Large-Scale Data Sorting Using Independent Equal-Length Subarrays

Shahriar Shirvani Moghaddam (sh_shirvani@sru.ac.ir)
Shahid Rajaee Teacher Training University
Kiaksar Shirvani Moghaddam
Iran University of Science and Technology

Research Article

Keywords: wireless networks, subarrays, uniform/Gaussian data, algorithms

Posted Date: May 27th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-530919/v1

Large-Scale Data Sorting Using Independent Equal-Length Subarrays

Shahriar Shirvani Moghaddam[1,*] **and Kiaksar Shirvani Moghaddam**[2]

[1]Faculty of Electrical Engineering, Shahid Rajaee Teacher Training University, Tehran, 1678815811, Iran
[2]School of Computer Engineering, Iran University of Science and Technology, Tehran, 1684613114, Iran
[*]sh_shirvani@sru.ac.ir

ABSTRACT

Design an efficient data sorting algorithm that requires less time and space complexity is essential for large data sets in wireless networks, the Internet of things, data mining systems, computer science, and communications engineering. This paper proposes a low-complex data sorting algorithm that distinguishes the sorted/similar data, makes independent subarrays, and sorts the subarrays' data using one of the popular sorting algorithms. It is proved that the mean-based pivot is as efficient as the median-based pivot for making equal-length subarrays. The numerical analyses indicate slight improvements in the elapsed time and the number of swaps of the proposed serial Merge-based and Quick-based algorithms compared to the conventional ones for low/high variance integer/non-integer uniform/Gaussian data, in different data lengths. However, using the gradual data extraction feature, the sorted parts can be extracted sequentially before ending the sorting process. Also, making independent subarrays proposes a general framework to parallel realization of sorting algorithms with separate parts. Simulation results indicate the effectiveness of the proposed parallel Merge-based and Quick-based algorithms to the conventional serial and multi-core parallel algorithms. Finally, the complexity of the proposed algorithm in both serial and parallel realizations is analyzed that shows an impressive improvement.

Introduction

There is a large number of data in the heterogeneous networks (HetNets), Internet of things (IoT) systems, Wireless sensor networks (WSN), and signal/information processing and data fusion in social networks, as well as advanced communication, localization, and object tracking systems. The data should be preprocessed by a low-complex time-efficient data/information sorting. Sorting is the essential process of ordering the non-sorted data and changing the position of elements in a data set, which can be an array, list, vector, or matrix, in ascending or descending order[1]. 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 time-efficient sorting algorithm.

Many sorting algorithms have been enhanced in view of time and space complexity, type of realization, etc.[2] 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[3]. Most of the data sorting algorithms are based on comparisons like those that are 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[4,5]. In contrast, Radix, Counting, Bucket, Pigeonhole, and Tag are non-comparison-based algorithms just for sorting integer data[6].

To sort large data sets, running (or processing) time of huge number of records is considerable and complexity cannot be ignored[7]. Therefore, designing an efficient sorting approach is required[8] that takes less time and space complexity[9] and comprises of comparison, swapping, and assignment operations[10]. Also, the simplicity of the algorithm, its computational complexity, the difference in algorithm execution time for unsorted and somewhat sorted data, the propagation of an error in the sorting process in case of interruption or error in a part of the process, the memory required for the algorithm, and its application, indicate the need to update sorting algorithms[11,12]. In other words, some issues have led to the need to modify data sorting algorithms, propose new algorithms, and making restrictions in the data sorting process, with the increasing advances of human beings in various fields of science and engineering, especially computer science and electrical engineering.

The two most commonly used sorting algorithms, Quick and Merge, are in the category of divide-and-conquer methods, which work well for unordered medium or large data sets. The Quick-sort makes two non-equal separate subarrays in each division, while the Merge-sort divides each array into dependent subarrays with equal lengths. The performance of these algorithms depends on the size of data and the randomness rate. Hence, in this research, we propose a sorting algorithm based

on a mean-based segmentation and one of the Merge and Quick sorts. Besides improving the elapsed time and number of swaps, the proposed algorithm allows gradual extraction of sorted data. Also, It provides a framework for parallel realization of independent subarrays approximately equal in length.

The leading question of this investigation is this: Is it possible to do the following items by proposing a new low-complex preprocessing or modification on the popular sorting algorithms?

- Improve the performance of the existing sorting algorithms such as Merge and Quick sorts that can easily be implemented.
- Change the sorting procedure to find a new version with almost similar complexity for different data types and lengths.
- Make independent subarrays of the primary data that are approximately in the same lengths.
- Find the sorted data gradually before ending the sorting process.
- Distinguish the sorted data, entirely or partly, in different levels of division.
- Hold the stability and adaptivity features for the improved versions of the stable and adaptive algorithms.
- Increase the stability and adaptivity features for the improved versions of the non-stable and non-adaptive algorithms.

Why Is It Necessary to Propose a Sorting Algorithm?

Nowadays, the concept of big data and very large and widespread databases come into existence. Researchers in computer science and all engineering disciplines strongly agree that data sorting is essential because this topic is necessary and considered by different branches of engineering sciences. In this investigation, by proposing and applying a new idea, the performance of the two popular sorting algorithms, i.e., Merge and Quick, has been improved. There is a question that why and what is the justification for the new research in this field? The response is that sorting is very much considered today because the volume of data and the number of databases has increased in big data, wired and wireless networks, and many processes that require sorted data. In an investigation published in[13], and other research works[14–21] explore this issue with different ideas. It shows that data sorting is a multidisciplinary subject at the edge of science.

Two popular algorithms in data sorting are Merge and Quick. These algorithms are in the category of divide-and-conquer comparison-based methods appropriate for medium, large, and very large data sets. Although quick-sort is an in-place sort but in some cases experiences stack overflow error. In contrast, Merge-sort is not an in-place sort and needs an $O(N)$ space complexity[1,4]. Merge-sort uses merge and mergesort functions. The mergesort function recursively divides the array into two halves and calls the merge function for each of the two halves. The merge function combines both halves after sorting[8,21]. Quick-sort selects an element as a pivot and divides the given array around the pivot, one greater and another smaller than the pivot. This segmentation continues for each subarray until we find single-member subarrays[12,14]. Supposing that the data length is N, the Quick and Merge sorts have the memory complexity of order $O(NlogN)$ and $O(N)$, respectively. In the worst, average, and best cases of the Merge-sort, the time complexity is $O(NlogN)$, 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 $O(NlogN)$[4,6,15].

Quick and Merge sorting algorithms are suitable for long unordered and highly disordered data, non-integer and integer, respectively[16]. In 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 Quick-sort that makes two unequal subarrays around the pivot capable to be sorted in a parallel fashion, mainly Merge-sort is in serial form and can be sorted using multicore processing systems in non-independent parallel parts[10,15,17]. The ideas reported in[10,15,18,19] were examined for the Merge and Quick sorts in parallel processing. The results were compared to the conventional Merge and Quick sorts by proposing a division of tasks between processors. The proposed algorithm was described in a theoretical way, examined in tests, and compared to other algorithms. The experimental and numerical results showed that adding each processor causes sorting to become faster and more efficient, especially for large data sets[20]. 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 idea to make independent equal-length subarrays sorted one-by-one in a multi-core realization that each core processes one subarray. The purpose of this paper and the explanations, figures, and tables are to express the issue, the proposed solution, and the results obtained using a low-complex preprocessing applicable for most of the existing sorting algorithms.

To find an algorithm applicable for medium, large, and very large integer/non-integer data sets, in the next section, we propose an algorithm based on the mean-based segmentation combined with one of the two popular sorting algorithms. It offers some new features and removes the bottlenecks of the algorithms mentioned above. To display the effectiveness of the proposed idea on the performance of the Merge and Quick sorts, in subsequent sections, we do numerical analyses to:

- Evaluate the processing time and the number of swaps required for integer/non-integer uniform and Gaussian data in medium, large, and very large data sets.
- Show the effectiveness of the mean-based pivot to make equal-length independent subarrays in medium, large, and very large data sets.
- Demonstrate the superiority of the parallel realizations of the proposed algorithms to the serial realizations and conventional parallel implementations.
- Compare the proposed and conventional algorithms in view of the time complexity order in the best, worst, and average cases, and other viewpoints.

The Proposed Data Sorting Algorithm

In this investigation, instead of proposing a new sorting algorithm, we improve the total performance of the sorting process by applying a low-complex preprocessing idea based on the mean value that makes independent subarrays in approximately equal lengths and let us do the sorting in both serial and parallel realizations, which can be applied to the existing sorting algorithms easily. Generally, the favorable features due to this idea are valid for any sorting algorithm that may be used after preprocessing. When the size of records is large or very large, Quick-sort and Merge-sort perform well and outperform the other sorting algorithms[1,21]. Hence, without losing the generality, in the following sections these popular sorting algorithms are examined.

