM

The proposed algorithm is unlike the Quick-sort, which experiences $O(NlogN)$ time complexity at best and average cases and $O(N^2)$ at the worst-case [30]. Using the six new ideas proposed in this investigation, the number of swaps, divisions, and processing time reduce, i.e., lower complexity than the Quick-sort. The worst-case for the Quick-sort occurs when the pivot divides the data into two subarrays, one with zero length and another with the total length of the data minus one, at each division level. If this happens for the proposed algorithm, or if the data in an array are the same or sorted, it will not enter the next division. In these two cases, the proposed algorithm only needs to calculate the mean value of the data and then check the similarity or order of the data, which will be an operation with linear time complexity, $O(N)$. On the other hand, the worst-case for the proposed algorithm occurs when the divisions continue until single-member subarrays. In each division, the two subarrays have the greatest possible difference.

In Section 5, we proved theoretically and numerically that using the mean-based pivot, the maximum difference between the length of subarrays in each division is about 40%. Also, in the average-case of the proposed algorithm,

the length of the two subarrays is almost equal, and therefore, we will have fewer levels and divisions than the Quick-sort. In general, the time taken by the proposed K-S mean-based sort is as follows:

$$T(N) = T(m) + T(N-m) + T_B(N) + T_M(N) + T_P(N) \tag{18}$$

The first two times are for two recursive calls, the last three times respectively are for obtaining the Boolean variables 1 and 2, evaluating the mean value, and partitioning an array into two subarrays, which is each of them in order of $\Theta(N)$. As we know

$$\Theta(N) + \Theta(N) + \Theta(N) + \Theta(N) = \Theta(N) \tag{19}$$

Hence,

$$T(N) = T(m) + T(N-m) + \Theta(N) \tag{20}$$

m is the number of subarray elements whose elements are equal or greater than the mean value, and $(N-m)$ is the remainder elements, located in the second subarray that are smaller than the mean value. Without loss of generality, it is assumed that the mean value is not equal to the value of any data in the array. In the following subsections, we use Big O (O), Omega (Ω), and Theta (Θ), which describe the upper bound, the lower bound, and the exact bound of the complexity [46], [47], respectively.

A. THE BEST-CASE

When the data in an array are similar, or sorted in an ascending or descending order, the best-case occurs. In this case, we need an $O(N)$ process for evaluating the mean value of the primary array, $O(N)$ for the comparison process in partitioning, and $2O(N)$ for obtaining two Boolean variables. The first Boolean variable indicates that the data are similar, and the second one demonstrates that the elements are in an increasing or decreasing order. If the first Boolean variable is true, means that all elements are similar. If the second Boolean variable is true, i.e., data are sorted. Hence, we have four operations with linear complexity, one for making the mean value, one for comparisons to the mean value, and two others for obtaining the Boolean variables. Hence, the upper bound of the complexity order for the best-case is linear as

$$T_{best}(N) = \Omega(N) \tag{21}$$

B. THE AVERAGE-CASE

The average-case occurs when the partition process always divides the primary array into two subarrays around the mean value, similar in length. It is like the best-case for the Quick-sort. In the average-case it is clear that

$$T_{av}(N) = 2T_{av}(\frac{N}{2}) + \Theta(N) \tag{22}$$

In this case, we have one N-element array in the first level, two $\frac{N}{2}$-element subarrays in the second level, four $\frac{N}{4}$-element subarrays in the third level, and so on. Finally, it has 2^{n_d} number of single-element subarrays in the n_dth level of division as the last one. Considering $\frac{N}{2^{n_d}}$ equals 1, n_d satisfies the following equation.

$$n_d = log_2 N \tag{23}$$

Knowing that $log_2\, N$ is the same as $logN$ in the complexity analysis, therefore,

$$N + 2 \times \frac{N}{2} + 4 \times \frac{N}{4} + \ldots + 2^{n_d} \times 1 = n_d.N = NlogN \tag{24}$$

