IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 25, 2021, accepted January 19, 2022, date of publication January 25, 2022, date of current version February 1, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3145981

A General Framework for Sorting Large Data Sets Using Independent Subarrays of Approximately Equal Length

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 (SRTTU), Tehran 16788-15811, Iran
[2]School of Computer Engineering, Iran University of Science and Technology (IUST), Tehran 16846-13114, Iran

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

This work was supported by Shahid Rajaee Teacher Training University (SRTTU) under Contract 1304.

ABSTRACT Designing an efficient data sorting algorithm that requires less time and space complexity is essential for computer science, different engineering disciplines, data mining systems, wireless networks, and the Internet of things. This paper proposes a general low-complex data sorting framework that distinguishes the sorted or similar data, makes independent subarrays approximately in equal length, and sorts the subarrays' data using one of the popular comparison-based sorting algorithms. Two frameworks, one for serial realization and another for parallel realization, are proposed. The time complexity analyses of the proposed framework demonstrate an improvement compared to the conventional Merge and Quick sorting algorithms. Following complexity analysis, the simulation results indicate slight improvements in the elapsed time and the number of swaps of the proposed serial Merge-based and Quick-based frameworks compared to the conventional ones for low/high variance integer/non-integer data sets, in different data sizes and the number of divisions. It is about $(1-1.6\%)$ to $(3.5-4\%)$ and $(0.3-1.8\%)$ to $(2-4\%)$ improvements in the elapsed times for 1, 2, 3, and 4 divisions, respectively for small and very large data sets in Merge-based and Quick-based scenarios. Although these improvements in serial realization are minor, making independent low-variance subarrays allows the sorted components to be extracted sequentially and gradually before the end of the sorting process. Also, it proposes a general framework for parallelizing conventional sorting algorithms using non-connected (independent) or connected (dependent) multi-core structures. As the second experiment, the numerical analyses that compare the results of the parallel realization of the proposed framework to the serial one in 1, 2, 3, and 4 divisions, show a speedup factor of $(2-4)$ for small to $(2-16)$ for very large data sets. The third experiment shows the effectiveness of the proposed parallel framework to the parallel sorting based on the random-access machine model. Finally, we prove that the mean-based pivot is as efficient as the median-based and much better than the random pivot for making subarrays of approximately equal length.

INDEX TERMS Data sorting, Gaussian, integer, merge, multi-core, non-integer, parallel realization, quick, Rayleigh, time and space complexity, uniform.

I. INTRODUCTION

Sorting is the essential process of ordering the non-sorted data (information) and changing the position of elements in a data set, increasingly or decreasingly [1]. In this paper, we are dealing with three questions and related answers:

- What is a data set? It is a set including natural, integer, or real numbers, letters and words made by the alphabet, or specific symbols and properties. It can be categorized from different aspects, such as numerical and non-numerical, integer and non-integer, random (uniform, Gaussian, Rayleigh) and non-random (almost sorted, somewhat sorted), and small, medium, and large.
- What is sorting? Sorting is the process of rearranging unordered data in descending order (large to small) or ascending (small to large).
- How to sort data? Change the position of elements in a data set, which can be an array, list, vector, or matrix.

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Various examples of databases sorted by numbers, characters, symbols, or any combination of them [2], [3] are:

- Phone-book sorted by name;
- Color codes in paint factories;
- Dictionary sorted from A to Z;
- Bank account numbers and card numbers;
- Car license plates in the police department;
- Student numbers in schools and universities;
- Usernames and passwords for different emails and sites;
- National codes and identification numbers in the civil registry office;
- Passport numbers in the central police department and immigration databases.

Different applications of data sorting in science and engineering [4], [5] are:

- Recovering lost data;
- Detecting similar data;
- Finding the median of the data;
- Faster data retrieval and search;
- Speeding up the clustering process;
- Accelerating the classification process;
- Obtaining the frequency and histogram of the data;
- Determining the data dynamics and dynamic range;
- Extracting the most prominent and smallest amounts of data;
- Assist in the process of deleting data below or above a threshold level.

Also, new applications for large data sets [6]–[11], include:

- Social media;
- Search engines;
- Internet of things;
- Machine learning;
- Ultra-dense networks;
- Big data and Massive data sets;
- Data mining, filtering, and cleaning;
- Massive multi-input-multi-output (MIMO) systems.

Data sorting algorithms can be categorized into two major groups, such as comparison-based and non-comparison-based. Some of them are based on the divide-and-conquer method, and a part of them can be realized recursively [12]. Most of the data sorting algorithms are based on comparisons like those considered in this investigation. Insertion, Bubble, Selection, Merge, Quick, Heap, Shell, Tim, Comb, Cycle, Cocktail, Strand, Binary Insertion, Tree, Cartesian Tree, and Even-odd sorting algorithms belong to the comparison-based category [13], [14]. In contrast, Radix, Counting, Bucket, Pigeonhole, and Tag are non-comparison-based algorithms applicable for sorting integer data [15]. From different points of view, various classes of sorting algorithms [12]–[15] are summarized as:

- Type of realization: Serial and parallel;
- How do the swaps: In-place and non-in-place;
- Type of algorithm code: Recursive and non-recursive;
- Application: Specific-application and general-application;
- Sorting class: Comparison-based and non-comparison-based;
- Adaptivity (Taking advantage of already sorted elements in the list): Adaptive and non-adaptive;
- Stability (Appearing two similar elements in the same order in the sorted output as they appear in the input array): Stable and non-stable.

To evaluate and compare the performance of data sorting algorithms, several metrics are introduced [2], [16], such as:

- Time complexity;
- Memory space complexity;
- Number of comparisons and swaps;
- The elapsed (processing) time of sorting process;
- Number of divisions in divide-and-conquer methods;
- Sensitivity of the method to the type and size of data;
- How much the sorting is gradual in the serial realization;
- Possibility of multi-core parallel processing in connected or non-connected multi-core scenarios.

Many sorting algorithms have been enhanced in time and space complexity, type of realization, and so on [17]. To sort large data sets, the processing time of a large number of records is considerable, and complexity cannot be ignored [18]. Therefore, designing an efficient sorting approach is required [19] that takes less time and space complexity [20] and comprises comparison, swapping, and assignment operations [21]. Also, the simplicity of the algorithm, its computational complexity, the difference in algorithm execution time for unsorted and somewhat sorted data, error propagation in the process in case of interruption or error in a part, the memory required for the algorithm, and its application, indicate the need to update sorting algorithms [22], [23].

The two most commonly used sorting algorithms, Quick and Merge, are divide-and-conquer methods, which work well for unordered medium, large, and very large data sets. The Quick-sort makes two separate subarrays in each division, while the Merge-sort divides each array into dependent subarrays with equal lengths. Quick-sort selects an element as a pivot and divides the given array around the pivot, one greater and another smaller. This segmentation continues until it finds single-member subarrays [23], [24]. Although Quick-sort is an in-place sorting, in some cases, a stack overflow error is encountered. In contrast, Merge-sort that is not an in-place sort [1], [13], using mergesort function recursively divides the array into two halves and calls the merge function for each to combine both halves [19], [25].

Despite the Merge and Quick algorithms being widely used in many applications, they have drawbacks. The Merge-sort does not allow gradual sorting and cannot be processed in parallel using distinct processors. In contrast, though the Quick-sort allows parallel processing using stand-alone processors, it is not time-efficient because the subarrays are not in equal length. Thus, the leading question of this investigation is this: Is it possible and how to do the following items by proposing a new low-complex framework applicable for popular comparison-based sorting algorithms?

- Detect the sorted data (entirely or partly) and similar data in different divisions.
- Make independent subarrays of the primary data approximately in the same lengths.
- Sort data gradually before ending the sorting process.
- Create a general framework for the serial and parallel realization of sorting algorithms efficiently.
- Find a new sorting procedure with similar complexity for different data types and sizes.
- Hold the stability and adaptivity features for the improved versions of the stable and adaptive algorithms.
- Increase the stability and adaptivity for the improved versions of the non-stable and non-adaptive algorithms.
- Improve the performance of sorting algorithms, such as Merge and Quick, that are easy to implement.

This paper is structured as follows. Section II looks at the related work and shows why a new framework is needed. Section III illustrates the proposed framework as a serial gradual or parallel multi-core scenario applicable for different sorting algorithms. More details are also demonstrated in pseudo-codes 1, 2, and figures 1, 2, for serial and parallel realizations. A comprehensive time complexity analysis of the proposed framework in different cases, such as average, best, and worst, for serial and parallel realizations is done in Section IV. In addition, it is compared to the time complexity of the conventional Quick and Merge sorting algorithms. Section V presents numerical analyses and shows the superiority of the proposed framework. This section includes three subsections, one for serial realization, another for parallel realization, and the last one presents some highlights of the theoretical and numerical analyses. The following section, Section VI, compares the proposed sorting framework in parallel realization to the recently published sorting algorithm based on the parallel random-access machine model. Section VII proves the effectiveness of the mean-based divisions in symmetric and asymmetric distributions. Section VIII compares the conventional and proposed algorithms from different viewpoints. Finally, Section IX concludes the paper, and Section X introduces future work related to the proposed framework.

II. RELATED WORK AND WHY A NEW FRAMEWORK IS REQUIRED

Nowadays, there is a large number of data in the heterogeneous networks (HetNets), Internet of things (IoT), wireless sensor networks (WSNs), social media, signal/information processing, and data fusion in social networks, as well as advanced communication, localization and object tracking systems [26]–[30]. The existence of various sorting algorithms, numerous databases, and the growing need for sorting, categorizing, prioritizing, and searching, well illustrates the importance of data sorting. Also, increasing the number of users in communication and Internet networks and gathering information from large databases, justifies the necessity to propose a low-complex time-efficient sorting algorithm.

Research works in [19], [21], [32]–[39], and [31], [40] show that data sorting is a multidisciplinary subject at the edge of science. Some issues have led to the need to modify existing and propose new data sorting algorithms, make restrictions in the data sorting process, performance analysis of algorithms, suggest new criteria for evaluating algorithms, new implementations of the conventional algorithms, and improve the performance of the existing algorithms with the increasing advances of human beings in various fields of science and engineering, especially computer science and electrical engineering [19], [25], [31]–[38], [40]–[48].

The first category deals with using parallel processing instead of serial realization of sorting algorithms, proposing ideas for efficient memory use and central processing unit (CPU), and decreasing the number of swaps and interactions between the processors. Ref. [41] proposes a solution to reduce the running time and the number of swaps by the parallel realization and efficient use of memory. It presents an efficient parallel recursive divide-and-conquer algorithm applicable for Bubble, Selection, and Insertion sorting algorithms. The main feature of this algorithm is high data locality and highly parallel realization. The time complexity is $O(N^{log3})$ instead of $O(N^2)$ of the conventional Insertion-sort. This algorithm has not been suggested and tested for large data sets and other popular sorting algorithms like Merge and Quick. In [42], it is shown that using multiple passes over the data set, has resulted in a significant improvement in the number of swaps and reducing the sorting time. The results show the superiority of the proposed technique for CPU-only and hybrid CPU–GPU implementations. Hence, in large files that the swapping time is dominant, algorithms that minimize the swapping operations usually are superior to those which only focus on CPU time optimizations. Ref. [43] involves the random access memory (RAM)-based sorting, which makes use of multiple read and write characteristics of RAM. To sort one-dimensional data, it performs sequential search and swap operations simultaneously on numbers to reduce the cost of the implementation. Also, there is no additional hardware logic to sort and store, using the minimum memory and at most two registers. It has no attention on the type of sorting algorithm and just proposes an idea to have time-efficient and cost-efficient implementations. In the paper [44], the authors use the dual-core Window-based platform to study the effect of parallel processes number and the number of cores on the performance of three message passing interfaces (MPI) parallel implementations for some sorting algorithms. It is done by simulations, and more experiments are needed to verify the results for large data sets and different sorting algorithms. In [45], two solutions to use memory in shared-memory architectures are proposed. The first solution is based on a data structure (the interval tree) that allows concurrent computation of intersections between subscription and update regions. The second one is based on a parallel extension of the sort-based matching algorithm. Experimental evaluations of the proposed solutions confirm their effectiveness in taking advantage of multiple execution

units in a shared-memory architecture. In [46], a comparative study on three parallel sorting algorithms, i.e., Bitonic-sort, Sample-sort, and parallel Radix-sort, is presented. To study the interaction between the algorithms and the architecture, three different architectures, one based on a mesh-connected computer, the second one based on a hypercube, and finally, a distributed shared-memory machine, are proposed. The results of this work show that the choice of algorithm depends upon the number of elements to be sorted.