The main idea is to distinguish the sorted and similar data, divide the primary data array into 2^n independent subarrays by comparing data to the mean-value, and perform the well-known algorithms appropriate for unsorted or almost sorted small data sets located in subarrays. The new algorithm includes three phases. In the first phase of each level of division, it distinguishes whether the data are unsorted, sorted, or similar. In the second phase of each level, 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 the predefined number of divisions. After getting the smaller data sets in the final unsorted subarrays, they will be sorted using one of the well-known sorts, i.e., Merge and Quick, entitled core-sort. Algorithm 1 presents the C# pseudo-code of the proposed algorithm. The proposed algorithm divides the data into two parts, which can be examined independently in the following steps. Therefore, after sorting the data in each subarray, there is no need to compare the data in different subarrays, and the location of the sorted data will not change.

It can adapt the performance based on the data length, randomness rate, running time, and the number of processors by changing the number of divisions. Also, it enables Merge-sort to sort data gradually, which is not possible in the conventional version of this algorithm. Moreover, it offers a general framework for parallel realizations with independent processors applicable to all types of sorting algorithms, such as Insertion and Merge sorts, 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 higher level of adaptivity and stability by applying the proposed idea for the Quick-sort, which is mainly not an adaptive and stable sort. It means that in the different divisions of the proposed idea, if the primary data includes sorted parts or there are similar data partially, it will not be disturbed.

Numerical Analyses and Simulation Results

This section and consequent ones try to show the effectiveness of the proposed idea on the data sorting. Therefore, the performance of the two well-known sorting algorithms that are equipped with the proposed mean-based idea are compared to the conventional ones based on the required elapsed (running) time, the number of swaps, and the complexity order. The type of data, such as sorted, somewhat sorted, and random, the length of data, i.e., 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.

Evaluations and comparisons are based on the average required running time and the average number of swaps for sorting integer/non-integer uniform/Gaussian random data with 2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}, 2^{24}, 2^{27}, 2^{30}, lengths and low (10) and high (1000) standard deviations. The simulations are run in four data sets, i.e., integer/non-integer uniform and integer/non-integer Gaussian. The trend of the results is the same as that for non-integer uniform data. Hence, in this section, we focus on the sorting non-integer uniform data in a descending order. 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 package 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.

Algorithm 1 The pseudo-code of the proposed mean-based sorting algorithm in C#.

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

```
Function Sort(arr, m):
    cnt ⟵ 0    iCnt ⟵ 0
    Division(arr, 0, arr.Length − 1, m)
    for i ← 0 to (indexes.Length / 2) − 1 do
        if status[i] = 0 then
            sort from arr[indexes[2 ∗ i]] to arr[indexes[2 ∗ i + 1]] using Core-Sort
        else if status[i] = 2 then
            reverse from arr[indexes[2 ∗ i]] to arr[indexes[2 ∗ i + 1]]
        end
    end
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
        indexes[2 ∗ iCnt] ⟵ si
        indexes[2 ∗ iCnt + 1] ⟵ ei
        iCnt ⟵ iCnt + 1
    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
```

Assuming that there is no similar or sorted data in phase 1 of each level, namely *status*, we compare the performance of the proposed algorithm 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 Figures 1, 2, and those in Figures 7, 8, consequently. The results of the proposed algorithm 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.

Performance Evaluation of Serial Realization

Figures 1 and 2 demonstrate the performance metrics for the Merge-sort and Quick-sort, respectively. The proposed idea decreases the elapsed time and number of swaps in both sorting algorithms, because it sorts the whole data in 2, 4, 8 or 16 independent subarrays, respectively in 1, 2, 3, and 4 divisions. On the contrary, calculating the mean value, comparing each layer'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, the total time and number of swaps for both Merge-sort and Quick-sort are close to the associated values for the conventional ones.

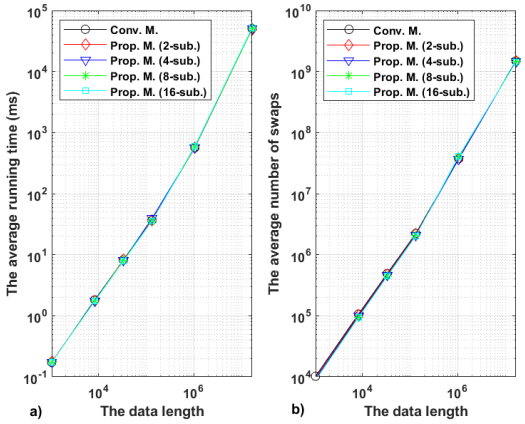

Figure 1. The performance of the proposed and conventional Merge-sort, (**a**) Average running time, (**b**) Average number of swaps.

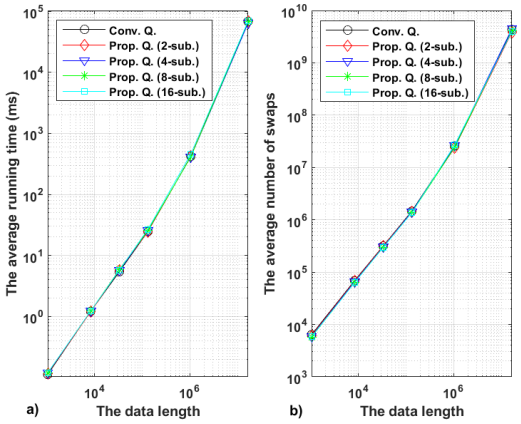

Figure 2. The performance of the proposed and conventional Quick-sort, (**a**) Average running time, (**b**) Average number of swaps.

Simulations show slight improvements in the time and complexity when it uses Merge and Quick sorts as the core-sort. It is shown that the number of swaps and the elapsed time of the proposed mean-based Quick and Merge sorts have been slightly improved compared to the conventional ones. Using the proposed idea, 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 some 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 data, which are not reported here to avoid repeating similar results.

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

Moreover, the proposed algorithm can sort and extract data gradually, which is not possible in the conventional Merge-sort. In this view of point, it also improves Quick-sort's performance because the subarrays are almost in a same length. It means that the required time to extract the sorted parts of data can be estimated.

Effectiveness of the Mean-Based Divisions

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 non-symmetric distributions. It needs summing up the N numbers and dividing the result by N with a linear complexity[13].

We show that the mean value helps us to divide the primary array into two subarrays effectively with approximately equal lengths, much better than the random pivot. Also, we show 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 in size. This test is essential because it shows that the created subarrays have approximately equal lengths. It decreases the time required for the next steps of the proposed sorting algorithms in serial and parallel realizations.

Assuming $N = 2^n$ 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 lengths of data in the left, L_l, and right, L_r, subarrays around the 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,

$$L_l - L_r = \pm\frac{N}{2} \tag{1}$$

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

$$L_l + L_r = N \tag{2}$$

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

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

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

Therefore, the length of one subarray is on average three times greater than another. Hence, the average difference 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 parallel processing. Since the evaluation of parallel processing is based on the larger subarray, it experiences three times higher time than the other on average. In contrast, in the proposed mean-based algorithm, in each division, approximately half of data is in the left subarray and the next half in the right subarray.

By averaging over $J = 1000$ times of iteration, the normalized dissimilarity index (NDSI) in each division i, equation 5, for non-integer uniform data in low and high standard deviations is obtained as depicted in Figure 3. It demonstrates 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 approximately equal to each other in more than 98% of cases.

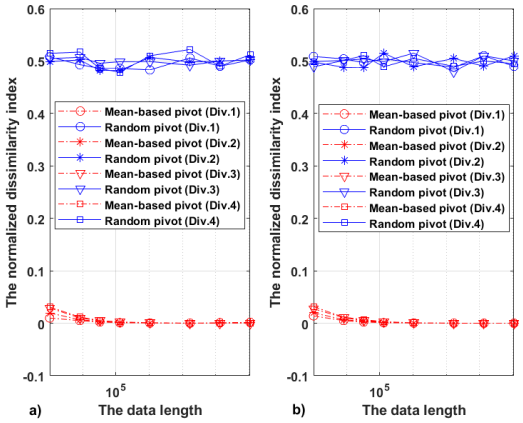

Figure 3. NDSI vs. length of non-integer uniform data in the mean-based and random pivot scenarios, (**a**) Low (10), (**b**) High (1000) standard deviations.

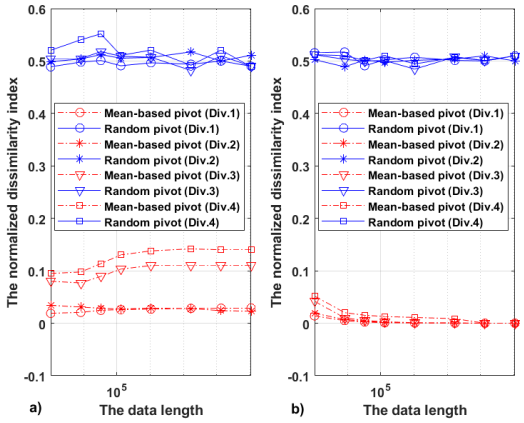

Figure 4. NDSI vs. length of integer uniform data in the mean-based and random pivot scenarios, (**a**) Low (10), (**b**) High (1000) standard deviations.

$$NDSI(i) = \frac{1}{J}\sum_{j=1}^{J}\frac{|L_l(i,j) - L_r(i,j)|}{L_l(i,j) + L_r(i,j)} \tag{5}$$

As shown in Figure 4, in high variance integer uniform data, the results for both 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 5, 6, when we use mean-based pivot for Gaussian data, over 92% similarity when data are non-integer and above 88% in integer data can be achieved. In all above-mentioned data sets, the difference between the lengths of two subarrays in the case of random pivot is about 200%.