Also, for evaluating the mean value, comparisons to make the subarrays, and obtaining the Boolean variables, we need similar complexities as $\{\Theta(N), 2\Theta(\frac{N}{2}), 4\Theta(\frac{N}{4}), \ldots, 2^{n_d}\Theta(1)\}$ that totally is in the order of $\Theta(NlogN)$. Therefore, the complexity order of the proposed mean-based algorithm for the average-case is as

$$T_{av}(n) = \Theta(NlogN) \tag{25}$$

C. THE WORST-CASE

According to the simulation results reported in Figs. 7, 9, based on the normalized similarity index of the integer/non-integer Gaussian data in low and high standard deviations, the least similarity between the lengths of the subarrays in each level of division is about 60% which means about 40% difference. Assuming that the length of the upper subarray is larger than that for the lower subarray, we have

$$L_U - L_L = 0.4L_U \tag{26}$$
$$L_U + L_L = N \tag{27}$$

By solving these two equations, we have

$$L_U = \frac{5}{8}N, \quad L_L = \frac{3}{8}N \tag{28}$$

For this case, the complexity order can be found by solving the following equation

$$T_{worst}(N) = T_{worst}(\frac{5}{8}N) + T_{worst}(\frac{3}{8}N) + \Theta(N) \tag{29}$$

Based on [46], [47], we conclude that

$$T_{worst}(N) = O(NlogN) \tag{30}$$

VII. COMPARISON IN DIFFERENT VIEWPOINTS

Briefly, Table 3 demonstrates a comparison between the proposed K-S mean-based sorting algorithm and the popular sorting algorithms such as Quick, Merge, Heap, Insertion, Selection, and Bubble, in the view of the time complexity, space complexity, stability, adaptivity, type of swaps (in-place or not), detectability of a part with similar data, detectability of a part with sorted data, ability to sort data gradually, making parallel independent subarrays, and applicability for low, medium, or large data sets.

As shown in Table 3, although the proposed algorithm has a similar time complexity to the Merge-sort in the worst-case

TABLE 3. Comparisons of the proposed K-S mean-based algorithm with popular sorting algorithms.

Sorting algorithm / Metric		Proposed K-S mean-based	Quick	Merge	Heap	Insertion	Bubble	Selection
Time complexity	Best-case	$\Omega(N)$	$\Omega(N \log N)$	$\Omega(N \log N)$	$\Omega(N \log N)$	$\Omega(N)$	$\Omega(N)$	$\Omega(N^2)$
	Average-case	$\theta(N \log N)$	$\theta(N \log N)$	$\theta(N \log N)$	$\theta(N \log N)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$
	Worst-case	$O(N \log N)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Memory complexity		$O(N \log N)$	$O(N \log N)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
In-place		Yes	Yes	No	Yes	Yes	Yes	Yes
Stability		Moderate	No	Yes	No	Yes	Yes	No
Adaptivity		Moderate	No	Yes	No	Yes	Yes	Yes
Suitable data length		Medium and large	Medium and large	Medium and large	Medium and large	Medium and small	Medium and small	Medium and small
Detectability of the parts with similar elements		Yes	No	No	No	No	No	No
Detectability of the parts with sorted elements		Yes	No	No	No	No	No	No
Ability to sort data gradually		Yes	Yes	No	Yes	No	No	Yes
Ability to make Independent subarrays		Yes, with a similar length	Yes	No	No	No	No	No

and average-case, performs better than the Quick and Merge sorts in the best-case, with a linear time complexity. This algorithm has a memory complexity similar to the Quick-sort. The proposed algorithm, same as the Quick-sort, qualifies as an in-place sorting algorithm as it uses extra space only for sorting recursive function calls but not for manipulating the input. It is applicable for sorting three types of data, especially for medium and large data sets. Moreover, it can detect the part with identical elements and distinguish the part with sorted data. Besides, it can gradually sort the data and make independent subarrays with approximately similar lengths. All features of the proposed algorithm are valid and applicable for different data types (Gaussian and uniform random distributions) and a wide range of data's standard deviation from low to very high.

VIII. CONCLUSION

In this article, we proposed an efficient mean-based sorting algorithm that distinguishes sorted subarrays and those that have similar elements. The difference between the lengths of the left and right subarrays was analyzed. A measure to evaluate the difference between the locations of the sorted and unsorted data, namely NMAE, was proposed. Regarding the processing time, the number of swaps and divisions, and NMAE, the effectiveness of the proposed algorithm to the Merge, Quick, and Heap sorts for both integer and non-integer data was shown.