The second category is focused on the parallel processing of the Merge and Quick sorting algorithms for large and very large integer/non-integer data sets [33]. In the Merge-sort, the data may not be sorted until the end, and after a complete review, the data is extracted in a sorted manner. Unlike the Quick-sort that makes two unequal subarrays around the pivot capable of being sorted in a parallel fashion, mainly the Merge-sort is in serial form and can be sorted using multi-core processing systems in non-independent parallel parts [21], [32], [34]. The ideas reported in [21], [32], [35], [36] are examined for the Merge and Quick sorts in parallel processing. The results are compared to the conventional Merge and Quick sorts by proposing a division of tasks between processors. The proposed algorithm is described theoretically, examined in tests, and compared to other algorithms. The experimental and numerical results show that adding each processor causes sorting to become faster and more efficient, especially for large data sets [37], [38]. Paper [47] gives a parallel implementation of the Merge-sort on a concurrent read exclusive write-parallel random-access machine (CREW-PRAM) that uses N_P processors and reduces the running time. Also, it proposes a more complex version of the algorithm for the exclusive read exclusive write (EREW)-PRAM with a low running time. This implementation is tested for the Merge-sort and has not been considered for other comparison-based sorting algorithms. Ref. [48] proposes and simulates two parallel sorting algorithms, namely, parallel Quick-sort algorithm (PQSA) and merging subarrays from Quick-sort algorithm (MSQSA). It is focused on the Quick-sort, and its results cannot be valid for other sorting algorithms such as Merge. Neither these algorithms make the independent subarrays nor the processing time required for each processor is the same. Also, after finishing the processing of each processor, it does not guarantee that the data is entirely sorted. To address these weaknesses, we are looking for a low-complex framework to make independent equal-length subarrays sorted one-by-one in a multi-core realization that each core processes one subarray.

The third category focuses on using mean value to speed up the sorting algorithms. The first time, a mean-based (or relative) sorting algorithm was proposed in [49], [50]. Then, we extended it to an efficient algorithm in [31] by solving the drawbacks and adding some new features. Based on the simulation results and the time complexity analyses, we showed the effectiveness of the proposed mean-based algorithm to the conventional Merge, Heap, and Quick sorts in serial realizations. The self-organizing tree algorithm (SOTA) based on the mean value is used mainly for searching and clustering. It is applied as an efficient tool for classification in neural networks and image processing, tree-based routing in wireless sensor networks, and channel coding and decoding in communication systems [51]. Unlike [31], SOTA has not been applied to revise comparison-based sorting algorithms. Although the K-S mean-based sorting algorithm [31] has a similar idea to the binary tree and SOTA, it solves some problems in data sorting in serial and parallel realizations that have not been addressed in SOTA and binary tree. The results of [31] triggered us to propose a framework applicable for the comparison-based sorting algorithms. Therefore, we proposed a trial version of the proposed framework in [40] and applied it to the Insertion-sort. Numerical analysis shows its superiority in serial and parallel realizations in terms of the number of swaps, the required elapsed time, and complexity order. As noted in the literature [1], [12], [31], [34], [39], Insertion-sort is an excellent sorting algorithm for small and medium somewhat sorted data sets, not for entirely disordered large and very large data sets. Besides extending the proposed algorithm to find a general framework, Quick-sort and Merge-sort are examined, which perform well for large-scale data sets and outperform the other sorting algorithms. Hence, without losing the generality of this framework, these popular sorting algorithms are investigated as the core-sorts.

In this work, we use a limited number of mean-based divisions and then toggle to the conventional sorting algorithms, mainly to achieve a good graduality in serial realizations and a time-efficient parallel realization using stand-alone processors. To find a framework applicable for medium, large, and very large integer/non-integer data sets, in the next section, we propose a framework based on mean-based segmentation combined with one of the two popular sorting algorithms. It offers some new features and removes the bottlenecks and drawbacks of the algorithms mentioned above. To display the effectiveness of the proposed framework on the performance of the Merge and Quick sorts, in subsequent sections, we do theoretical and numerical analyses to:

- Extract the time complexity order of the mean-based framework in serial and parallel realizations and compare them to the conventional Merge and Quick sorting algorithms in the best, worst, and average cases.
- Evaluate the serial processing time and the number of swaps required for integer/non-integer data in medium, large, and very large data sets.
- Demonstrate the superiority of the proposed parallel realizations of the Merge-sort and Quick-sort to the proposed and conventional serial realizations.
- Show the effectiveness of the mean-based pivot to make approximately equal-length independent subarrays in symmetric and asymmetric data sets.
- Indicate the reduction in the complexity order of the proposed parallel realization to that of the existing

parallel method based on the random-access machine model.

III. THE PROPOSED MEAN-BASED DATA SORTING FRAMEWORK

In this investigation, instead of proposing a new sorting algorithm or improving the performance of a conventional sorting algorithm, we improve the performance of the comparison-based sorting algorithms by introducing a low-complex framework based on the mean value. It makes independent subarrays in approximately equal lengths and lets us sort in both serial and parallel realizations.

The main idea is to distinguish the sorted and similar data, if any, divide the primary data array into 2^M independent subarrays by comparing data to the mean value, and perform the well-known algorithms appropriate for unsorted or almost sorted data sets located in subarrays. The proposed framework includes three phases and several levels of division.

In the first phase of each level, it distinguishes whether the data are sorted or reversely sorted, or includes similar elements. It is named sorted/similar (SS) checker. If the data is reversely sorted, it must be inverted to create the sorted data. In the second phase, it calculates the data's mean value. It divides the data into two subarrays around the mean-based pivot, one subarray made by the elements equal or greater than the pivot and the rest of the elements in the next subarray. If the data find a favorable decision in the first phase, the second phase for that section will be terminated. Otherwise, the first and second phases continue for each subarray until we reach a predefined number of divisions. After getting the smaller data sets in the final unsorted subarrays, in the third phase, they will be sorted using core-sort that is one of the widespread sorting algorithms, i.e., Merge and Quick. Since the proposed framework divides the data into two parts, it can be examined independently at the subsequent levels. Therefore, after sorting the data in each subarray, there is no need to compare or replace the data in different subarrays, and the location of the sorted data does not change.

It can adapt the performance based on the data size, randomness rate, running time, and the number of processors by changing the number of divisions. Moreover, it enables the Merge-sort to sort data gradually, which is impossible in the conventional version of this algorithm. Furthermore, it offers a general framework for parallel realization with independent processors applicable to all types of sorting algorithms, such as Merge-sort, which are mainly in the category of serial sorts. If the core-sort has stability and adaptivity features, like Merge-sort, by using the proposed divisions, these features remain unchanged. It also introduces a higher level of adaptivity and stability by applying the proposed idea for the Quick-sort, which is mainly not an adaptive and stable sorting algorithm. It means that in different divisions, if the data includes sorted parts or similar data partially, it will not be disturbed.

In the following subsections, the serial realization of the proposed framework is described by using the pseudo-code presented in Algorithm 1 and the block diagram in Figure 1. Similarly, for parallel realization, it is explained in Algorithm 2 and Figure 2.

Algorithm 1: The Pseudo-Code of the Serial Realization of the Proposed Mean-Based Data Sorting Framework

arr: Array of data;
arr.Length: Length of the array;
M: Number of divisions;
cnt: Global counter for divisions;
si: Starting index;
ei: Ending index;
iCnt: Global counter to store the index in indexes;
status: Global array for storing the status of each subarray, 0 (default value of array) is unsorted, 1 is sorted, 2 is reversely sorted, 3 is similar;
bi: Boundary index.

```
Function Sort(arr, M):
    cnt ⟵ 0
    iCnt ⟵ 0
    Division(arr, 0, arr.Length − 1, M)
Function Division(arr, si, ei, M):
    for i ← si + 1 to ei do
        signCnt ⟵ signCnt + sign(arr[i] − arr[i − 1])
    end
    if signCnt = ei − si then
        if arr[si] < arr[si + 1] then
            status[iCnt] ⟵ 1
        else
            status[iCnt] ⟵ 2
        end
    else if signCnt ⟵ 0 then
        status[iCnt] ⟵ 3
    end
    if cnt < 2^(M−1) and status[cnt] = 0 then
        bi ⟵ Partition(arr, si, ei)
        cnt ⟵ cnt + 1
        Division(arr, si, bi, M)
        if si = 0 or ei = arr.Length − 1 then
            cnt ⟵ 1
        end
        Division(arr, bi + 1, ei, M)
    else
        Sort from si to ei using Core-Sort
    end
    return void
Function Partition(arr, si, ei):
    bi ⟵ si − 1
    mean ⟵ mean of arr[si] to arr[ei]
    for i ⟵ si to ei do
        if arr[i] >= mean then
            bi ⟵ bi + 1
            Swap arr[bi] with arr[i]
        end
    end
    return bi
```

A. SERIAL REALIZATION

Algorithm 1 presents the pseudo-code of the proposed framework for serial realization, based on three functions:

- Partition, to make two subarrays by comparing the main array or subarrays with the mean value, in each level;
- Division, to determine the number of levels and find the locations of the first and last elements in subarrays;
- Sort, that does the sorting for the final subarrays.

The global array in each level stores the status of each subarray. It is 0 for unsorted, 1 for sorted, 2 for reversely sorted, and 3 for an array containing similar elements.

As depicted in Figure 1, first, input data is checked by an SS checker for possible similarity or sorted data, if any. Otherwise, the mean value is calculated. By comparing the data with the mean value, it divides each array into two subarrays, one less than and another greater than the mean value. This procedure continues until 2^M subarrays are made. In the meantime, maybe some subarrays are sorted or have similar elements that force the algorithm to terminate these subarrays. Finally, Figure 1 shows that each subarray should be sorted by a core-sort. This process is sequential. For example, after sorting the first subarray, the second SS checker of $(M-1)$th level is activated. If there is no sorted or similar data, subarray 2 can be sorted. Then, the second SS checker of $(M-2)$th level is triggered. In this level, first, the left, and then the right parts are processed. Finally, the second SS checker of the last subarray of $(M-1)$th level is activated.

B. PARALLEL REALIZATION

Algorithm 2 for parallel realization is based on four functions as follows:

- Division, to divide array to 2^M similar length parts in each level;
- CoreSSChecker, to distinguish the data is sorted, reversely sorted, having similar elements, or unsorted;
- Partition, to calculate the mean value of these parts and divide each part into two sub-parts (one less and another greater than mean) using independent cores;
- Sort to sort final sub-parts, each one using a core, after doing the above processes for M times.

As shown in Figure 2, the parallel realization includes 2^M processors to do the checking in the SS checker, calculate the mean value, divide an array into two subarrays, and do the sorting in core-sort, all in a parallel manner. At first, each core selects $\frac{N}{2^M}$ elements from the input data. Each core lets its part be checked by the associated SS checker and calculates the sum of elements in its part. The mean value of this level can be derived by summing the mentioned calculations divided by the whole array length. After comparing the data in each part with the mean value, the corresponding core finds elements smaller and larger than the mean value. In the next step, the less parts are placed from left to right, and this is the process for larger parts at the same time. This procedure continues to up to reaching 2^M subarrays, each one including a portion less than and another greater than the mean value. Then, the whole data is sorted in a parallel manner, each by a core to check the sorted/similar, if any, and sort the rest of the data using core-sort.

Algorithm 2: The Pseudo-Code of the Parallel Realization of the Proposed Mean-Based Data Sorting Framework

```
Function Sort(arr, M):
    level ⟵ 0
    Division(arr, 0, arr.Length − 1, M, level)
Function Division(arr, si, ei, M, level):
    if level < 2^(M−1) then
        Divide arr from si to ei to 2^(M−level) similar length part
          and store si and ei of each part to partSI and partEI
        Call CoreSSChecker(arr, si, ei) for all 2^(M−level)
          parts in independent cores
        If all parts returned from CoreSSChecker were true,
          continue, otherwise return
        Calculate sum of all parts using independent cores
        mean ← mean of arr from si to ei using calculated
          sum of each part
        Call Partition(arr, si, ei) for all 2^(M−level) parts
          in independent cores (ignore the parts if CorePart
          returned false)
        lessThanMean ← copy from partSI to partBI of all
          parts
        greaterThanMean ← copy from partBI to partEI of
          all parts
        Replace from arr[si] to
          arr[si + lessThanMean.length − 1] with
          lessThanMean
        Replace from arr[si + lessThanMean.Length] to
          arr[ei] with greaterThanMean
        Call Division(arr, si,
          si + lessThanMean.Length − 1, level + 1)
          and Division(arr, si + lessThanMean.Length, ei,
          level + 1) by independent cores
    else
        if CoreSSChecker(arr, si, ei) then
            Sort from si to ei using Core-Sort
        end
    end
    return void
Function Partition(arr, si, ei):
    bi ⟵ si − 1
    mean ⟵ mean of arr[si] to arr[ei]
    for i ⟵ si to ei do
        if arr[i] >= mean then
            bi ⟵ bi + 1
            Swap arr[bi] with arr[i]
        end
    end
    return bi
Function CoreSSChecker(arr. si, ei):
    signCnt ← 0
    for i ← si + 1 to ei do
        signCnt ⟵ signCnt + sign(arr[i] − arr[i − 1])
    end
    if signCnt = ei − si or signCnt = 0 then
        if arr[si] < arr[si + 1] then
            Reverse arr from si to ei
        end
        return false
    end
    return true
```