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 same locations. To have K data from N data after M divisions, we have

$$K = \frac{N}{2^M} + \varepsilon \tag{6}$$

which means equation (6) is valid for different data distributions with different values for ε. The value of ε depends on the proximity of the mean and median values, which depends on the data distribution. Hence, the required number of divisions is

$$M = \log_2\left(\frac{N}{K-\varepsilon}\right) \tag{7}$$

When the number of 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, 50% of the sorted data is obtained in 50% of the total elapsed time. 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. The same rule is valid for M divisions, and 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 total elapsed time, respectively.

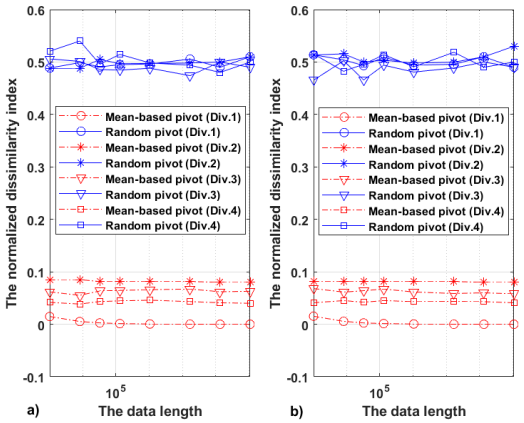

Figure 5. NDSI vs. length of non-integer Gaussian data in the mean-based and random pivot scenarios, (**a**) Low (10), (**b**) High (1000) standard deviations.

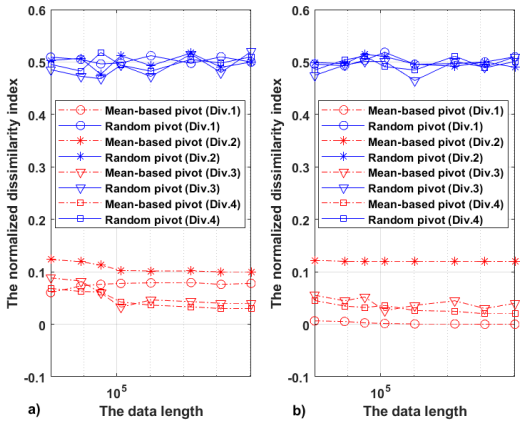

Figure 6. NDSI vs. length of integer Gaussian data in the mean-based and random pivot scenarios, (**a**) Low (10), (**b**) High (1000) standard deviations.

Performance Evaluation of Parallel Realization

Nowadays, a typical personal computer CPU has 2, 4, 6, or 8 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[18,19,22] to achieve a fast sorting algorithm. Most of the sorting algorithms can be just realized in a serial form or semi-parallel fashion. Still, some small parts can be realized either serial or parallel with independent parts, such as Quick-sort[23]. 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 semi-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 idea in this work makes approximately equal length subarrays that really can be sorted by independent processors in a realistic parallel realization.

Generally, the running time required for a subarray with a longer length determines the elapsed time in the parallel processing. By making two independent subarrays approximately equal in size, 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.

Figures 7, 8 demonstrate the performance metrics for the conventional and proposed parallel versions of the Merge-sort and Quick-sort, respectively. Although the proposed idea does not make a noticeable improvement in the required elapsed time and number of swaps for the Merge-sort and Quick-sort in the serial realization, it allows the Merge-sort to be realized in a parallel fashion that was possible for Quick-sort. By making 2, 4, 8, and 16 independent subarrays respectively in 1, 2, 3, and 4 divisions, we 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. In the improved parallel version of the Quick-sort, the elapsed time and number of swaps reduce higher than the factors of 2, 4, 8, and 16, because the subarrays produced by the mean-based pivot are much closer in length. It meets a lower number of swaps and sub-divisions than the conventional Quick-sort.

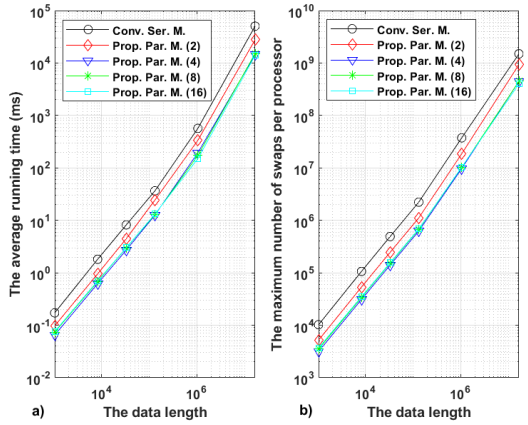

Figure 7. The performance of the proposed parallel and conventional Merge-sort, (**a**) Average running time, (**b**) Maximum number of swaps per processor.

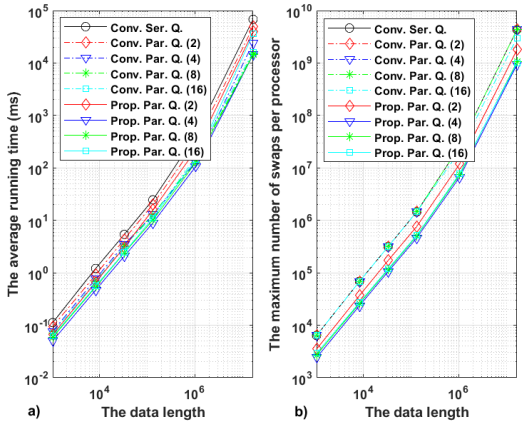

Figure 8. The performance of the proposed and conventional parallel Quick-sort, (**a**) Average running time, (**b**) Maximum number of swaps per processor.

Simulation results reported in Figures 3, 4, 5, 6 indicated that subarrays are approximately in equal length for different data types, integer/non-integer uniform/Gaussian distributions. Therefore, the superiority of the proposed mean-based sorting algorithms over the conventional ones can be seen similarly for integer uniform and integer/non-integer Gaussian data in serial and parallel realizations for a wide range of variances and different lengths of data.

Time Complexity Analysis

In this section, the time complexity of the proposed mean-based Quick and Merge algorithms is analyzed. This complexity consists of two parts. The first part shows the time complexity of phase one, which includes the time to compute counter and reversing the data if they are reverse 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. Otherwise, if it is zero, the data are similar.

In each level, we need to find the sign of differences, calculate the counter, reverse the data if it is reversely sorted, find the mean value, and make two subarrays. Each of these calculations needs a linear time complexity of $O(N)$. The proposed

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	$O(M(3N+P))+O(N\log\frac{N}{2^M})$	$O(M(3N+P))+O(N\log\frac{N}{2^M})$	$O(2N)$
	Proposed parallel (per processor)	$O(M(3N+P))+O(\frac{N}{2^M}\log\frac{N}{2^M})$	$O(M(3N+P))+O(\frac{N}{2^M}\log\frac{N}{2^M})$	$O(2N)$
Merge	Conventional	$O(N\log N)$	$O(N\log N)$	$O(N\log N)$
	Proposed serial	$O(2N)$	$O(M(3N+P))+O(N\log\frac{N}{2^M})$	$O(3N)$
	Proposed parallel (per processor)	$O(2N)$	$O(M(3N+P))+O(\frac{N}{2^M}\log\frac{N}{2^M})$	$O(3N)$

Table 1. The time complexity of the conventional and proposed Quick and Merge sorts in the best, average, and worst cases.

algorithm needs a linear complexity of $O(N)$ for the memory space complexity, when core-sort is one of the Merge and Quick sorts. Accordingly, in the first division, if data are sorted or have similar data, $2O(N)$ time complexity is required. If data are reversely sorted, it needs $3O(N)$ time complexity. When data are none of the above cases, we need to calculate the mean value and make two subarrays.

The upper bound of the time complexity in the first level is $O(4N)$. After the first division, assuming that there is no sorted/similar data, we have two $\frac{N}{2}$-member subarrays. In the second division, an upper bound of time complexity of $4\times 2O(\frac{N}{2})$ is required for the two existing subarrays. After that, assuming no sorted/similar data, we have four $\frac{N}{2^2}$-member subarrays. Finally, in the Mth division, it is $4\times 2^{M-1}O(\frac{N}{2^{M-1}})$. 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 both serial and parallel realizations proposed in this investigation, the following upper bound of time complexity of phase one is required:

$$4O(N)+4\times 2O(\frac{N}{2})+4\times 2^2O(\frac{N}{2^2})+\ldots+4\times 2^{M-1}O(\frac{N}{2^{M-1}})=O(4MN) \tag{8}$$

To decrease the complexity order of the mean value evaluation for medium, large, and very large data sets, the approximate mean value can be used. The type of data distribution and data variance have neglected effect on the exact and approximate mean values[13]. By this modification, $O(4MN)$ can be decreased to $O(M(3N+P))$, assuming P significantly smaller than N. Therefore, considering this modification and the time complexities caused by the conventional Quick and Merge sorts in the average-case as well as the above-mentioned evaluations for the best-case and worst-case of the proposed algorithms, we have Table 1. It depicts the time complexity for the proposed Quick and Merge sorts in three scenarios, i.e., the conventional, the proposed serial, and the proposed parallel per processor. In this table, $O(M(3N+P))$ is the upper bound associated with the time complexity of phase one, and the second part is related to the core-sort's time complexity.