We showed the main drawback of the Quick-sort, i.e., making independent subarrays with non-similar lengths in each division due to the randomness of the pivot. We improved it in the proposed algorithm by considering a mean-based pivot. Moreover, we decreased the number of unnecessary swaps and divisions by adding an ability to detect the sorted part and similar data. As the future work, it is considered to combine the results of this work and those reported in [38], [39], [43], [44] to realize the K-S sorting in the parallel processing.

Each subarray can be processed without any interaction with the other ones because they are independent. If the data of one subarray is more important, it can be sorted first, and then we can do the sorting for the rest of the subarrays later. As the final remark, the proposed K-S mean-based sort is a proper algorithm to extract the median or a specific number of min./max. values faster than the conventional methods.

REFERENCES

[1] Z. Shen, X. Zhang, M. Zhang, W. Li, and D. Yang, "Self-sorting-based MAC protocol for high-density vehicular ad hoc networks," *IEEE Access*, vol. 5, pp. 7350–7361, 2017.

[2] G. Maier, F. Pfaff, C. Pieper, R. Gruna, B. Noack, H. Kruggel-Emden, T. Längle, U. D. Hanebeck, S. Wirtz, V. Scherer, and J. Beyerer, "Experimental evaluation of a novel sensor-based sorting approach featuring predictive real-time multiobject tracking," *IEEE Trans. Ind. Electron.*, vol. 68, no. 2, pp. 1548–1559, Feb. 2021.

[3] M. Haggag, S. Abdelhay, A. Mecheter, S. Gowid, F. Musharavati, and S. Ghani, "An intelligent hybrid experimental-based deep learning algorithm for tomato-sorting controllers," *IEEE Access*, vol. 7, pp. 106890–106898, 2019.

[4] C. Ni, Z. Li, X. Zhang, X. Sun, Y. Huang, L. Zhao, T. Zhu, and D. Wang, "Online sorting of the film on cotton based on deep learning and hyperspectral imaging," *IEEE Access*, vol. 8, pp. 93028–93038, 2020.

[5] M. Nati, S. Mayer, A. Capossele, and P. Missier, "Toward trusted open data and services," *Internet Technol. Lett.*, vol. 2, no. 1, pp. 1–5, 2018.

[6] X. Huang, Z. Liu, and J. Li, "Array sort: An adaptive sorting algorithm on multi-thread," *J. Eng.*, vol. 2019, no. 5, pp. 3455–3459, May 2019.

[7] Y. Chauhan and A. Duggal, "Different sorting algorithms comparison based upon the time complexity," *Int. J. Res. Anal. Rev.*, vol. 7, no. 3, pp. 114–121, Sep. 2020.

[8] B. S. Khan and M. A. Niazi, "Emerging topics in Internet technology: A complex networks approach," *Internet Technol. Lett.*, vol. 1, no. 4, pp. 1–6, 2018.