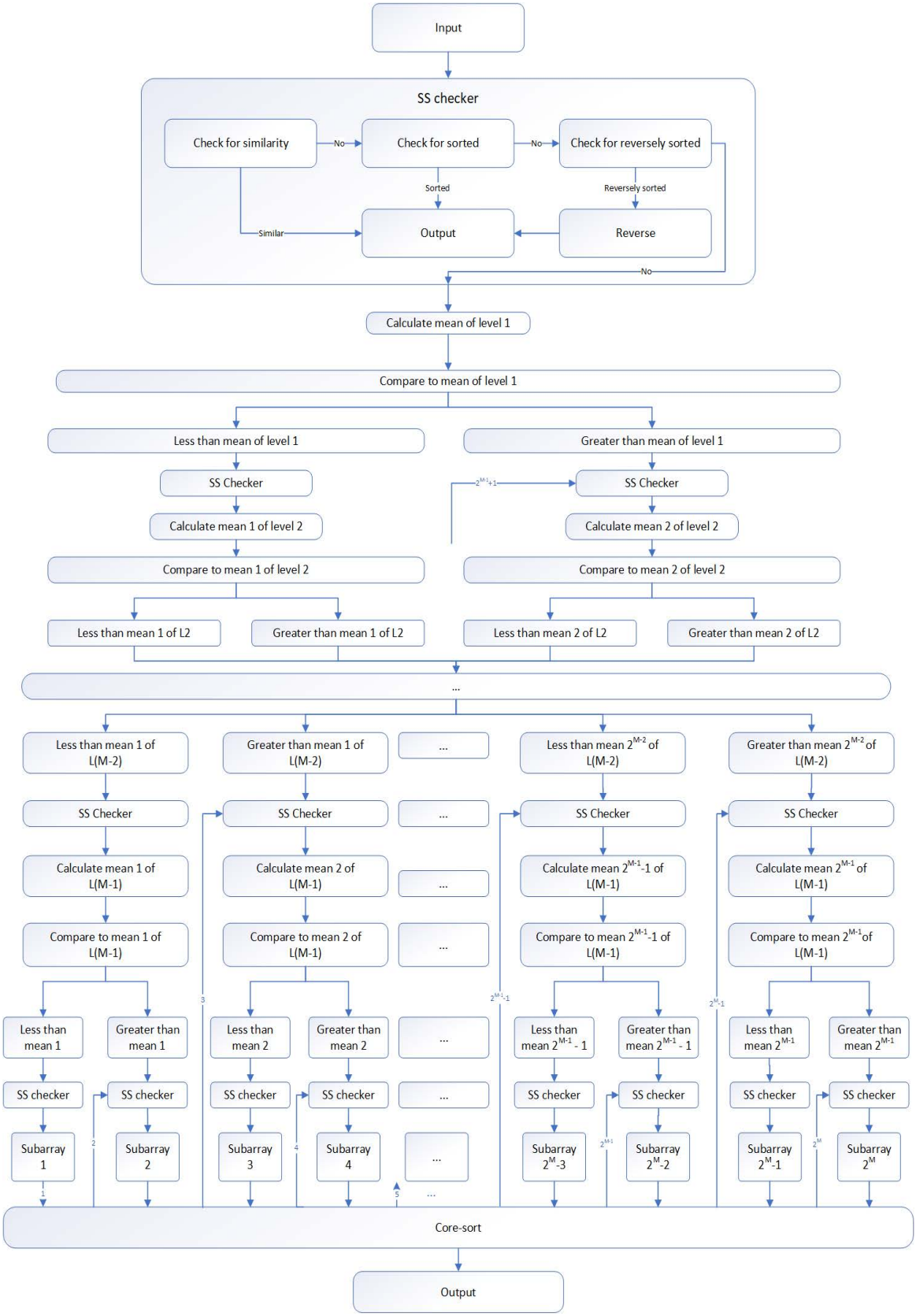

FIGURE 1. Serial realization of the proposed sorting framework.

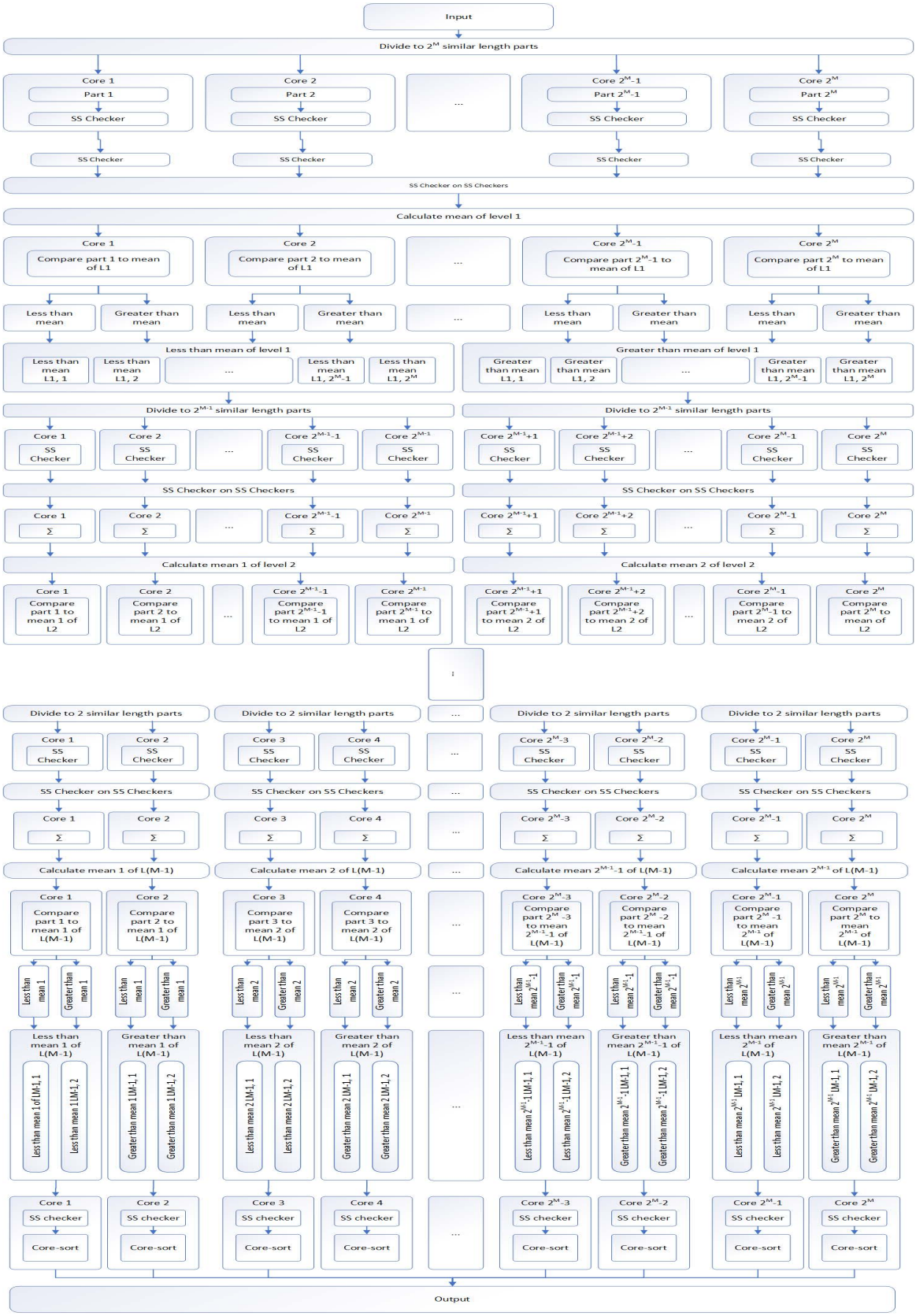

FIGURE 2. Parallel realization of the proposed sorting framework.

C. AN EXAMPLE

For example, an unordered data set including 32 integers is sorted using 3-division serial and 2^3-processor parallel frameworks, respectively in Figures 3, 4. In this example, different subarrays are created, such as sorted, reversely sorted, similar elements, and unsorted.

IV. TIME COMPLEXITY ANALYSIS

Supposing that the data size is N and log is the logarithm in base 2, in serial realization, the conventional Quick-sort has the memory complexity of order $O(logN)$ (or $O(N)$, $O(NlogN)$ depending on the type of realization and memory usage). It is a linear complexity order, $O(N)$, for the Merge-sort. In the worst, average, and best cases of the Merge-sort, the time complexity is $O(N \log N)$, while the time complexity for the worst-case of the Quick-sort is $O(N^2)$, and that for the average and best cases is $O(N \log N)$ [13], [15], [32].

In this section, the time complexity of the proposed mean-based framework based on the Quick and Merge sorting algorithms in serial and parallel realizations is analyzed. This complexity consists of two parts. The first part shows the time complexity of phase one, which includes the time required to compute counter and reverse the data if they are reversely sorted at each level, the time needed to calculate the mean value at each level of the divisions, and the swaps required to make independent subarrays. The second part is the time complexity of the core-sort used to sort the data in the final subarrays. To detect the sorted data (in ascending or descending order), at each level, we compute the counter. The value of this counter is the total difference sign of the adjacent elements. If it is $(N-1)$, i.e., the data are sorted. If it is zero, the data are similar. Otherwise, the data is neither sorted nor contains similar elements.

In serial realization, each of the mentioned operations in the first part, finding the sign of differences and calculating the counter, reversing the data if it is reversely sorted, finding the mean value, and making two subarrays, needs a linear time complexity of $O(N)$ because the total size of data should be searched. Also, the proposed framework needs a linear memory space complexity of $O(N)$. Accordingly, in the first division, if data are sorted or have similar data, $O(2N)$ time complexity is required. If data are reversely sorted, it needs $O(3N)$ time complexity. When data are none of the above cases, we need to calculate the mean value of each array and divide it into two subarrays around the mean value. Hence, the upper bound of the time complexity in the first level is

$$T_{S1} = O(4N) \tag{1}$$

After the first division, assuming there is no sorted/similar data, we have two $\frac{N}{2}$-member subarrays. In the second division, an upper bound of time complexity required for the two existing subarrays is

$$T_{S2} = 4 \times 2O(\frac{N}{2}) = O(4N) \tag{2}$$

Finally, in the Mth division, it is

$$T_{SM} = 4 \times 2^{M-1} O(\frac{N}{2^{M-1}}) = O(4N) \tag{3}$$

After M level of divisions, we have 2^M sub-arrays with an approximate number of $\frac{N}{2^M}$ elements. To reach this level, in serial realization proposed in this investigation, the upper bound of the required time complexity of phase one is

$$T_{St} = \sum_{i=1}^{M} T_{Si} = O(4M.N) \tag{4}$$

As mentioned above, each of adding (or subtraction), mean value calculation, comparison, and division processes in serial realization lasts in $T_S = O(N)$. In N_P processors hypercube, it is as (5) for parallel realization [52]–[55]

$$T_P = O(\frac{N}{N_P} + \log N_P) \tag{5}$$

Using speedup factor defined as a metric for comparing the time complexity order of parallel and serial realizations as

$$SU_{P-S} = \frac{T_S}{T_P} \tag{6}$$

we have

$$SU_{P-S} = \frac{N.N_P}{N + N_P \log N_P} \tag{7}$$

If the number of processors equals the data size, it is as (6) [30].

$$SU_{P-S} = \frac{N}{1 + \log N} \tag{8}$$

For a limited number of processors, when the data size is too large (i.e., $N \gg N_P$), it is given as

$$SU_{P-S} \approx N_P \tag{9}$$

In the parallel realization of the proposed framework, in the first level

$$T_{P1} = 4O(\frac{N}{N_P} + \log N_P) \tag{10}$$

For the second level, we have

$$T_{P2} = 4 \times 2O(\frac{\frac{N}{2}}{N_P} + \log N_P) \tag{11}$$

Finally, at the Mth level, it is given as

$$T_{PM} = 4 \times 2^{M-1} O(\frac{\frac{N}{2^{M-1}}}{N_P} + \log N_P) \tag{12}$$

Hence, the total time complexity of parallel realization to reach independent subarrays approximately in equal length is

$$T_{Pt} = 4MO(\frac{N}{N_P}) + 4\sum_{i=1}^{M} 2^{i-1} O(\log N_P) \tag{13}$$

or

$$T_{Pt} = O(\frac{4M.N}{N_P}) + O(4 \times (2^M - 1) \log N_P) \tag{14}$$

FIGURE 3. The 3-division serial realization of the proposed framework for sorting a unordered data set including 32 integers.

FIGURE 4. The 2^3-processor parallel realization of the proposed framework for sorting a unordered data set including 32 integers.

If $N_P = 2^M$, it is as

$$T_{Pt} = O(4M(\frac{N}{2^M} + 2^M - 1)) \quad (15)$$

In the following subsections, when the Merge-sort and Quick-sort are used as the core-sort, we extract the time complexity order of average, best, and worst cases in serial and parallel realizations of the proposed framework.