The worst-case of the conventional Quick-sort happens when the array is already sorted, increasingly or decreasingly, or all data elements are similar. If the counter is $(N-1)$, it means that data are sorted. In the case of similar data, it is zero. The proposed algorithm needs a time complexity of $O(2N)$. Hence, the total required time complexity for the worst-case in both serial and parallel realizations of the proposed Quick-sort is $O(2N)$.

The best-case for the Merge-sort happens when data are sorted or all elements are similar. As mentioned above, its time complexity is $O(2N)$. In the worst-case for this algorithm, the data are sorted in reverse order. In addition to the mentioned above time complexity, reversing the data needs an $O(N)$ time complexity, i.e., the total time complexity is $O(3N)$. In division M, 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 average-case is the upper bound, and most of the time, the data sorting needs a lower complexity.

Assuming 2^M equal-length subarrays, the difference between the time complexity of the serial and parallel processing for the average-case of all algorithms and the best-case for the Quick-sort is a factor of 2^M. In parallel processing, the time required for one of 2^M independent parallel processors, which takes longer to process, should be considered. For the cases that core-sort time complexity is $O(NlogN)$, the proposed algorithm in serial and parallel realizations needs $2^MO(\frac{N}{2^M}log\frac{N}{2^M})$ and $O(\frac{N}{2^M}log\frac{N}{2^M})$, respectively. When the number of divisions is as small as the length of the primary data, then the complexity for the serial and parallel realizations can be approximated by $O(NlogN)$ and $O(\frac{N}{2^M}logN)$, respectively. Table 2 shows the time complexity of the worst, average, and best cases of the conventional and proposed mean-based serial and parallel Quick and Merge sorts for the case that data length is $N=2^{24}$ and the number of divisions is $M=1,2,3,4$.

Algorithm			Time complexity order		
			Best-case	Average-case	Worst-case
Quick	Conventional		$O(2^{28})$	$O(2^{28})$	$O(2^{48})$
	Proposed serial	$M=1$	$O(2^{25})+O(2^{28})$	$O(2^{25})+O(2^{28})$	$O(2^{25})$
		$M=2$	$O(2^{26})+O(2^{28})$	$O(2^{26})+O(2^{28})$	$O(2^{25})$
		$M=3$	$O(2^{27})+O(2^{28})$	$O(2^{27})+O(2^{28})$	$O(2^{25})$
		$M=4$	$O(2^{27})+O(2^{28})$	$O(2^{27})+O(2^{28})$	$O(2^{25})$
	Proposed parallel (per processor)	$M=1$	$O(2^{25})+O(2^{27})$	$O(2^{25})+O(2^{27})$	$O(2^{25})$
		$M=2$	$O(2^{26})+O(2^{26})$	$O(2^{26})+O(2^{26})$	$O(2^{25})$
		$M=3$	$O(2^{27})+O(2^{25})$	$O(2^{27})+O(2^{25})$	$O(2^{25})$
		$M=4$	$O(2^{27})+O(2^{24})$	$O(2^{27})+O(2^{24})$	$O(2^{25})$
Merge	Conventional		$O(2^{28})$	$O(2^{28})$	$O(2^{28})$
	Proposed serial	$M=1$	$O(2^{25})$	$O(2^{25})+O(2^{28})$	$O(2^{25})$
		$M=2$	$O(2^{25})$	$O(2^{26})+O(2^{28})$	$O(2^{25})$
		$M=3$	$O(2^{25})$	$O(2^{27})+O(2^{28})$	$O(2^{25})$
		$M=4$	$O(2^{25})$	$O(2^{27})+O(2^{28})$	$O(2^{25})$
	Proposed parallel (per processor)	$M=1$	$O(2^{25})$	$O(2^{25})+O(2^{27})$	$O(2^{25})$
		$M=2$	$O(2^{25})$	$O(2^{26})+O(2^{26})$	$O(2^{25})$
		$M=3$	$O(2^{25})$	$O(2^{27})+O(2^{25})$	$O(2^{25})$
		$M=4$	$O(2^{25})$	$O(2^{27})+O(2^{24})$	$O(2^{25})$

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 $M=1,2,3,4$).

The best-case and worst-case examined in Table 1 may occur when the data are sorted, sorted in a reverse manner, or includes similar data. In the proposed algorithm, it is proved that in these cases, the time complexity order is linear that does not violate any of the principles governing the theory of information. This method is standard in assessing the time complexity of algorithms by referring to[4,5] and the Wikipedia of each algorithm. The benefit of the proposed idea is that the time complexity of the new proposed versions of the Merge and Quick sorts for both types of data, random and non-random, decreases. It is a weakness that some other modifications have and may result in one condition better than the other.

In 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, $O(N)=O(2N)=O(3N)=...=O(9N)$. They experience the same growth rates, namely, the linear growth rate[24,25]. On the contrary, there is a remarkable difference between $O(N)$ and $O(N^2)$ or $O(N)$ and $O(NlogN)$. According to Tables 1 and 2, the complexity order of the worst and best cases of the proposed algorithms is linear much lower than that for the conventional algorithms. In the average case, we add a complexity order for evaluating the mean value and partitioning the primary array to subarrays. At the same time, the total time for sorting is much lower than the conventional algorithms because the time complexity order required to making subarrays is much lower than that for sorting the data in subarrays.

Comparison to the Sorting Algorithm Based on Parallel Random Access Machine Model

It is convenient to use the parallel random-access machine (PRAM) model for the analysis of parallel sorting algorithm. 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 wringing at the same time 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[27,28]. The possibility of parallelism between sorting processes is a significant issue for sorting of data sets. The PRAM machine model can be used to model the division of tasks with low time complexity. Marszalek, Wozniak, and others[18,19,26–28], using N_P processors for sorting a data set including N elements on CREW RAM machine, achieved a time complexity order[18,19]

$$CO_1 = O(\frac{N}{N_P}(logN)^2) \tag{9}$$

To compare the time complexity order of the sorting algorithm proposed in this paper with the algorithm presented by Marszalek and Wozniak[18,19], we assume that the number of mean-based equal-length independent subarrays, 2^M is equal to the number of processors, N_P. Also, the data is random, which means in the two-part divisions of the data there are no ordered subarrays or those with the same data. Therefore, the counter is disabled. Moreover, an approximate average is used, because its value is similar to the exact value, especially in large-scale data and consequent subarrays made by mean-based divisions. Hence, the complexity order of the proposed algorithm is equal to:

$$CO_2 = O(MN) + O(N\log\frac{N}{2^M}) \tag{10}$$

By defining the complexity reduction percentage (CRP) as a metric to compare the time complexity of the proposed algorithm to that for the parallel sorting algorithm presented by Marszalek and Wozniak[18,19] as follows:

$$CRP = (1 - \frac{CO_2}{CO_1}) \times 100 \tag{11}$$

It is given as

$$CRP = \frac{logN(logN-1) - M(2^M-1)}{(logN)^2}) \times 100 \tag{12}$$

As demonstrated in Table 3, when the length of the data increases, the superiority of the proposed sorting algorithm over the previous algorithm becomes more apparent. By increasing the number of divisions and processors, the CRP metric decreases. Even if the previous algorithm is less complex than the proposed algorithm (e.g., when we have 32 processors for a data with a length of 2^{10}), the processing time of the proposed algorithm is shorter because after consecutive mean-based divisions, we have smaller subarrays with more minor variance that can be sorted faster. Another reason is that we have fewer swaps with smaller displacements. If the number of processors is too large, the previous algorithm will be preferable in view of complexity order. 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 (or equivalently the number of divisions) that the proposed algorithm is less complex than the previous algorithm are 16, 32, 32, 32, 64, 64, 64, and 128, respectively. Generally, to have a lower time complexity to the previous algorithm, the maximum number of divisions should satisfy the following inequality

$$M(2^M - 1) \leq logN(logN - 1) \tag{13}$$

Comparison in Different Viewpoints

Table 4 shows the improvements achieved in the revised versions of the Merge and Quick sorts compared to the conventional ones in different viewpoints. In view of time complexity, 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/or 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.