[9] H. Peng, Y. Xiao, Y. N. Ruyue, and Y. Yifei, "Ultra dense network: Challenges, enabling technologies and new trends," *China Commun.*, vol. 13, no. 2, pp. 30–40, Feb. 2016.
[10] P. Pirinen, "A brief overview of 5G research activities," presented at the 1st Int. Conf. 5G Ubiquitous Connectivity, Levi, Finland, Nov. 2014.
[11] W. Saad, M. Bennis, and M. Chen, "A vision of 6G wireless systems: Applications, trends, technologies, and open research problems," *IEEE Netw.*, vol. 34, no. 3, pp. 134–142, May/Jun. 2020.
[12] S. Z. Iqbal, H. Gull, and A. W. Muzaffar, "A new friends sort algorithm," presented at the 2nd IEEE Int. Conf. Comput. Sci. Inf. Technol., Beijing, China, Aug. 2009.
[13] I. Hayaran and P. Khanna, "Couple sort," in *Proc. 4th Int. Conf. Parallel, Distrib. Grid Comput.*, Waknaghat, India, Dec. 2016, pp. 390–393.
[14] S. M. Cheema, N. Sarwar, and F. Yousa, "Contrastive analysis of bubble & merge sort proposing hybrid approach," presented at the 6th Int. Conf. Innov. Comput. Technol., Dublin, Ireland, Aug. 2016.
[15] A. Alotaibi, A. Almutairi, and H. Kurdi, "One by one (OBO): A fast sorting algorithm," presented at the 15th Int. Conf. Future Netw. Commun., Leuven, Belgium, Aug. 2020.
[16] A. H. Elkahlout and A. Y. A. Maghari, "A comparative study of sorting algorithms: Comb, cocktail and counting sorting," *Int. Res. J. Eng. Technol.*, vol. 4, no. 1, pp. 1387–1390, Jan. 2017.
[17] J. Alnihoud and R. Mansi, "An enhancement of major sorting algorithms," *Int. Arab J. Inf. Technol.*, vol. 7, no. 1, pp. 55–61, Jan. 2010.
[18] *Geeksforgeeks*. Accessed: Nov. 11, 2020. [Online]. Available: https://www.geeksforgeeks.org
[19] *Programix*. Accessed: Nov. 15, 2020. [Online]. Available: https://www.programix.com
[20] S. S. Moghaddam, M. Shirvanimoghaddam, and A. Habibzadeh, "Clustering-based handover and resource allocation schemes for cognitive radio heterogeneous networks," presented at the 28th Int. Telecommun. Netw. Appl. Conf., Sydney, NSW, Australia, Nov. 2018.
[21] S. S. Moghaddam and M. Ghasemi, "A low-complex/high throughput resource allocation for multicast D2D communications," presented at the 7th Int. Conf. Comput. Commun. Eng., Kuala-Lumpur, Malaysia, Sep. 2018.
[22] S. S. Moghaddam, "Introductory: Primary and secondary users in cognitive radio based wireless communication systems," in *Cognitive Radio in 4G/5G Wireless Communication Systems*. London, U.K.: IntechOpen, 2018, ch. 1, pp. 1–12.
[23] S. Mishra, S. Saha, and S. Mondal, "Divide and conquer based non-dominated sorting for parallel environment," presented at the IEEE Congr. Evol. Comput., Vancouver, BC, Canada, Jul. 2016.
[24] M. Yan, W. Shang, and M. Zhang, "The analysis of coordinate-recorded merge-sort based on the divide-and-conquer method," presented at the 15th Int. Conf. Comput. Inf. Sci., Okayama, Japan, Jun. 2016.
[25] S. S. Moghaddam and K. S. Moghaddam, "Efficient base-centric/user-centric clustering algorithm based on thresholding and sorting," presented at the 14th Int. Conf. Innov. Inf. Technol., AlAin, UAE, Nov. 2020.
[26] Y. Yang, P. Yu, and Y. Gan, "Experimental study on the five sort algorithms," presented at the 2nd Int. Conf. Mech. Automat. Control Eng., Hohhot, China, Jul. 2011.
[27] M. S. Rana, M. A. Hossin, S. M. H. Mahmud, H. Jahan, A. K. M. Z. Satter, and T. Bhuiyan, "MinFinder: A new approach in sorting algorithm," *Procedia Comput. Sci.*, vol. 154, pp. 130–136, Jan. 2019.
[28] G. Kocher and N. Agrawal, "Analysis and review of sorting algorithms," *Int. J. Sci. Eng. Res.*, vol. 2, no. 3, pp. 81–84, Mar. 2014.
[29] K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani, and N. I. Zanoon, "Review on sorting algorithms: A comparative study," *Int. J. Comput. Sci. Secur.*, vol. 7, no. 3, pp. 120–126, 2013.
[30] W. Xiang, "Analysis of the time complexity of quick sort algorithm," presented at the Int. Conf. Inf. Manage., Innov. Manage. Ind. Eng., Shenzhen, China, Nov.2011.
[31] D. N. Raju, "An efficient new approach mean based sorting," presented at the IEEE UP Sect. Conf. Elect. Comput. Electron., Allahabad, India, Dec. 2015.
[32] W. H. Butt and M. Y. Javed, "A new relative sort algorithm based on arithmetic mean value," presented at the IEEE Int. Multitopic Conf., Karachi, Pakistan, Dec. 2008.
[33] M. Shabaz and A. Kumar, "SA sorting: A novel sorting technique for large-scale data," *J. Comput. Netw. Commun.*, vol. 2019, pp. 1–7, Jan. 2019.
[34] P. C. Roy, K. Deb, and M. M. Islam, "An efficient nondominated sorting algorithm for large number of fronts," *IEEE Trans. Cybern.*, vol. 49, no. 3, pp. 859–869, Mar. 2019.
[35] P. Gupta, "Conventional vs enhanced sorting algorithm: A review," *Int. J. Res. Sci. Innov.*, vol. 5, no. 1, pp. 120–127, 2018.
[36] S. K. Gill, V. P. Singh, P. Sharma, and D. Kumar, "A comparative study of various sorting algorithms," *Int. J. Adv. Stud. Sci. Res.*, vol. 4, no. 1, pp. 367–372, 2018.
[37] L. Khreisat, "A survey of adaptive quicksort algorithms," *Int. J. Comput. Sci. Secur.*, vol. 12, no. 1, pp. 1–10, 2018.
[38] Z. Marszałek, M. Wo niak, and D. Połap, "Fully flexible parallel merge sort for multicore architectures," *Complexity*, vol. 2018, pp. 1–19, Dec. 2018.
[39] Z. Marszalek, "Parallelization of modified Merge sort algorithm," *Symmetry*, vol. 9, no. 176, pp. 1–18, 2017.
[40] A. Maus, "A faster all parallel mergesort algorithm for multicore processors," presented at the Norwegian Inform. Conf., Oslo, Norway, Aug. 2018.
[41] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," presented at the ACM Int. Conf. Companion Object Oriented Program. Syst., Lang. Appl., Portland, OR, USA, Oct. 2011.
[42] D. Jimenez-Gonzalez, J. J. Navarro, and J. L. Larriba-Pey, "The effect of local sort on parallel sorting algorithms," presented at the 10th EuroMicro Workshop Parallel, Distrib. Netw.-Based Process., Canary Islands, Spain, 2002.
[43] V. Prifti, R. Bala, R. Tafa, D. Saatciu, and J. Fajzaj, "The time profit obtained by parallelization of quicksort algorithm used for numerical sorting," presented at the Sci. Inf. Conf., London, U.K., 2015.
[44] Z. Marszalek, "Parallel fast sort algorithm for secure multiparty computation," *J. Universal Comput. Sci.*, vol. 24, no. 4, pp. 488–514, 2018.
[45] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, 4th ed. New York, NY, USA: McGraw-Hill, 2002.
[46] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, Sep. 1980.
[47] Y. Chee, "A real elementary approach to the master recurrence and generalizations," presented at the 8th Annu. Conf. Theory Appl. Models Comput., Tokyo, Japan, May 2011.