A. AVERAGE-CASE

The complexity order of the serial realization of the conventional Quick and Merge sorting algorithms for the average-case is as [34], [38]

$$T_{SQ} = T_{SM} = O(N \log N) \quad (16)$$

Assuming there are 2^M subarrays with the length of $\frac{N}{2^M}$, the time complexity order to sort each subarray by the conventional Quick and Merge sorts is as

$$T_{SQ} = T_{SM} = O(\frac{N}{2^M} \log(\frac{N}{2^M})) \quad (17)$$

Considering equation (17) to 2^M times, and the evaluations mentioned above (equation 4), the total time complexity order of the proposed framework in serial realization for average-case is as

$$T_{PS-AC} = O(4M.N) + O(N \log(\frac{N}{2^M})) \quad (18)$$

In the parallel realization, equation (17) should be considered one time. By considering this time complexity and the evaluations mentioned above (equation 15), the total time complexity order of the proposed framework in parallel realization for average-case is as

$$T_{PP-AC} = O(4M(\frac{N}{2^M} + 2^M - 1)) + O(\frac{N}{2^M} \log(\frac{N}{2^M})) \quad (19)$$

When the following inequality is valid

$$3M.N + 4M \times 2^M(2^M - 1) < N \log N \quad (20)$$

the time complexity of the proposed framework in parallel realization will be almost as part 2, required to parallel sorting of 2^M subarrays.

B. BEST-CASE

Best-case occurs when it detects whether the data is ordered, has similar values, or is reversely sorted. By using any sorting algorithm, complexity order required in serial realization of the proposed framework, in the orderly data or data with similar elements is as

$$T_{PS-BC} = \begin{cases} O(2N), & ordered/similar \\ O(3N), & reversely\ ordered \end{cases} \quad (21)$$

In parallel realization, for the best-case, there is complexity order $O(2 \times \frac{N}{N_P} + 2 \times \log N_P)$ or $O(3 \times \frac{N}{N_P} + 3 \times \log N_P)$. So, with 2^M processors, we have

$$T_{PP-BC} = \begin{cases} O(\frac{2N}{2^M} + 2M), & ordered/similar \\ O(\frac{3N}{2^M} + 3M), & reversely\ ordered \end{cases} \quad (22)$$

C. WORST-CASE

Worst-case for the Quick-based proposed framework occurs when all divisions are done up to 2^M subarrays and in none of these steps, ordered, reversely ordered, and similar data do not happen. Then, in one subarray, several subarrays, or all of 2^M subarrays, there is ordered, similar data, or reversely ordered data. Therefore, worst-case in the serial realization of the Quick-based framework has the following time complexity

$$T_{PS-WC} = O(4M.N) + O((2^M - k)\frac{N}{2^M} \log(\frac{N}{2^M}) + k\frac{N^2}{2^{2M}}) \quad (23)$$

where k is the number of subarrays sorted, reversely sorted, or have similar elements. The first part is the complexity order to create 2^M subarrays.

For the parallel realization of the Quick-based framework, it is

$$T_{PP-WC} = O(4M(\frac{N}{2^M} + 2^M - 1)) + O(\frac{N^2}{2^{2M}}) \quad (24)$$

The first part is the complexity of reaching 2^M subarrays by parallel realization.

In the Merge-based framework, when data in final subarrays are sorted or similar, the best-case for sorting each subarray happens. In contrast, when final subarrays are reversely ordered, worst-case for the Merge-sort happens. Because there is no difference between the time complexity order of the worst-case and average-case of the Merge-sort, the time complexity order of the worst-case in the serial realization of the Merge-based framework is

$$T_{PS-WC} = O(4M.N) + O(N \log(\frac{N}{2^M})) \quad (25)$$

and it for parallel realization is

$$T_{PP-WC} = O(4M(\frac{N}{2^M} + 2^M - 1)) + O(\frac{N}{2^M} \log(\frac{N}{2^M})) \quad (26)$$

D. COMPARISONS BASED ON TIME COMPLEXITY

Table 1 depicts the time complexities of the conventional Quick and Merge sorts in the best-case, average-case, and worst-case, and the evaluations mentioned above for the three cases of the Quick-based and Merge-based proposed framework in serial and parallel realizations. In each of the average-case and worst-case complexity orders, when the complexity is the sum of two terms, the more significant term determines the complexity that depends on the values of M and N. In this table, the first part is the upper bound associated with the time complexity of phase one. The second part is related to the core-sort's time complexity in phase two. Hence, showing these two terms illustrates the effect of the complexity of the first and second phases of the proposed framework on the final complexity order.

In division M of the proposed framework, if the counter is $(\frac{N}{2^{M-1}} - 1)$, or all elements are in one subarray, the following divisions and sorting for that subarray in level M will be finished. It means that the time complexity derived for the

TABLE 1. The time complexity of the conventional and proposed quick and merge sorts in the best, average, and worst cases.

Algorithm		Time complexity order		
		Best-case	Average-case	Worst-case
Quick	Conventional	$O(N \log N)$	$O(N \log N)$	$O(N^2)$
	Proposed serial	$\begin{cases} O(2N), \ ordered/similar \\ O(3N), \ reversely\ ordered \end{cases}$	$O(4M.N) + O(N \log(\frac{N}{2^M}))$	$O(4M.N)+$ $O((2^M - k)\frac{N}{2^M}\log(\frac{N}{2^M}) + k\frac{N^2}{2^{2M}})$
	Proposed parallel	$\begin{cases} O(\frac{2N}{2^M} + 2M), \ ordered/similar \\ O(\frac{3N}{2^M} + 3M), \ reversely\ ordered \end{cases}$	$O(4M(\frac{N}{2^M} + 2^M - 1)) +$ $O(\frac{N}{2^M}\log(\frac{N}{2^M}))$	$O(4M(\frac{N}{2^M} + 2^M - 1)) +$ $O(\frac{N^2}{2^{2M}})$
Merge	Conventional	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
	Proposed serial	$\begin{cases} O(2N), \ ordered/similar \\ O(3N), \ reversely\ ordered \end{cases}$	$O(4M.N) + O(N \log(\frac{N}{2^M}))$	$O(4M.N) + O(N \log(\frac{N}{2^M}))$
	Proposed parallel	$\begin{cases} O(\frac{2N}{2^M} + 2M), \ ordered/similar \\ O(\frac{3N}{2^M} + 3M), \ reversely\ ordered \end{cases}$	$O(4M(\frac{N}{2^M} + 2^M - 1)) +$ $O(\frac{N}{2^M}\log(\frac{N}{2^M}))$	$O(4M(\frac{N}{2^M} + 2^M - 1)) +$ $O(\frac{N}{2^M}\ log(\frac{N}{2^M}))$

average-case is the upper bound, and most of the time, the data sorting needs a lower complexity.

According to (16), (18), the time complexity is changed from $O(N \log N)$ to $O(4M.N) + O(N \log(\frac{N}{2^M}))$, in serial realization. When the number of divisions satisfies the inequality (27),

$$N > 2^{3M} \tag{27}$$

part 2 of the time complexity determines the time complexity. In this case, the proposed serial realization will be slightly faster than the conventional serial realization of the Merge and Quick sorting algorithms. Hence, the amount of speedup factor, defined as $\frac{T_{CS}}{T_{PS}}$ is

$$SU_{PS-CS} = \frac{N \log N}{NlogN - M} \tag{28}$$

Of course, in evaluating the novel implementation of the proposed serial realization, there is an improvement in the results compared to the conventional algorithms because in dividing the data into smaller subarrays, the variance of the subarrays decreases. The distance between the locations that need swap has also decreased. In contrast, for a large number of divisions, which do not apply to inequality (27), the serial realization of the proposed framework will be slower than the conventional serial realization.

By comparing serial and parallel realizations of the proposed framework, the speedup factor, $\frac{T_{PS}}{T_{PP}}$, is as

$$SU_{PP-PS} = \frac{(3M.N + N \log N).2^M}{3M.N + N \log N + 4M \times 2^M(2^M - 1)} \tag{29}$$

As long as inequality (24) holds, the proposed parallel framework takes precedence over the proposed serial framework

$$3M.N + N \log N > 4M.2^M \tag{30}$$

It can be seen that this inequality is valid for the cases that $N > 2^M$.

In the Big O notation that shows the upper bound of the complexity, $O(f(n))$ is the set of all functions with an eventual growth rate less than or equal to that of f. So,

$$O(N) = O(2N) = O(3N) = \cdots = O(9N) \tag{31}$$

They experience the same growth rates, namely, the linear growth rate [56], [57]. On the contrary, there is a remarkable difference between $O(N)$ and $O(N^2)$ or $O(N)$ and $O(N \log N)$. Considering constants 2, 3, and 4, in the time complexity analysis and Table 1, the difference between the time complexities is better shown, especially for a small number of divisions, M, and small data sizes, N. For example, when comparing $O(NlogN)$ and $O(4MN)$, the difference between $logN$ and $4M$ differs from $logN$ and M, in a small number of divisions.

As an example, Table 2 shows the time complexity order of the conventional and proposed Quick and Merge algorithms in the best, average, and worst cases, when $N = 2^{24}$ and $M = 1, 2, 3, 4$. In the worst-case of the proposed serial realization of the Quick-sort, it is assumed that $k = 2^M$. Also, the proposed serial and parallel realizations experience lower time complexity than that of conventional sorting algorithms. In the average case, we add a complexity order for evaluating the mean value and partitioning the primary array to subarrays. The total time for sorting is much lower than the conventional algorithms because the time complexity order required to make subarrays is much lower than that for sorting the data in subarrays. Also, by comparing the time complexity of the proposed framework in parallel and serial realizations, it can be seen that the parallel realization is always performed better than the serial one, and this superiority is more evident in large and very large data sets.

V. NUMERICAL ANALYSES AND SIMULATION RESULTS

This section and consequent ones try to show the effectiveness of the proposed mean-based framework for data sorting. Therefore, the performance of the serial and parallel realizations of the proposed framework equipped with two

TABLE 2. The time complexity of the conventional and proposed quick and merge algorithms in the best, average, and worst cases ($N = 2^{24}$ and $M = 1, 2, 3, 4$).

Algorithm			Time complexity order		
			Best-case	Average-case	Worst-case
Quick	Conventional		$O(3 \times 2^{27})$	$O(3 \times 2^{27})$	$O(2^{48})$
	Proposed serial	$M = 1$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{26}) + O(23 \times 2^{24})$	$O(2^{26}) + O(2^{47})$
		$M = 2$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{27}) + O(11 \times 2^{25})$	$O(2^{27}) + O(2^{46})$
		$M = 3$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(3 \times 2^{26}) + O(21 \times 2^{24})$	$O(3 \times 2^{26}) + O(2^{45})$
		$M = 4$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{28}) + O(5 \times 2^{26})$	$O(2^{28}) + O(2^{44})$
	Proposed parallel	$M = 1$	$O(2^{24})$ or $O(3 \times 2^{23})$	$O(2^{25}) + O(23 \times 2^{23})$	$O(2^{25}) + O(2^{46})$
		$M = 2$	$O(2^{23})$ or $O(3 \times 2^{22})$	$O(2^{24}) + O(11 \times 2^{23})$	$O(2^{24}) + O(2^{44})$
		$M = 3$	$O(2^{22})$ or $O(3 \times 2^{21})$	$O(2^{23}) + O(21 \times 2^{21})$	$O(2^{23}) + O(2^{42})$
		$M = 4$	$O(2^{21})$ or $O(3 \times 2^{20})$	$O(2^{22}) + O(5 \times 2^{22})$	$O(2^{22}) + O(2^{40})$
Merge	Conventional		$O(3 \times 2^{27})$	$O(3 \times 2^{27})$	$O(3 \times 2^{27})$
	Proposed serial	$M = 1$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{26}) + O(23 \times 2^{24})$	$O(2^{26}) + O(23 \times 2^{24})$
		$M = 2$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{27}) + O(11 \times 2^{25})$	$O(2^{27}) + O(11 \times 2^{25})$
		$M = 3$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(3 \times 2^{26}) + O(21 \times 2^{24})$	$O(3 \times 2^{26}) + O(21 \times 2^{24})$
		$M = 4$	$O(2^{25})$ or $O(3 \times 2^{24})$	$O(2^{28}) + O(5 \times 2^{26})$	$O(2^{28}) + O(5 \times 2^{26})$
	Proposed parallel	$M = 1$	$O(2^{24})$ or $O(3 \times 2^{23})$	$O(2^{25}) + O(23 \times 2^{23})$	$O(2^{25}) + O(23 \times 2^{23})$
		$M = 2$	$O(2^{23})$ or $O(3 \times 2^{22})$	$O(2^{24}) + O(11 \times 2^{23})$	$O(2^{24}) + O(11 \times 2^{23})$
		$M = 3$	$O(2^{22})$ or $O(3 \times 2^{21})$	$O(2^{23}) + O(21 \times 2^{21})$	$O(2^{23}) + O(21 \times 2^{21})$
		$M = 4$	$O(2^{21})$ or $O(3 \times 2^{20})$	$O(2^{22}) + O(5 \times 2^{22})$	$O(2^{22}) + O(5 \times 2^{22})$

well-known sorting algorithms, i.e., Merge and Quick, are compared to the conventional ones based on the required elapsed time and the number of swaps.