Number of divisions	Number of processors	Complexity reduction percentage							
		Data length							
		2^{10}	2^{13}	2^{15}	2^{17}	2^{20}	2^{24}	2^{27}	2^{30}
1	2	89	92	93	94	95	96	96	97
2	4	84	89	91	92	94	95	95	96
3	8	69	80	84	87	90	92	93	94
4	16	30	57	67	73	80	85	88	90
5	32	-65	1	24	40	56	69	75	79

Table 3. Complexity reduction percentage (CRP) of the proposed parallel sorting algorithm to that offered in[18].

Metric	Sorting algorithm			
	Merge		Quick	
	Conventional	Proposed	Conventional	Proposed
The time complexity for 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

Table 4. Comparison in different viewpoints.

Conclusion

In This paper, we proposed an idea 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 and continues it up to the predefined number of divisions.

We showed that the mean-based pivot is more efficient than the random pivot, theoretically and numerically. Based on the numerical 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 to the conventional Quick and Merge sorts.

The proposed algorithm allows gradual extraction of sorted data. Besides, each subarray can be processed without any interaction and knowledge about 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 algorithm are the best.

- Making independent subarrays with equal lengths.
- Distinguishing the subarrays with sorted data or including similar data.
- Making the ability to sort the data gradually in a serial realization.
- Adding the feasibility to sort data in a parallel manner using independent or dependent processors.

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

- Applying the proposed idea to other sorting algorithms and prove its validity and effectiveness.
- Finding the best pivot instead of the mean value to obtain subarrays in the same length that depends on the data distribution function and its mean value and variance.
- Simulating the proposed algorithm for a large-scale data in a multi-core structure with high number of processors and finding the required processing time and number of swaps.
- Combining the proposed versions of the Merge and Quick sorts presented in this investigation to the multi-core parallel implementations reported in[18–20,26] in two cases, when the number of divisions is greater or lower than the number of processors.

References

1. Kocher, G. Agrawal, N. Analysis and review of sorting algorithms. *Int. J. Scientific Eng. and Res.* **2(3)**, 81–84 (2014).
2. Rana, Md.S., et al. MinFinder: A new approach in sorting algorithm, Presented at *the 9th Int. Congr. Inf. Commun. Technol.*, Guangxi, China (2019).
3. Al-Kharabsheh, K.S., et al. Review on sorting algorithms: A comparative study. *Int. J. Comput. Sci. Secur.* **7(3)**, 120–126 (2013).
4. Geeksforgeeks. Available online: https://www.geeksforgeeks.org (accessed on 11 01 2021).
5. Programix. Available online: https://www.programix.com (accessed on 15 01 2021).
6. Elkahlout, A.H. Maghari, A.Y.A. A comparative study of sorting algorithms: Comb, Cocktail and Counting sorting. *Int. Res. J. Eng. Technol.* **4(1)**, 1387–1390 (2017).
7. Alotaibi, A., Almutairi, A. Kurdi H. One by one (OBO): A fast sorting algorithm, Presented at *the 15th Int. Conf. Future Netw.* Commun., Leuven, Belgium (2020).
8. Cheema, S.M., Sarwar, N. Yousa, F. Contrastive analysis of bubble merge sort proposing hybrid approach. Presented at *the 6th Int. Conf. Innovative Comput. Technol.*, Dublin, Ireland (2016).
9. Hayaran, I. Khanna, P. Couple sort. Presented at *the 4th Int. Conf. Parallel, Distrib. Grid Comput.*, Waknaghat, India (2016).
10. Maus, A. A faster all parallel Mergesort algorithm for multicore processors. Presented at Norwegian Informatics Conf., Oslo, Norway (2018).

11. Shabaz, M. Kumar, A. SA sorting: A novel sorting technique for large scale data. *J. Comput. Netw. Commun.* **2019**, 1–7 (2019).

12. Xiang, W. Analysis of the time complexity of Quick sort algorithm. Presented at *the Int. Conf. Inf. Manage., Innov. Manage. Ind. Eng.*, Shenzhen, China (2011).

13. Shirvani Moghaddam, S. Shirvani Moghaddam, K. On the performance of mean-based sort for large data sets. *IEEE Access* **9**, 37418–37430 (2021).

14. Chauhan, Y. Duggal, A. Different sorting algorithms comparison based upon the time complexity. *Int. J. Res. Analytical Reviews* **7(3)**, 114–121 (2020).

15. Marszalek, Z. Parallelization of modified merge sort algorithm. *Symmetry* **9(176)**, 1–18 (2017).

16. Iqbal, S.Z., Gull, H. Muzaffar, A.W. A new friends sort algorithm. Presented at *the 2nd IEEE Int. Conf. Compu. Sci. Inf. Technol.*, Beijing, China (2019).

17. Pasetto, D. Akhriev, A. A comparative study of parallel sort algorithms. Presented at *the ACM Int. Conf. Companion on Object Oriented Programming Systems, Languages and Applications*, Portland Oregon, USA (2011).

18. Marszalek, Z., Wozniak, M., PoBap, D. Fully flexible parallel Merge sort for multicore architectures. *Complexity* **2018**, 1–19 (2018).

19. Marszalek, Z. Parallel fast sort algorithm for secure multiparty computation. *J. Universal Comput. Sci.* **24(4)**, 488–514 (2018).

20. Jimenez-Gonzalez, D., Navarro, J.J., Larriba-Pey, J.L. The effect of local sorton parallel sorting algorithms. Presented at *the 10th EuroMicro Workshop on Parallel, Distrib. and Network-based Process.*, Canary Islands, Spain (2002).

21. Alnihoud, J. Mansi, R. An enhancement of major sorting algorithms. *The Int. Arab J. Inf. Technol.* **7(1)**, 55–61 (2020).

22. Huang, X., Liu, Z. Li, J. Array sort: An adaptive sorting algorithm on multi-thread. *IET J. Eng.* **2019(5)**, 3455–3459 (2019).

23. Prifti, V., Bala, R., Tafa, R., Saatciu, D. Fajzaj, J. The time profit obtained by parallelization of Quicksort algorithm used for numerical sorting. Presented at *the Sci. and Inf. Conf.*, London, UK (2015).

24. Chee, Y. A real elementary approach to the master recurrence and generalizations. Presented at *the 8th Annual Conf. Theory Applications of Models of Computation*, Tokyo, Japan (2011).

25. Bentley, J.L., Haken, D. Saxe, J.B. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News***12(3)**, 36–44 (1980).

26. Wozniak, M., Marszalek, Z., Gabryel, M., Nowicki, R. Preprocessing large data sets by the use of Quick sort algorithm.*Advances in Intelligent Systems and Computing* **364**, 111-–121 (2015).

27. Wozniak, M., Marszalek, Z., Gabryel, M. Nowicki, R. Triple Heap sort algorithm for large data sets. Presented at *In Looking into the Future of Creativity and Decision Support Systems*, Cracow, Poland (2015).

28. Marszalek, Z., Wozniak, M., Borowik, G., Wazirali, R., Napoli, C., Pappalardo, G., Tramontana, E. Benchmark tests on improved Merge for big data processing. Presented at *the Asia-Pacific Conference on Computer Aided System Eng.* Quito, Ecuador (2015).

Legends of Figures and Tables

Figure 1. The performance of the proposed and conventional Quick-sort, (**a**) Average running time, (**b**) Average number of swaps.

Figure 2. The performance of the proposed and conventional Quick-sort, (a) Average running time, (b) Average number of swaps.

Figure 3. NDSI vs. length of non-integer uniform data in the mean-based and random pivot scenarios, (a) Low (10), (b) High (1000) standard deviations.

Figure 4. NDSI vs. length of integer uniform data in the mean-based and random pivot scenarios, (a) Low (10), (b) High (1000) standard deviations.

Figure 5. NDSI vs. length of non-integer Gaussian data in the mean-based and random pivot scenarios, (a) Low (10), (b) High (1000) standard deviations.

Figure 6. NDSI vs. length of integer Gaussian data in the mean-based and random pivot scenarios, (a) Low (10), (b) High (1000) standard deviations.

Figure 7. The performance of the proposed parallel and conventional Merge-sort, (a) Average running time, (b) Maximum number of swaps per processor.
Figure 8. The performance of the proposed and conventional parallel Quick-sort, (a) Average running time, (b) Maximum number of swaps per processor.
Table 1. The time complexity of the conventional and proposed Quick and Merge sorts in the best, average, and worst cases.
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).
Table 3. Complexity reduction percentage (CRP) of the proposed parallel sorting algorithm to that offered in18.
Table 4. Comparison in different viewpoints.

Acknowledgements

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

Author contributions

Conceptualization, S.S.M. and K.S.M.; methodology, S.S.M. and K.S.M.; software, K.S.M.; validation, S.S.M. and K.S.M.; formal analysis, S.S.M.; investigation, S.S.M. and K.S.M.; resources, S.S.M. and K.S.M.; data curation, K.S.M.; writing, review, and editing, S.S.M. and K.S.M.; supervision, S.S.M.; project administration, S.S.M.

Competing Interests

The authors declare no competing interests.