SHAHRIAR SHIRVANI MOGHADDAM (Senior Member, IEEE) was born in Khorramabad, Iran, in 1969. He received the Ph.D. degree in electrical engineering from the Iran University of Science and Technology, Tehran, Iran, in 2001. Since 2003, he has been with the Faculty of Electrical Engineering, Shahid Rajaee Teacher Training University, Tehran, Iran, where he is currently an Associate Professor. He has numerous articles in prestigious scientific journals. He has presented dozens of articles in national and international conferences. He has authored two books, one on digital communications and another on electrical engineering, and edited one book about cognitive radio (CR). His research interests include resource allocation and power control in CR-based networks, ultra-dense networks, heterogeneous networks, device-to-device and vehicle-to-vehicle communications, and digital array processing.

KIAKSAR SHIRVANI MOGHADDAM (Student Member, IEEE) was born in Tehran, Iran, in 2000. He is currently pursuing the B.Sc. degree in computer engineering with the School of Computer Engineering, Iran University of Science and Technology. He is currently a Teaching Assistant with the Iran University of Science and Technology. He has been a lecturer of some courses about C and C++ programming languages, LATEX software, fundamentals of computer engineering, and Rubik's cube. His activities are focused on the development of Web and windows applications and algorithm design in cross-platform languages.