The type of data, such as sorted, somewhat sorted, and random, the size of data, i.e., small, medium, large, and very large, the data variance for low and high values, the number of mean-based divisions, and the number of processors, are the main variables.

Numerical results are based on the evaluations and comparisons for sorting integer/non-integer random data with 2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}, 2^{24}, 2^{27}, 2^{30}, sizes, and low (10) and high (1000) standard deviations. The simulations are run in six data sets, i.e., integer/non-integer uniform, integer/non-integer Gaussian, and integer/non-integer Rayleigh. In subsections A and B, and Section VI, we just focus on sorting non-integer uniform data in descending order because the trend for ascending order and the other data sets is the same as that for non-integer uniform data. Consequently, these six data sets are investigated deeply in Section VII to see the effectiveness of the proposed mean-based framework. All numerical results are averaged over 1000 realizations for four cases, 1-division, 2-division, 3-division, and 4-division, which means 2, 4, 8, and 16, independent subarrays, respectively. Associated $C^{\#}$ codes for descending sorts and some MATLAB evaluations are run on a 64bit system with Ubuntu 20.04.1 LTS, Intel core i7-9750H with 16GB RAM, 12MB Cache, 2.6GHz Max. CPU speed.

Assuming that there is no similar or sorted data in phase 1 of each level, we compare the performance of the proposed Merge-based and Quick-based framework to that of the conventional ones. Simulation results show an upper bound for the number of swaps and elapsed time of the average-case as reported in Table 3, Figure 5, and Table 4, Figure 6 for serial realization, and Figures 7, 8 for parallel realization. The results of the proposed framework for the cases that data is partially or entirely sorted or have similar elements will be improved, because we check it before partitioning an array and performing the core-sort.

A. PERFORMANCE OF SERIAL REALIZATION

Table 3 and Table 4 demonstrate the performance metrics for the conventional Merge-sort and Quick-sort, respectively. As expected, the proposed framework decreases the elapsed time and the number of swaps in both sorting algorithms. It sorts the whole data in 2, 4, 8, or 16 independent subarrays, respectively, in 1, 2, 3, and 4 divisions. On the contrary,

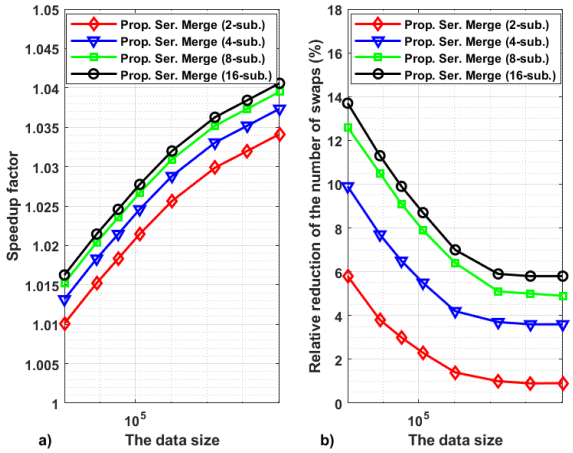

FIGURE 5. The relative performance of the proposed Merge-based serial framework to the conventional algorithm, a) Speedup factor, b) Relative reduction of the number of swaps.

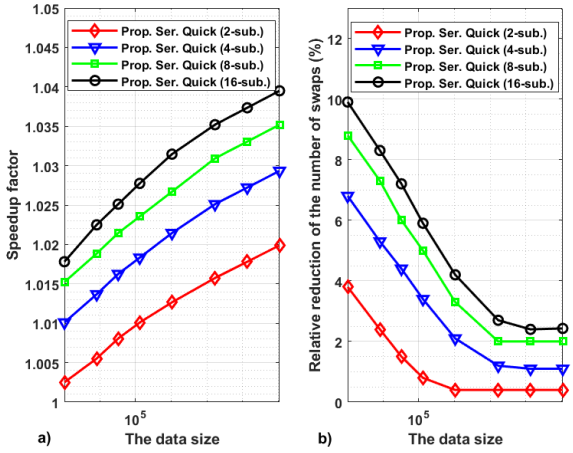

FIGURE 6. The performance of the proposed quick-based serial framework to the conventional algorithm, a) Speedup factor, b) Relative reduction of the number of swaps.

calculating the mean value, comparing each level's data with the mean value, and locating them in 2, 4, 8, and 16 independent subarrays are why the elapsed time and number of swaps increased. Hence, consistent with the slight improvement seen in the time complexity order in the previous section, Figures 5, 6 depict that total time and the number of swaps for both Merge-based and Quick-based serial realizations of the proposed framework are slightly better than the associated values for the conventional ones.

Figure 5.a shows a speedup factor between (1.01–1.016) and (1.034–1.04) for different divisions of the serial realization of the Merge-based framework, respectively, for small to very large data sets. As shown in Figure 6.a, it is about (1.003–1.018) and (1.02–1.04) for Quick-based serial realization of the proposed framework. Figures 5.b and 6.b depict relative reduction of the number of swaps to the conventional ones, from (6–14%) for small data sets to (1–6%) for very large data sets and from (4–10%) for small data sets to (0.5–2.5%) for very large data sets, respectively, for Merge-based and Quick-based serial realizations of the proposed framework. It means that increasing the number of divisions is the reason for decreasing the number of swaps.

The improvement rates of the elapsed time and the number of swaps are decreased, respectively, from $M = 1$ relative to the conventional algorithm, to $M = 2$ relative to $M = 1$, to $M = 3$ relative to $M = 2$, to $M = 4$ relative to $M = 3$, because in mean-based divisions, at each level of division, the subarrays reach a more minor variance than the main array. This reduction in variance is more significant at the first divisions,and then it will be reduced.

B. PERFORMANCE OF PARALLEL REALIZATION

A typical personal computer CPU has 2, 4, 8, 16, 32, 64, or 128 cores. Intel/AMD productions have the following features:

- Mainstream consumer-grade processors: 2 to 16 cores;
- High-end workstation: Up to 64 cores (AMD) or 18 (Intel);
- Single socket server processor: Up to 64 cores (AMD) or 56 cores (Intel);
- Dual socket (AMD): Up to 128 cores;
- Dual socket (Intel): Up to 112 cores;
- Scalable Intel platforms: Up to 8 sockets (Intel) for 448 cores.

Besides improving the running time by increasing the memory size and decreasing the number of swaps, multi-core parallel processing is another solution [35], [36], [58] to achieve a fast sorting algorithm. Most of the sorting algorithms can be just realized in a serial form or parallel fashion with connected cores. Still, a small number of sorting algorithms can be realized, either serial or parallel, with stand-alone cores, such as Quick-sort [59]. Parallel processing of sorting algorithms is done by different researchers using multi-core structures. Most of them do the serial realizations of sorting algorithms in a connected-core parallel manner by a flexible division of tasks between the processors, which means the results of the processors are dependent on each other. After combining the outputs of these processors, the sorting process can be finalized. On the contrary, the proposed framework in this work makes approximately equal length independent subarrays that really can be sorted by non-connected processors in a realistic parallel realization.

In parallel realization, running (run) time is defined as the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes. In the proposed parallel scenario, non-connected (independent) processors are applied. Therefore, the processor that processes the required processes of the largest subarray determines the processing time. By making two independent subarrays approximately equal in length, a lower running time than a similar algorithm with non-equal subarrays can be achieved. Moreover, the randomness rate in each subarray is decreased because the subarray's variance is reduced.

Although the proposed framework makes a slight improvement in the required elapsed time and number of swaps

TABLE 3. The required elapsed time and the number of swaps of the conventional merge-sort.

Data size	2^{10}	2^{13}	2^{15}	2^{17}	2^{20}	2^{24}	2^{27}	2^{30}
The required elapsed time (ms)	0.17	1.8	8.2	36	567	5.1×10^4	2.2×10^6	2.1×10^8
The number of swaps	10240	1.1×10^5	4.9×10^5	2.2×10^6	3.7×10^7	1.5×10^{10}	8.4×10^{12}	9.8×10^{14}

TABLE 4. The required elapsed time and the number of swaps of the conventional quick-sort.

Data size	2^{10}	2^{13}	2^{15}	2^{17}	2^{20}	2^{24}	2^{27}	2^{30}
The required elapsed time (ms)	0.11	1.2	5.4	24.6	421.4	5.8×10^4	3.3×10^6	3.7×10^8
The number of swaps	6377	6.9×10^4	3.2×10^5	1.5×10^6	2.6×10^7	4.3×10^9	3.4×10^{11}	3.3×10^{13}

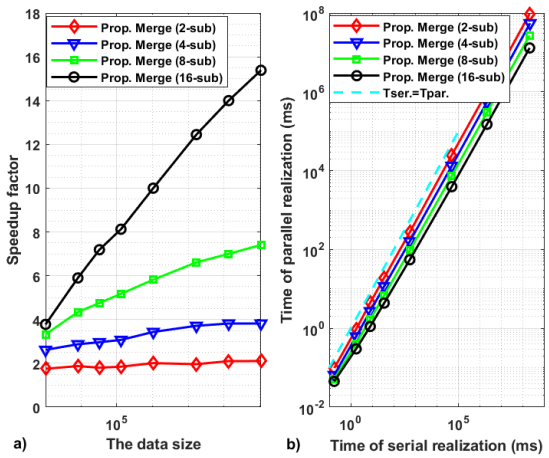

FIGURE 7. The performance of the proposed Merge-based framework, a) Speedup factor as the time of serial realization to time of parallel realization ratio, b) Time of parallel realization vs. the time of serial realization.

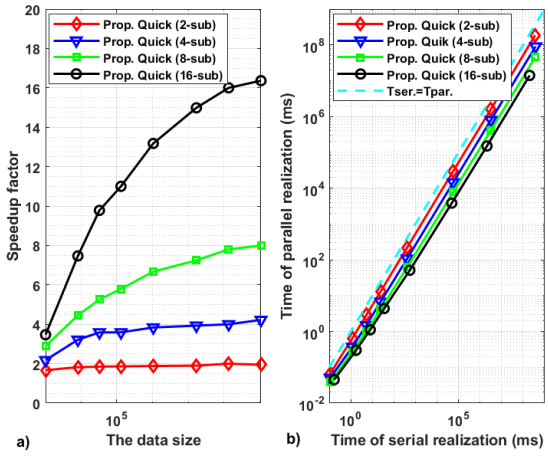

FIGURE 8. The performance of the proposed Quick-based framework, a) Speedup factor as the time of serial realization to time of parallel realization ratio, b) Time of parallel realization vs. the time of serial realization.

in the serial realization, it allows the Merge-sort to be realized in a parallel fashion and Quick-sort in a time-efficient parallel scenario. Figures 7.a, 8.a demonstrate the speedup factor defined as the elapsed time of the proposed parallel realization to that of the associated serial realization for the Merge-based and Quick-based scenarios, respectively. Also, Figures 7.b, 8.b show the time required for parallel realization versus that for serial realization, respectively, for the Merge-based and Quick-based frameworks. The line $T_{ser.} = T_{par.}$ means that serial and parallel realizations require the same elapsed time to sort data. In Figures 7.b and 8.b, it is shown that the elapsed time of parallel realization of the proposed framework is less than the time of serial realization, i.e., $T_{ser.} > T_{par.}$. Increasing the number of divisions (equivalently the number of processors) increases the slope of these curves relative to the line $T_{ser.} = T_{par.}$.

By making 2, 4, 8, and 16 independent subarrays respectively in 1, 2, 3, and 4 divisions, in parallel realization, we asymptotically expect $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{16}$ of the elapsed time and swaps required for serial processing that the simulation results prove it for very large data sets. As shown in Figure 5.a, the speedup factor for the Merge-based parallel realization of the proposed framework is in the intervals $(2-4\%)$ and $(2-15.5\%)$ for small and very large data sets, respectively. As depicted in Figure 6.a, it is about $(2-4\%)$ and $(2-16\%)$ for the Quick-based scenario.

C. HIGHLIGHTS OF THEORETICAL AND NUMERICAL ANALYSES

By summarizing the results of the time complexity analysis and simulations for both serial and parallel realizations of the proposed framework, the following can be extracted:

- The counter used, decreases the processing time and the number of swaps required to sort the data if there exists sorted data in an ascending or descending order, or data with similar elements. This capability of the proposed framework improves the best-case's and worst-case's time complexity of the conventional algorithms analyzed. If data elements are unsorted, randomly distributed, this counter can be deactivated.