Figures

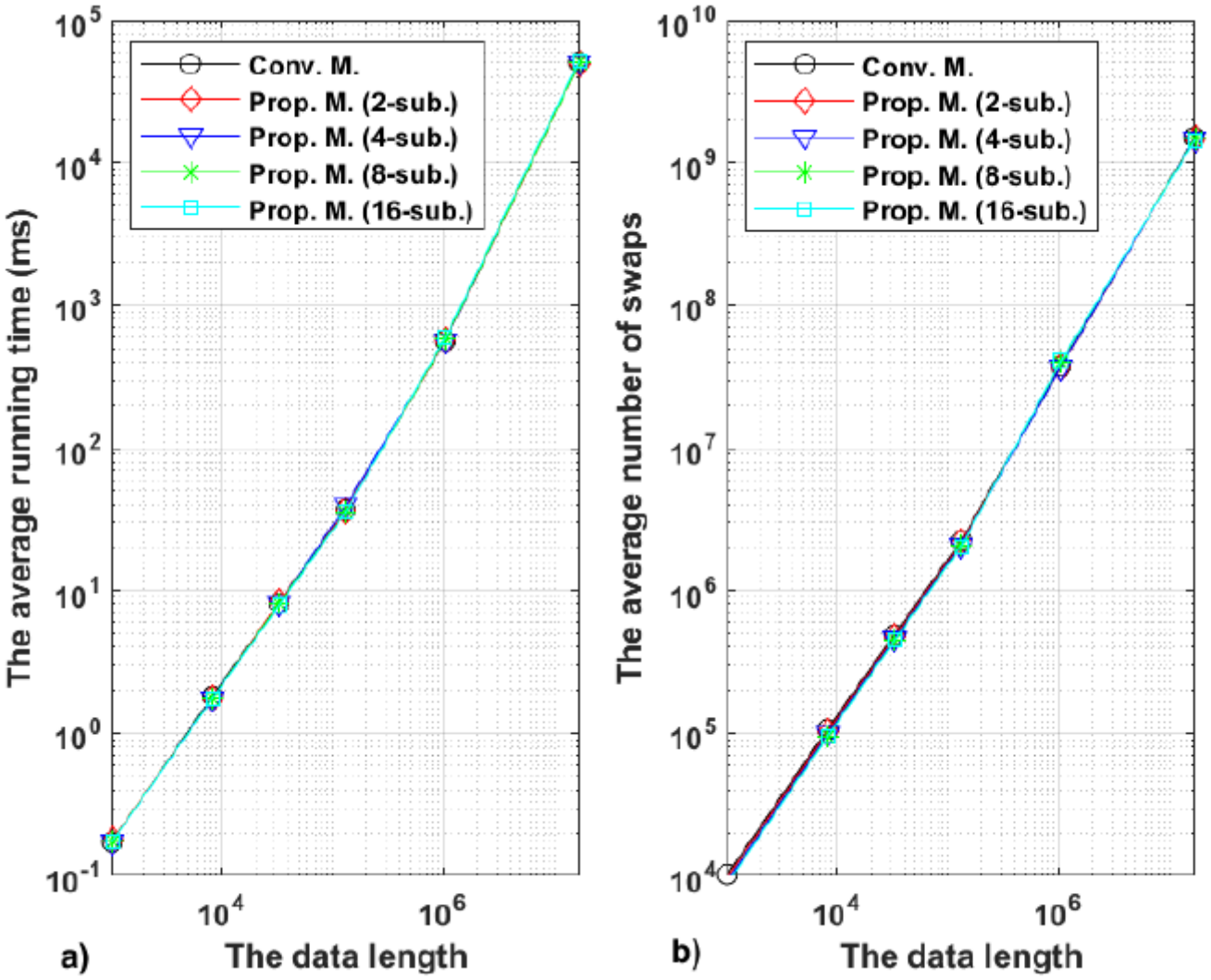

Figure 1

The performance of the proposed and conventional Merge-sort, (a) Average running time, (b) Average number of swaps.

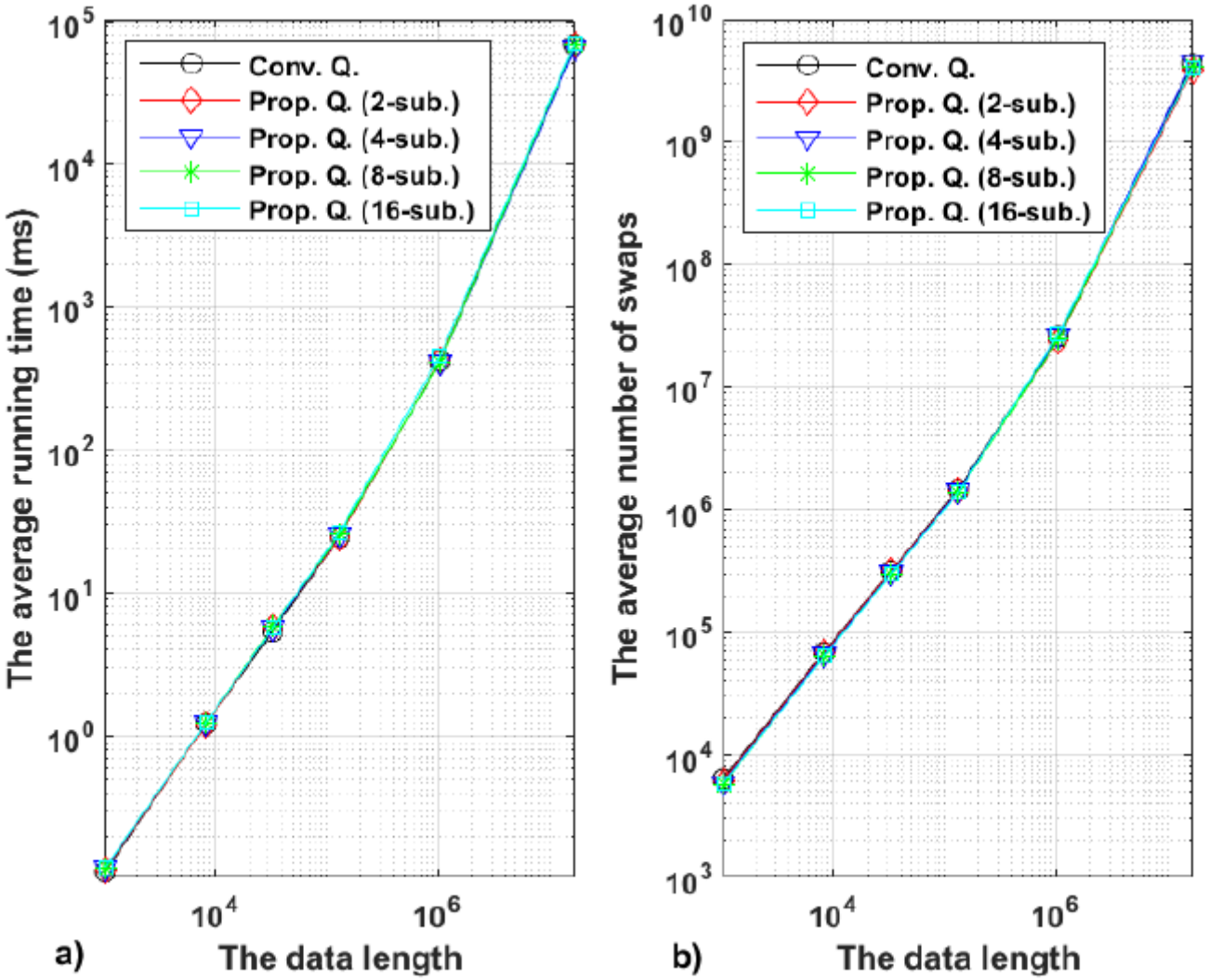

Figure 2

The performance of the proposed and conventional Quick-sort, (a) Average running time, (b) Average number of swaps.