- Serial realization of the proposed framework can sort and extract data gradually, which is not possible in the conventional Merge-sort. From this point, it also improves Quick-sort's performance because the subarrays are almost in the same length, and additional divisions are avoided due to unbalanced subarrays in the conventional Quick-sort. It means that the required time to extract the sorted parts of data in the consecutive subarrays can be estimated before ending the process.
- Using the proposed framework, as the independent subarrays become smaller, the distance between the swapped locations is reduced, which takes less time. In contrast, the required time to calculate the mean value and swaps to make subarrays in the partitioning phase are added. On average, they are almost equal. This issue has been addressed in the complexity analysis.
- Similar results are obtained for the integer uniform and integer/non-integer Gaussian and Rayleigh data, not reported here to avoid repeating similar trends.
- In random data, the number of swaps required to sort data by size N is greater than the total number of swaps for two subarrays with length $\frac{N}{2}$. Comparisons on the number of swaps of different sorting algorithms can be made by the maximum absolute error (MAE) criterion proposed in [31]. In this paper, it was shown that for data of size N, the difference between the locations in the unsorted and sorted data is equal to $\frac{N^2}{2}$ and in two subarrays of length $\frac{N}{2}$, it is totally $2 \times \frac{(\frac{N}{2})^2}{2} = \frac{N^2}{2^2}$. The ratio of 2 indicates the rate of reduction of swaps. In other words, if random data with a size of N can be replaced by two random data sets of $\frac{N}{2}$ size, sorting these two subarrays requires half the number of swaps.
- In the proposed framework, two independent subarrays are created at each level, and some swaps are added to make them. On the other hand, with a 50% reduction in the length of each of the two subarrays compared to the main array, fewer swaps are required. It is due to this fact that in the subarrays of each array, by using the proposed partitioning around the mean value, the variance decreases, and the data is somewhat sorted. We expect a reduction in the number of swaps compared to the conventional algorithms.
- Although increasing the number of divisions causes a more significant decrease in the number of swaps and elapsed time of the algorithm, the rate of improvement has a decreasing trend.
- Since the increase in data size is the reason for the decrease in the number of swaps, the elapsed time shows a further decrease. Firstly, because the distance between the elements to be swapped is reduced, and secondly, in the Quick-sort, the number of subsequent divisions is also reduced.
- In the serial realization, the Merge-based proposed framework experiences a bigger improvement than the Quick-based one because there are no independent subarrays in the conventional Merge-sort. Despite being possible in the conventional Quick-sort, they are approximately equal length in the proposed framework.
- In the serial realization, the percentage of reduction of elapsed time and number of swaps in the Merge-based algorithm is higher than the Quick-based algorithm because the Merge-sort does its internal divisions until it reaches the subarrays with two elements, and by making subarrays, more help to this algorithm becomes.
- The proposed parallelism shows its superiority over the serial realization in larger data sets. Also, the speedup factor in a similar number of divisions for larger data is higher than that for smaller data because in larger data it is more possible to have subarrays of the same length.
- Comparing the parallel and serial realizations show a reduction in the run-time. Moreover, the proposed Quick-based parallel framework has higher performance than the case in which the arrays of the conventional Quick-sort are processed in parallel because the subarrays assigned to the parallel processors are much more similar in length, so the number of subsequent divisions and swaps will be less, which means lower run-time.

VI. COMPARISON TO THE SORTING ALGORITHM BASED ON PARALLEL RANDOM-ACCESS MACHINE MODEL

It is convenient to use the parallel random-access machine (PRAM) model to analyze the parallel sorting algorithms. Depending on the method of access of the processors to the memory, there exist three types of PRAM machines, such as exclusive read exclusive write (EREW), concurrent read exclusive write (CREW), and concurrent read concurrent write (CRCW). EREW-PRAM allows reading/writing memory using only one processor. CREW-PRAM provides reading memory by any number of processors while simultaneously writing can be run only by one processor. This type reflects the architecture of the modern computer and practically makes it possible to write efficient parallel implementations. The third type, CRCW-PRAM allows accessing memory using any number of processors [60]–[62]. The possibility of parallelizing sorting processes is an important issue for sorting data sets. The PRAM machine model can model the division of tasks with low time complexity. Marszalek, Wozniak, and others [35], [36], [57], [61], [63], using N_P processors for sorting a data set including N elements on CREW RAM machine, achieved a time complexity order [35], [36] as (32)

$$T_{MW} = O(\frac{N}{N_P}(logN)^2) \tag{32}$$

It was shown that increasing the number of processors to be equal to the data size, i.e., $N_P = N$, changes the time complexity order of the Merge and Quick sorting algorithms in serial realization, $O(NlogN)$, to a complexity order of $O((logN)^2)$ in the proposed parallel realization in [35], [36], [62]–[64]. Although this reduction in time complexity is a good achievement, in most applications, the number of

processors is limited, especially in large and very large data sets. In other words, it is necessary to use a parallel realization of sorting algorithm that uses a limited number of processors. Hence, in this investigation, we are motivated to propose and test a new framework that uses a limited number of processors in parallel realization.

To compare the time complexity order of the sorting algorithm proposed in this paper with the algorithm presented by Marszalek and Wozniak [35], [36], we assume that the number of mean-based independent subarrays with approximately equal length, 2^M is equal to the number of processors, N_P. By using the speedup factor (SU) as a metric to compare the time complexity of the parallel sorting algorithm offered in this work (equation 19) and that presented by Marszalek and Wozniak (equation 33), as the following ratio

$$SU_{PP-MW} = \frac{T_{MW}}{T_{PP}} \tag{33}$$

we have

$$SU_{PP-MW} = \frac{(logN)^2}{logN + 3M + \frac{4M}{N} \times 2^M(2^M - 1)} \tag{34}$$

Generally, to have a lower time complexity to the previous algorithm, or equivalently to have a speedup factor more significant than one, the maximum number of divisions should satisfy the following inequality

$$3M.N + 4M.2^M(2^M - 1) \leq N \log N(logN - 1) \tag{35}$$

As demonstrated in Table 5, when the size of the data increases and the number of processors is limited, the superiority of the proposed sorting algorithm over the previous algorithm becomes more apparent. By increasing the number of divisions and processors, the speedup factor decreases. Blank elements of this table show that the algorithm proposed by Marszalek and Wozniak, experiences lower complexity and running time than the proposed one. For the cases that the previous algorithm is less complex than the proposed algorithm (e.g., the number of processors is 64 or higher for data with a size of 2^{10}), the elapsed time of the proposed algorithm may be shorter because, after consecutive mean-based divisions, we have smaller subarrays with a minor variance that can be sorted faster. Another reason is that we have fewer swaps with smaller distances. Therefore, when the number of processors is not too large, the proposed algorithm will be preferable because of time complexity order, the required running time, and the number of swaps. For data sets with 2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}, 2^{24}, 2^{27}, and 2^{30} elements, the maximum number of processors that the proposed algorithm of this work is less complex and faster than the previous algorithm are 32, 128, 256, 512, 2048, 8192, 32768, and 65536 (or equivalently, the number of divisions equal to 5, 7, 8, 9, 11, 13, 15, and 16), respectively.

VII. EFFECTIVENESS OF THE MEAN-BASED DIVISIONS

In Sections 4, 5, and 6, based on the time complexity and simulations, the performance improvement of serial and parallel realizations of the proposed framework compared to the conventional serial and parallel realizations was shown. The main reason for these improvements is the mean-based divisions and making independent subarrays of approximately equal length, which are achieved in both serial and parallel realizations. The difference between the lengths of the subarrays is the primary criterion for measuring the superiority of the mean-based divisions compared to the random pivot that can be seen using the proposed metric. Even if equal lengths do not change the order of complexity, they reduce the number of divisions required, the length of subsequent subarrays, and the number of swaps, all of which reduce the elapsed time of the sorting algorithm.

The best pivot to make equal-length subarrays is the median value. To evaluate the median value, a time-consuming process is needed. The mean value is mainly the same as the median value for symmetric and close to that for asymmetric distributions. It requires summing up the N numbers and dividing the result by N with a linear complexity [31].

We demonstrate that the mean value helps us effectively divide the primary array into two subarrays with approximately equal lengths, much better than the random pivot. Also, we highlight the effectiveness of the mean value for different random data in low and high variances, integer/non-integer, when the data set is medium, large, and very large. This analysis is essential because it shows that the created mean-based subarrays have approximately equal lengths. It decreases the time required for the next steps in serial realization and guarantees the same time for independent processors in the parallel realization of the proposed framework.

Assume $N = 2^n$ data at each array and the pivot is any random element of the array, its value may be randomly equal to one of the elements in the data set. Therefore, the absolute value of the difference, $|\Delta L|$, between the sizes of data in the left, L_l, and right, L_r, subarrays around the random pivot is a uniform random variable with the probability of $\frac{1}{N}$ in the range of $[0, N]$. The mean value of this uniform distribution is $\frac{N}{2}$. Hence,

$$L_l - L_r = \pm\frac{N}{2} \tag{36}$$

On the other hand, we know that the sum of data sizes of those mentioned above, upper and lower subarrays around the pivot must be equal to

$$L_l + L_r = N \tag{37}$$

Assuming that the left subarray length is greater than the right subarray length, we get

$$L_l = \frac{3N}{4} \tag{38}$$

$$L_r = \frac{N}{4} \tag{39}$$

Therefore, the length of one subarray is on average three times greater than another. Hence, the average difference

TABLE 5. Speedup factor of the proposed parallel sorting algorithm to that offered by Marszalek and Wozniak in [35], [36].

Number of divisions	Number of processors	Speedup factor							
		Data size							
		2^{10}	2^{13}	2^{15}	2^{17}	2^{20}	2^{24}	2^{27}	2^{30}
1	2	7.7	10.6	12.5	14.5	17.4	21.3	24.3	27.3
2	4	6.2	8.9	10.7	12.6	15.4	19.2	22.1	25
3	8	5.1	7.7	9.4	11.1	13.8	17.5	20.3	23.1
4	16	3.9	6.6	8.3	10	12.5	16	18.7	21.4
5	32	2.25	5.6	7.4	9	11.4	14.8	17.4	20
6	64	–	3.9	6.3	8.1	10.5	13.7	16.2	18.8
7	128	–	1.9	4.5	7	9.7	12.8	15.2	17.6
8	256	–	–	2.2	5.1	8.7	12	14.3	16.7
9	512	–	–	–	2.5	7.1	11.2	13.5	15.8
10	1024	–	–	–	–	4.4	10.2	12.7	15

between the lengths of the right and left subarrays in each division is about 50% of the total length of the main array. It is an essential drawback in serial realization because a higher number of divisions and swaps is needed for larger subarrays. For parallel realization, because the processing time is based on the larger subarray, it experiences three times higher time than the other on average. These facts lead us to find and use appropriate pivots.

In the following subsections, we show that the mean-based pivot used in the proposed framework is an efficient choice for symmetric and asymmetric distributions because, in each division, approximately half of the data will be located in the left subarray and the next half in the right subarray. To demonstrate this effectiveness, the normalized dissimilarity index (NDSI) in each division, i, is defined. It does the average of the absolute value of the difference to the total length of the array over J times of iteration, as (40).

$$NDSI(i) = \frac{1}{J}\sum_{j=1}^{J}\frac{|L_l(i,j) - L_r(i,j)|}{L_l(i,j) + L_r(i,j)} \tag{40}$$

A. SYMMETRIC DISTRIBUTION

Two well-known data distributions are uniform and Gaussian. Most data sorting research works focus on integer/non-integer uniform or integer/non-integer Gaussian data sets. For these data sets, the mean and median values are almost the same, and their statistical distributions are symmetric with zero Skewness evaluated by equations 41, 42 [64], [65].

$$S = \frac{\mu - \nu}{\sigma} \tag{41}$$

$$S = \frac{M_3}{M_2\sqrt{M_2}} \tag{42}$$

where μ and ν are the mean and median values, and M_2 and M_3 are the second and third central moments, respectively.

By averaging over $J = 1000$ times of iteration, NDSI for non-integer uniform data in low and high standard deviations is obtained as depicted in Figure 9. It indicates that the variance of data does not affect the performance of the mean-based scenario compared to the random one. It also shows that the lengths of two subarrays resulting from the mean-based pivot are almost equal in more than 98% of cases. As shown in Figure 10, in high variance integer uniform data, the results for random and mean-based scenarios are almost the same as those for non-integer uniform data. On the contrary, the NDSI for the mean-based pivot for low variance integer uniform data is increased to a value in the range of 4% to 14%, because the possibility of similar data in higher divisions is increased.

As depicted in Figures 11, 12, when we use mean-based pivot for Gaussian data, over 92% similarity of sizes in non-integer data and above 88% in integer data can be achieved. In all data sets mentioned above, the difference between the lengths of two subarrays in the case of random pivot is about 200%.

B. ASYMMETRIC DISTRIBUTION

As described in Figures 9, 10, 11, 12, there is a slight inequality in the length of the subarrays in the higher divisions, especially in the Gaussian distribution. Then, as the number of divisions increases, they tend to have a uniform distribution. Clearly, the results show that this difference in length between the two subarrays is not significant compared to the 200% difference in length between the two subarrays (i.e., the length of one subarray is three times the length of another subarray) made by a random pivot. As the first

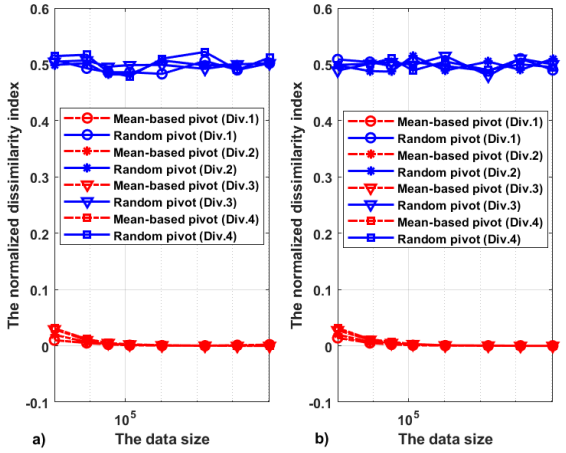

FIGURE 9. NDSI vs. size of non-integer uniform data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

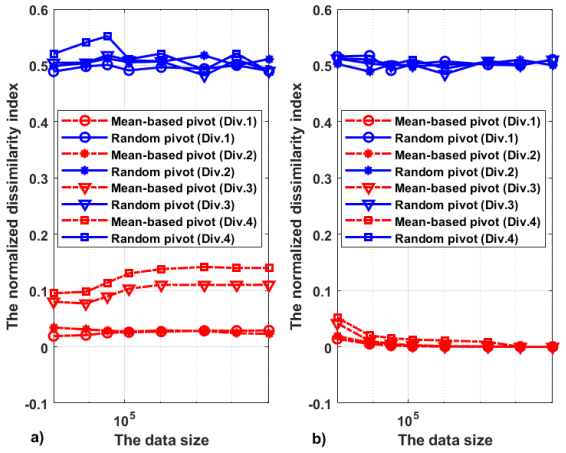

FIGURE 10. NDSI vs. size of integer uniform data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

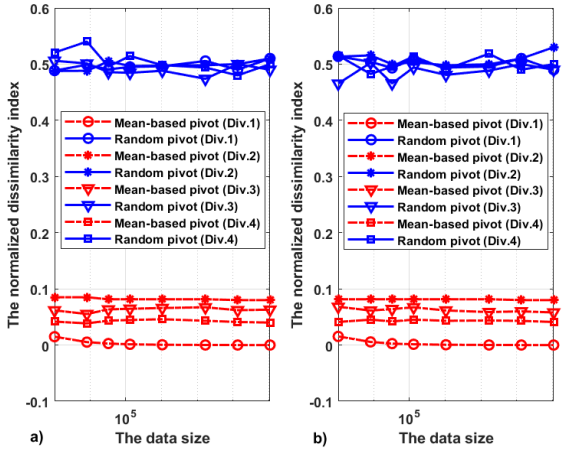

FIGURE 11. NDSI vs. size of non-integer Gaussian data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

division, in the successive divisions where even symmetric statistical distributions would be symmetric (e.g., uniform distribution) or asymmetric (e.g., Gaussian distribution),

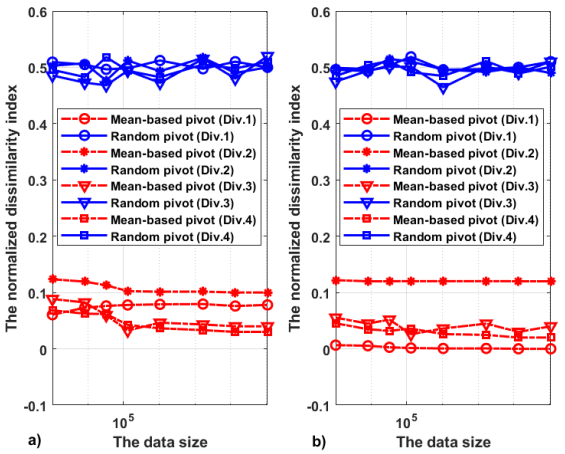

FIGURE 12. NDSI vs. size of integer Gaussian data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

the superiority of the mean-based method is valid. It is also examined for a widespread distribution that initially is asymmetric with non-zero Skewness. As depicted in Figures 13, 14, the superiority of the mean-based pivot to the random-pivot is also valid for the Rayleigh distribution. It means that the data Skewness, data variance, and type of data distribution do not change the effectiveness of the proposed mean-based pivot to the random pivot.

C. SOME USEFUL POINTS

In the simulations performed in this investigation, it was seen that for integer data with slight variance, even if the median is used as a pivot, the length of the two subarrays will not necessarily be equal because having the same data as the balance between the two subarrays in the division is somewhat confusing. Still, the situation is much better than the random pivot. The same is true of using the mean value, especially for integer data with low variance, and has nothing to do with its symmetry or non-symmetry (e.g., Rayleigh and the data in subsequent divisions of Gaussian distribution) and the amount of the Skewness of primary array or subarrays.

Using the existing algorithms to find the median requires the complexity order of $O(N \log N)$. If we add this complexity to the complexity of the sorting process, it will be higher than the complexity of the conventional sorting algorithms such as Merge-sort and Quick-sort. Therefore, in this paper, we have used the mean value, obtained with a linear complexity order. Of course, in medium and large data sets, by averaging a limited number of randomly positioned data (e.g., $\sqrt{N}$ number of data in random locations), the approximate average can be obtained in much less time (i.e., $O(\sqrt{N})$ instead of $O(N)$), which is very close to the exact value of the mean value. In symmetric random data, such as Gaussian and uniform, the mean and median values are almost the same. It can also be seen that in Rayleigh asymmetric distribution or subsequent divisions for the Gaussian distribution, which the distribution will be asymmetric, still, the difference in the lengths of the subarrays around the random pivot is high about

200%. In comparison, it is lower than 20% for subarrays made by the mean-based pivot. As the divisions continue, while reducing the variance of the data in each subarray, the statistical distribution of the subarray will be closer to the uniform distribution. Therefore, the mean value will be more similar to the median.

If the median calculation was cost-effective, it would have been used, but currently, no sorting algorithm uses it. Therefore, the mean-based pivot, equal to the median in symmetric distributions and very close to the median in asymmetric ones, is a time-efficient choice, because it can be calculated with a linear time complexity or less.

Sometimes, the difference between two subarrays is due to the multiple data equal to the mean value that we selected as a pivot. By dividing the data into two subarrays, one including the smaller ones and the second one containing data equal or greater than the pivot (mean value), it makes a difference between them. In this case, duplicating similar data to mean value, is pervasive when the data set includes integer values with low variance. To solve this type of unbalanced subarrays, we can divide similar data between two subarrays, equally. In this way, one subarray includes the smaller and the second one consists of the larger ones. Still, identical elements should be similarly assigned to two subarrays, at the end of one list and the beginning of another list.

After M times of division, the number of data in each subarray is approximately $\frac{1}{2^M}$ times the total number of data, highly sorted compared to the primary data in the exact locations. To have K data from N data after M divisions, we have

$$K = \frac{N}{2^M} + \varepsilon \tag{43}$$

which means it is valid for different data distributions with varying values for ε. The value of ε depends on the proximity of the mean and median values, which vary from one distribution to another. Hence, the required number of divisions is

$$M = \log(\frac{N}{K - \varepsilon}) \tag{44}$$

When the data in a subarray reaches a sufficient number, the subsequent division in that subarray can be stopped. On average, in one division, when two independent subarrays are made, in serial realization, 50% of the sorted data is obtained in 50% of the total elapsed time while both subarrays can be sorted at the same time in parallel realization. In two divisions, four independent subarrays are made that the first one can be sorted in 25%, the second one in 50%, and the third one in 75% of the total elapsed time for serial realization. The same rule is valid for M divisions. The first up to $2^M - 1$ subarrays, each containing an average of $\frac{N}{2^M}$ elements, can be sorted in $\frac{1}{2^M}, \frac{2}{2^M}, \ldots, \frac{(2^M-1)}{2^M}$ of the total elapsed time of serial realization, respectively. It is expected that in parallel realization proposed in this work, we sort 2^M approximately equal-length subarrays simultaneously by using 2^M non-connected processors. As we showed in Figures 7, 8, the required time for parallel processing is about $\frac{1}{2^M}$ times of that for serial realization, especially in large and very large data sets.

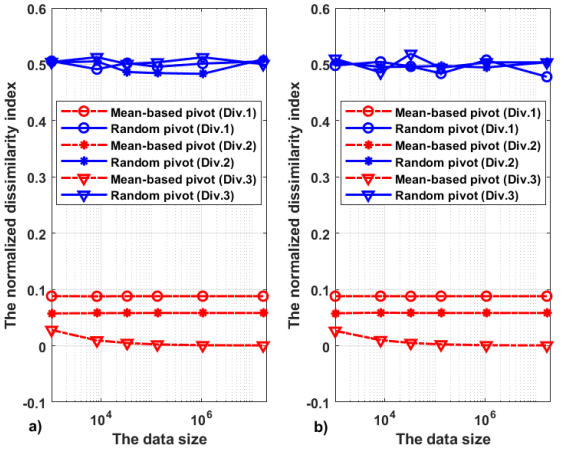

FIGURE 13. NDSI vs. size of non-integer rayleigh data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

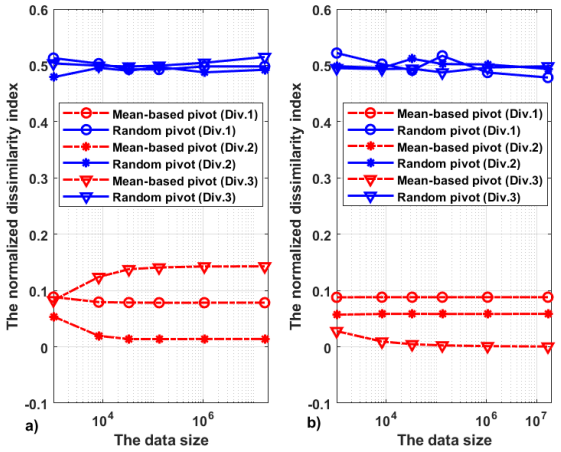

FIGURE 14. NDSI vs. size of integer rayleigh data in the mean-based and random pivot scenarios, a) Low (10), b) High (1000) standard deviations.

Simulation results reported in Figures 9, 10, 11, 12, 13, 14 indicated that subarrays are approximately equal in length for different data types, integer/non-integer symmetric (uniform/Gaussian)/asymmetric (Rayleigh) distributions. Therefore, the superiority of the proposed mean-based sorting algorithms over the conventional ones in serial and parallel realizations can be seen similarly for other types of data in a wide range of variances and different sizes of data, that are not reported here.

VIII. COMPARISON FROM DIFFERENT VIEWPOINTS

Table 6 shows the improvements achieved in the revised versions of the Merge and Quick sorting algorithms compared to the conventional ones in different viewpoints. All algorithms experience a reduction in complexity order. For stable and adaptive sorting algorithms, such as Merge-sort, these features remain while improving for the Quick-sort, which is not stable and adaptive. The revised versions of the Merge, and Quick sorts, can detect sorted parts and

TABLE 6. Comparison from different viewpoints.

Metric	Sorting algorithm			
	Merge		Quick	
	Conventional	Proposed	Conventional	Proposed
The time complexity for the large data set	Moderate	Low	Moderate	Low
Stability	Yes	Yes	No	Moderate
Adaptivity	Yes	Yes	No	Moderate
Ability to detect subarray with sorted data	No	Yes	No	Yes
Ability to detect subarray with similar elements	No	Yes	No	Yes
Ability to sort data gradually	No	Yes	Yes	Yes
Ability to make independent subarrays with approximately equal lengths	No	Yes	No	Yes
Ability to sort data in parallel realization with independent processors	No	Yes	Yes	Yes

parts including similar elements. Besides, the abilities to sort data gradually, useful for serial realization, and making independent subarrays, approximately in equal lengths, appropriate for parallel realization, are achieved.

IX. CONCLUSION

Increasing the volume of data and the number of databases in big data, wired and wireless networks, and many processes that require sorted data justifies why new research in the field of sorting is needed. In this paper, we proposed a general framework suitable for the existing sorting algorithms in medium, large, and very large data sets of orderly, somewhat disordered, and completely disordered (random) scenarios. It first distinguishes whether the data is sorted (increasingly or decreasingly) or contains similar data. Then, the proposed mean-based partitioning step, makes two independent subarrays approximately in equal lengths. These steps will be continued up to a predefined number of divisions, M, making 2^M, independent subarrays. Finally, these subarrays can be sorted by conventional comparison-based sorting algorithms.

Herein, we offered serial and parallel realizations for this framework and analyzed them in terms of time complexity, running time, and the number of swaps. The proposed framework is time-efficient and straightforward applicable for conventional sorting algorithms, that was checked for the Merge and Quick ones. It adds useful features such as gradual sorting, making independent subarrays of approximately equal length with lower standard deviation, and higher stability and adaptivity.