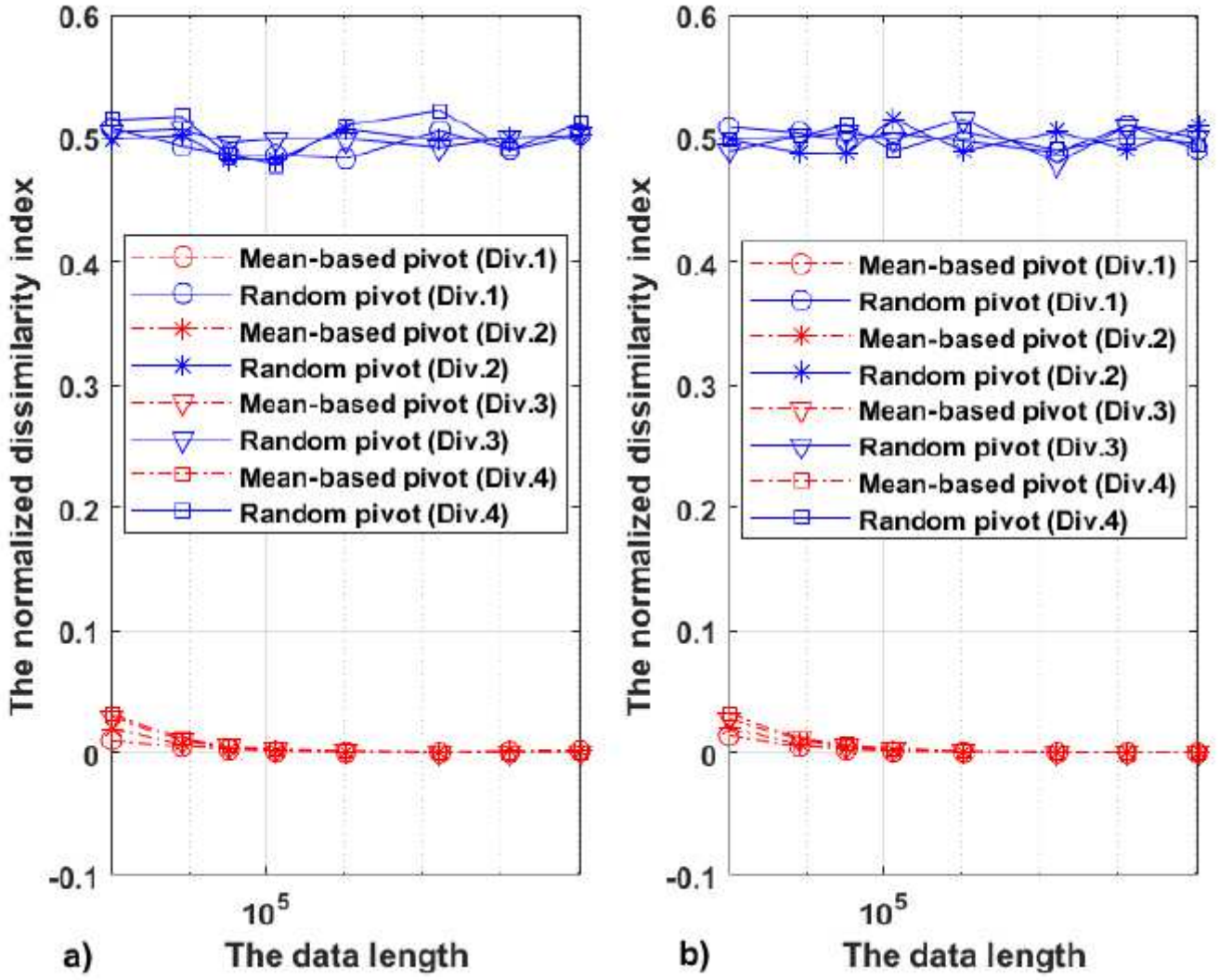

Figure 3

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

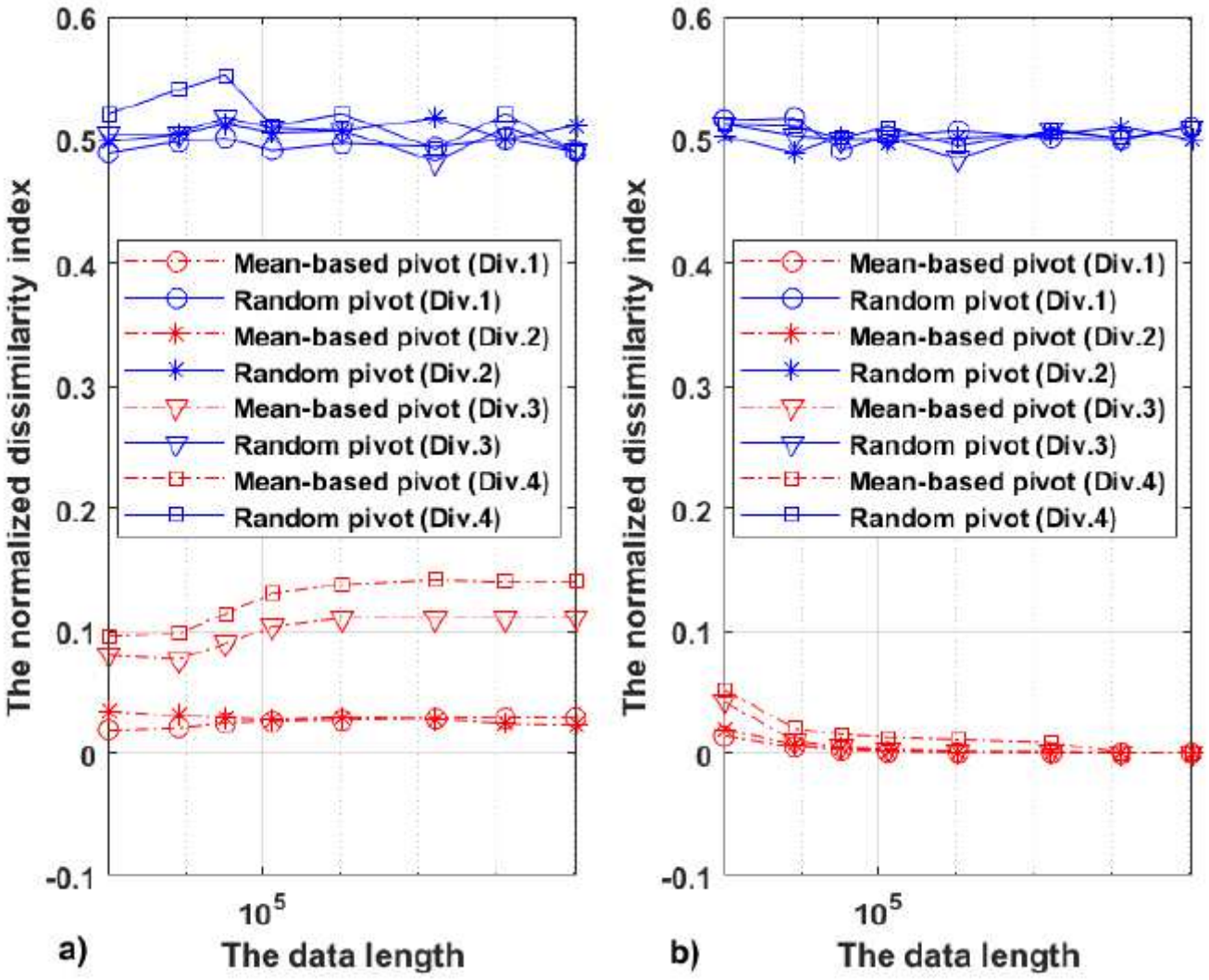

Figure 4

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

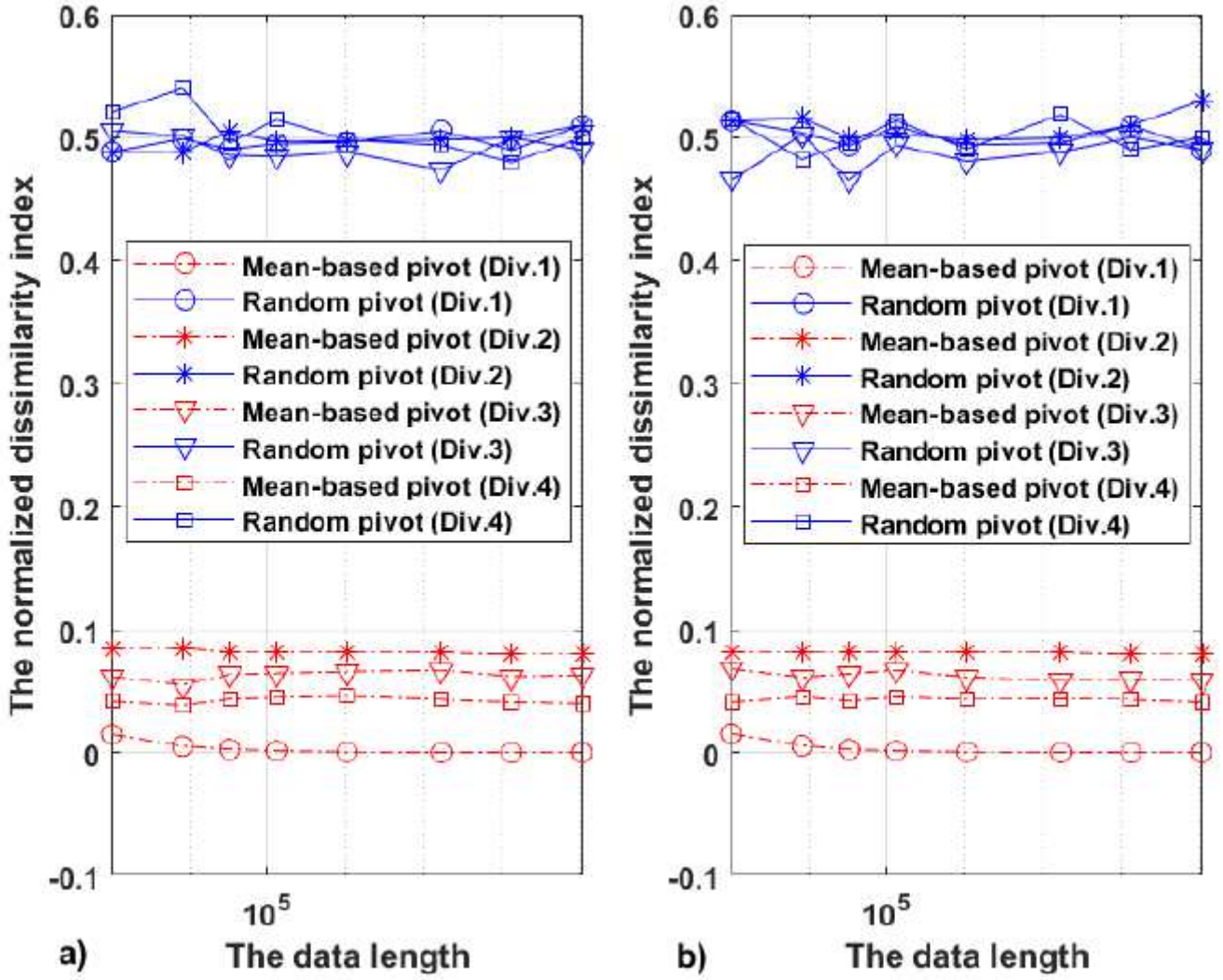

Figure 5

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