We showed theoretically and numerically that the mean-based pivot is more efficient than the random pivot for symmetric and asymmetric random distributions in three examples, i.e., Gaussian, uniform, and Rayleigh. Based on the simulation results and time complexity analysis, it was shown that the proposed mean-based sorting algorithm does not take up much memory and has a good time complexity, which means a lower number of swaps and shorter running time compared to the conventional Quick and Merge sorts.

The proposed algorithm allows the gradual extraction of sorted data. Besides, each subarray can be processed without interaction with the other ones because the created subarrays are independent. It also enables serial sorts to be processed in parallel form using independent equal-length subarrays. Moreover, the data of one subarray can be sorted first, and the rest of the subarrays can be sorted later. Briefly, four features of the proposed framework are the best.

- Detecting the subarrays with sorted data or including similar data.
- Making independent subarrays with approximately equal lengths.
- Creating the ability to sort the data in a gradual serial realization.
- Adding the feasibility to sort data in a parallel manner using non-connected or connected processors.

X. FUTURE WORK

Related to the framework proposed in this work, six items to achieve more improvements in the future are as follows:

- Optimizing the proposed framework in serial and parallel realizations to have lower space and time complexity.
- Applying the proposed framework to other sorting algorithms and proving its validity and effectiveness in serial and parallel realizations.

- Finding the best pivot instead of the mean value to obtain subarrays in the same length depending on the data distribution and its mean value and variance.
- Hardware implementation of the proposed framework for large and very large data sets in a multi-core structure with a high number of processors and finding the processing time and the number of swaps.
- Comparing the results of the proposed framework in parallel realization to those of standard template library (STL), Edahiro's Map-sort (EM), parallel Merge-sort (MS), Tsigas-Zhang (TZ) parallel Quick-sort, Tsigas-Zhang job list (TZJL), and alternative Quick-sort (AQ) algorithms reported in [66] in terms of sorting throughput, scalability, CPU affinity, and micro-architecture analysis.
- Combining the proposed versions of the Merge and Quick sorts presented in this investigation to the multi-core parallel implementations reported in [35], [36], [60]–[63] in two cases, when the number of divisions is lower or higher than the number of processors.

REFERENCES

[1] 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.
[2] S. Shirvani Moghaddam, "Keynote talk: Data sorting for large data sets," presented at the Workshop Microw. Theory Techn. Wireless Commun. (MTTW), Riga, Latvia, Oct. 7–8, 2021. [Online]. Available: https://www.researchgate.net/publication/355357231
[3] *Quora*. Accessed: Nov. 7, 2021. [Online]. Available: https://www.quora.com
[4] I. Valova and M. Noirhomme-Fraiture, "Processing of large data sets: Evolution, opportunities and challenges," presented at the PCaPAC, Ljubljana, Slovenia, Oct. 20–28, 2008.
[5] S. Shirvani Moghaddam and K. Shirvani Moghaddam, "Efficient base-centric/user-centric clustering algorithm based on thresholding and sorting," presented at the 14th IEEE Int. Conf. Innov. Inf. Technol. (IIT), AlAin, UAE, 2020.
[6] F. L. Yaeger, "DF sort: Beyond sorting," IBM Syst. Softw. Develop., San Jose, CA, USA, Tech. Rep., Oct. 2010. [Online]. Available: https://dokumen.tips/download/link/dfsort-beyond-sorting
[7] A. D. Chapman, "Principles and methods of data cleaning: Primary species and species-occurrence data," Rep. Global Biodiversity Inf. Facility, Copenhagen, Denmark, Tech. Rep., 2005.
[8] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. San Mateo, CA, USA: Morgan Kaufmann, 2012.
[9] S. Shirvani Moghaddam, *Cognitive Radio in 4G/5G Wireless Communication Systems*. Rijeka, Croatia: InTechopen, vol. 2018, doi: 10.5772/intechopen.74815.
[10] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Comput. Sci.*, vol. 2, p. 160, Mar. 2021, doi: 10.1007/s42979-021-00592-x.
[11] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of big data challenges and analytical methods," *J. Bus. Res.*, vol. 70, pp. 263–286, Jan. 2017.
[12] 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.
[13] *Geeksforgeeks*. Accessed: Jul. 11, 2021. [Online]. Available: https://www.geeksforgeeks.org
[14] *Programix*. Accessed: Jul. 15, 2021. [Online]. Available: https://www.programix.com
[15] 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.
[16] A. Zutshi and D. Goswami, "Systematic review and exploration of new avenues for sorting algorithm," *Int. J. Inf. Manage. Data Insights*, vol. 1, no. 2, 2021, Art. no. 100042.
[17] S. Rana, M. A. Hossin, S. M. H. Mahmud, and H. Jahan, "MinFinder: A new approach in sorting algorithm," presented at the 9th Int. Congr. Inf. Commun. Technol., Guangxi, China, 2019.
[18] A. Alotaibi, A. Almutairi, and H. Kurdi, "OneByOne (OBO): A fast sorting algorithm," presented at the 15th Int. Conf. Future Netw., Commun., Leuven, Belgium, 2020.
[19] 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, 2016.
[20] I. Hayaran and P. Khanna, "Couple sort," presented at the 4th Int. Conf. Parallel, Distrib. Grid Comput., Waknaghat, India, 2016.
[21] A. Maus, "A faster all parallel Mergesort algorithm for multicore processors," Presented at the Norwegian Informat. Conf., Oslo, Norway, 2018.
[22] M. Shabaz and A. Kumar, "SA sorting: A novel sorting technique for large-scale data," *J. Comput. Netw. Commun.*, vol. 2019, Jan. 2019, Art. no. 3027578.
[23] W. Xiang, "Analysis of the time complexity of Quick sort algorithm," presented at the Int. Conf. Inf. Manage., Innov. Manage. Ind. Eng., Shenzhen, China, 2011.
[24] 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.
[25] M. Ali, Z. Nazim, W. Ali, A. Hussain, N. Kanwal, and M. K. Paracha, "Experimental analysis of On(log n) class parallel sorting algorithms," *Int. J. Comput. Sci. Netw. Secur.*, vol. 20, no. 1, pp. 139–148, Jan. 2020.
[26] 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.
[27] G. Maier, F. Pfaff, C. Pieper, R. Gruna, B. Noack, H. Kruggel-Emden, T. Langle, 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.
[28] 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.
[29] 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.
[30] 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.
[31] S. Shirvani Moghaddam and K. Shirvani Moghaddam, "On the performance of mean-based sort for large data sets," *IEEE Access*, vol. 9, pp. 37418–37430, 2021.
[32] Z. Marszalek, "Parallelization of modified Merge sort algorithm," *Symmetry*, vol. 9, no. 176, pp. 1–18, 2017.
[33] 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, 2019.
[34] 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, 2011.
[35] Z. Marszałek, M. Woźniak, and D. Połap, "Fully flexible parallel merge sort for multicore architectures," *Complexity*, vol. 2018, pp. 1–19, Dec. 2018.
[36] Z. Marszałek, "Parallel fast sort algorithm for secure multiparty computation," *J. UCS*, vol. 24, no. 4, pp. 488–514, 2018.
[37] 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.
[38] 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.
[39] Y. Yung, P. Yu, and Y. Gan, "Experimental study on the five sort algorithms," presented at the 2nd Int. Conf. Mechanic. Automat. Control Eng., Hohhot, China, 2011.
[40] K. Shirvani Moghaddam and S. Shirvani Moghaddam, "Sorting algorithm for medium and large data sets based on multi-level independent subarrays," presented at the 10th IEEE Int. Conf. Commun., Netw. Satell., Purwokerto, Indonesia, Jul. 2021.

[41] P. Ganapathi and R. Chowdhury, "Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort," *Comput. J.*, Aug. 2021, doi: 10.1093/comjnl/bxab107.
[42] A. Shatnawi, Y. AlZahouri, M. A. Shehab, Y. Jararweh, and M. Al-Ayyoub, "Toward a new approach for sorting extremely large data files in the big data era," *Cluster Comput.*, vol. 22, no. 3, pp. 819–828, Sep. 2019.
[43] A. S. Gowtham and D. Jaya Kumar, "Implementation of sorting of one-dimensional array using RAM based sorting algorithm," *J. Crit. Rev.*, vol. 7, no. 15, pp. 5904–5914, 2020.
[44] A. I. El Nashar, "Parallel performance of MPI sorting algorithms on dual-core processor windows-based systems," *Int. J. Distrib. Parallel Syst.*, vol. 2, no. 3, pp. 1–14, May 2011.
[45] M. Marzolla and G. D'Angelo, "Parallel data distribution management on shared-memory multiprocessors," *ACM Trans. Model. Comput. Simul.*, vol. 30, no. 1, pp. 1–25, Feb. 2020.
[46] A. Nancy, I. Ravishankar, S. Sharad, and W. Yan, "A Comparison of parallel sorting algorithms on different architectures," Texas A & M Univ., College Station, TX, USA, Tech. Rep. 10.5555/892860, Jan. 1999.
[47] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770–785, 1988.
[48] L. Zeng, "Two parallel sorting algorithms for massive data," presented at the 2021 IEEE Int. Conf. Artif. Intell. Comput. Appl., Dalian, China, 2021.
[49] D. N. Raju, "An efficient new approach mean based sorting," presented at the IEEE UP Sect. Conf. Elect. Comput. Electron., Allahabad, India, 2015.
[50] 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, 2008.
[51] F. Luo, L. Khan, F. Bastani, I. L. Yen, and J. Zhou, "A dynamically growing self-organizing tree (DGSOT) for hierarchical clustering gene expression profiles," *Bioinformatics*, vol. 20, no. 16, pp. 2605–2617, 2004.
[52] M. Axtmann and P. Sander, "Robust massively parallel sorting," presented at the Meeting Algorithms Eng. Exp. (Alenex), Barcelona, Spain, 2017.
[53] K. H. Kwon, "Parallel computation on hypercube-like machines," Ph.D. dissertation, Agricult. Mech. College, Louisiana State Univ., Baton Rouge, LA, USA, 1991.
[54] P. Krusche and A. Tiskin, "Efficient parallel string comparison," presented at the Parallel Comput., Archit., Algorithms, Appl. (ParCo), Aachen, Germany, 2007.
[55] *Stackoverflow*. Accessed: Nov. 9, 2021. [Online]. Available: https://stackoverflow.com
[56] Y. Chee, "A real elementary approach to the master recurrence and generalizations," presented at the 8th Annu. Conf. Theory Appl. Models Comput., Tokyo, Japan, 2011.
[57] 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.
[58] 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, 2019.
[59] 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.
[60] M. Wozniak, Z. Marszalek, M. Gabryel, and R. Nowicki, "Triple Heap sort algorithm for large data sets," presented at the 8th Int. Conf. Knowl., Inf. Creativity Support Syst., Cracow, Poland, 2015.
[61] Z. Marszalek, M. Wozniak, G. Borowik, R. Wazirali, C. Napoli, G. Pappalardo, and E. Tramontana, "Benchmark tests on improved Merge for big data processing," presented at the Asia–Pacific Conf. Comput. Aided Syst. Eng., Quito, Ecuador, 2015.
[62] L. Yang and L. Jin, "Integrating parallel algorithm design with parallel machine models," presented at the 26th SIGCSE Tech. Symp. Comput. sci. Educ., Nashville, TN, USA, 1995.
[63] M. Woźniak, Z. Marszalek, M. Gabryel, and R. Nowicki, "Preprocessing large data sets by the use of quick sort algorithm," in *Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions* (Advances in Intelligent Systems and Computing), vol. 364. 2015, pp. 111–121, doi: 10.1007/978-3-319-19090-7_9.
[64] D. P. Doane and L. E. Seward, "Measuring skewness: A forgotten statistic?" *J. Statist. Educ.*, vol. 19, no. 2, pp. 1–18, 2011.
[65] G. Brys, M. Hubert, and A. Struyf, "A comparison of some new measures of skewness," in *Developments in Robust Statistics Physica*. Heidelberg, Germany, 2003, pp. 97–112, doi: 10.1007/978-3-642-57338-5_8.
[66] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," in *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Lang. Appl. Companion*, New York, NY, USA, 2011, doi: 10.1145/2048147.2048207.

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 (IUST), Tehran, Iran, in 2001. Since 2003, he has been with the Faculty of Electrical Engineering, Shahid Rajaee Teacher Training University (SRTTU), Tehran, where he is currently an Associate Professor. Besides having numerous articles in prestigious scientific journals, he has presented dozens of papers in national and international conferences. Also, he is the author of two books, one on digital communications and another on electrical engineering, and an editor of one book about cognitive radio (CR). His research interests include resource allocation and power control in CR-based networks, ultra-dense networks (UDNs), heterogeneous networks (HetNets), device-to-device (D2D) and vehicle-to-vehicle (V2V) communications, data sorting, 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 (IUST). He has been the Lecturer of some courses about C and C++ programming languages, LATEX software, fundamentals of computer engineering, and Rubik's cube. He is also a Teaching Assistant at IUST. His research interests include developing web and windows applications and algorithm design in cross-platform languages.