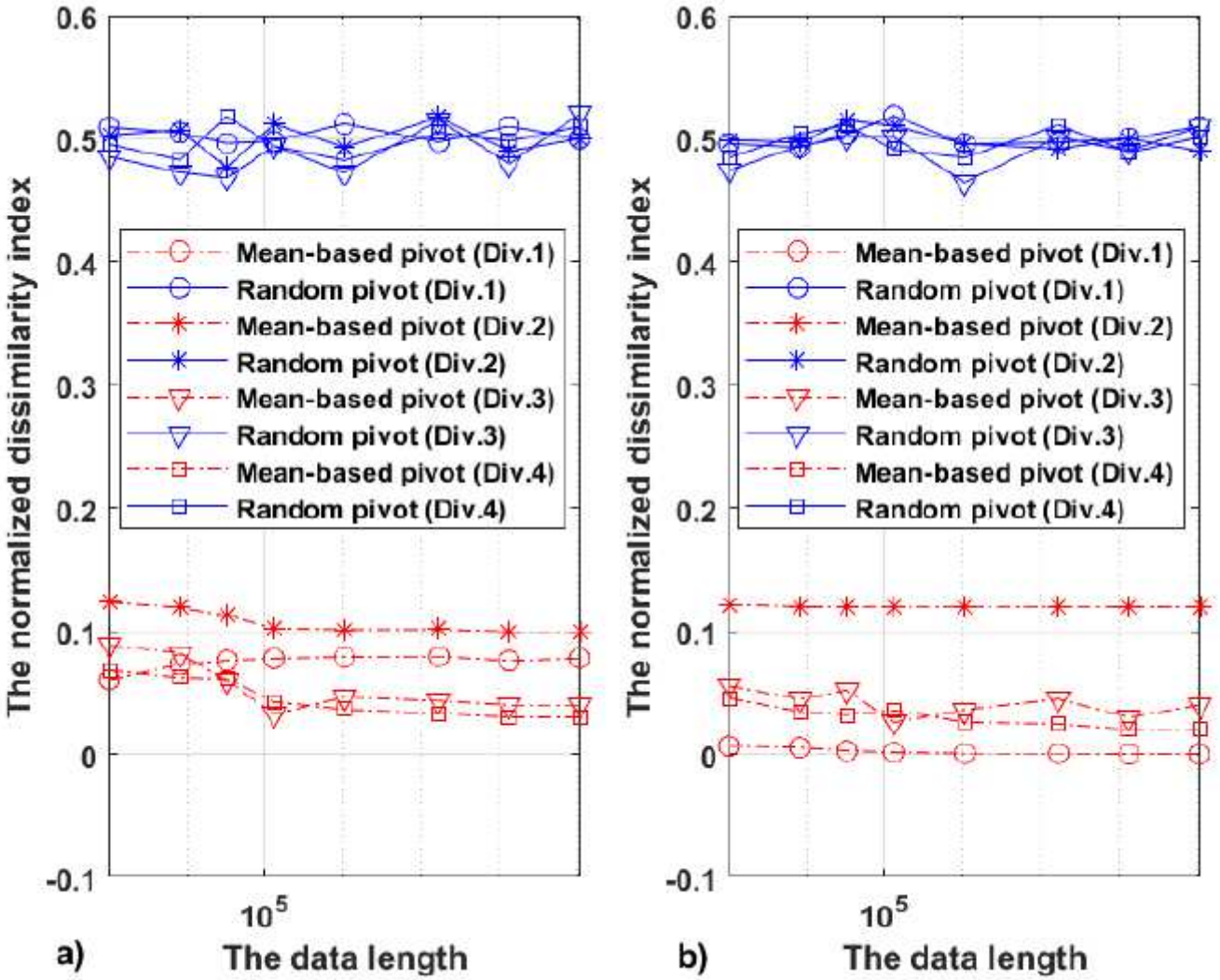

Figure 6

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

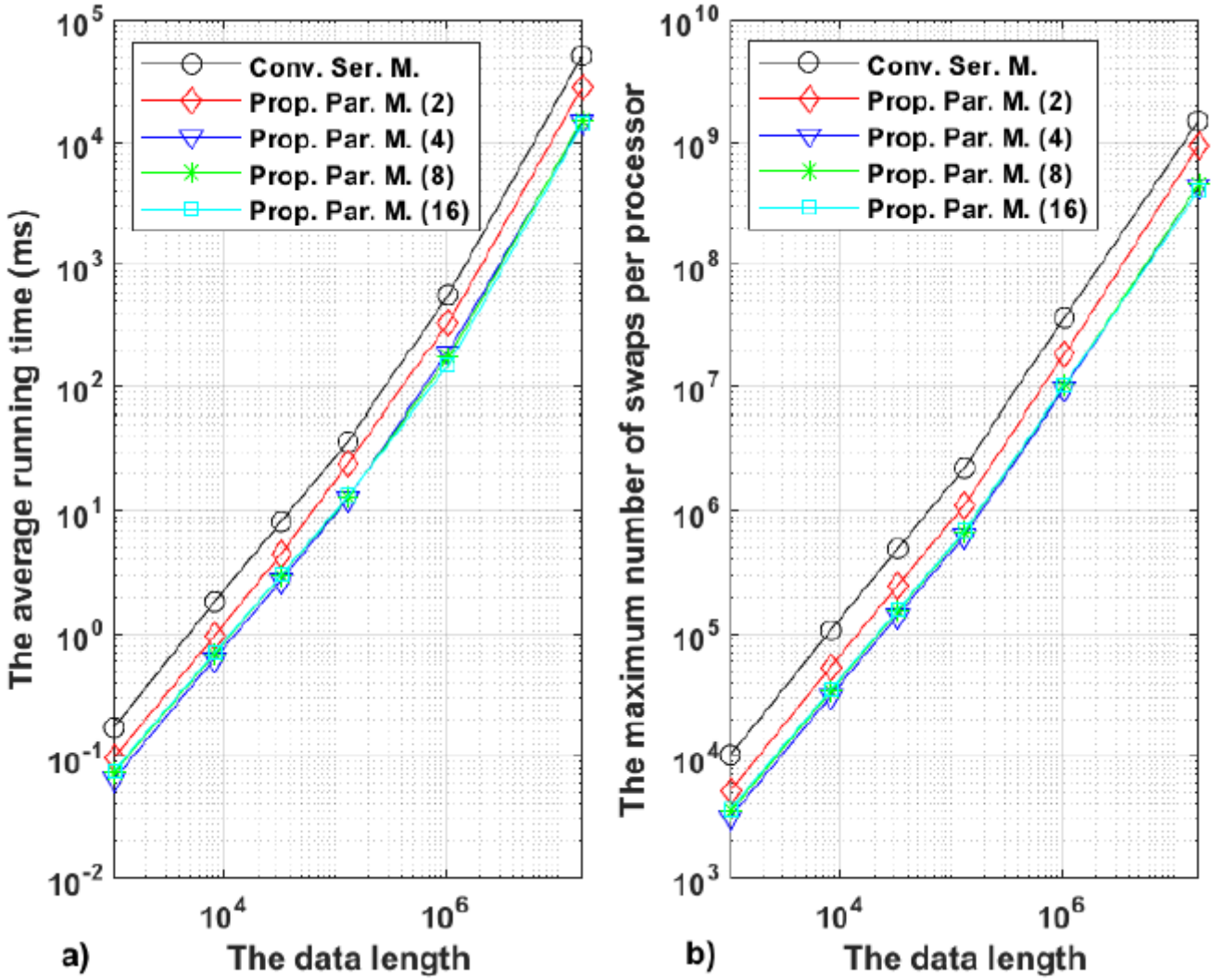

Figure 7

The performance of the proposed parallel and conventional Merge-sort, (a) Average running time, (b) Maximum number of swaps per processor.

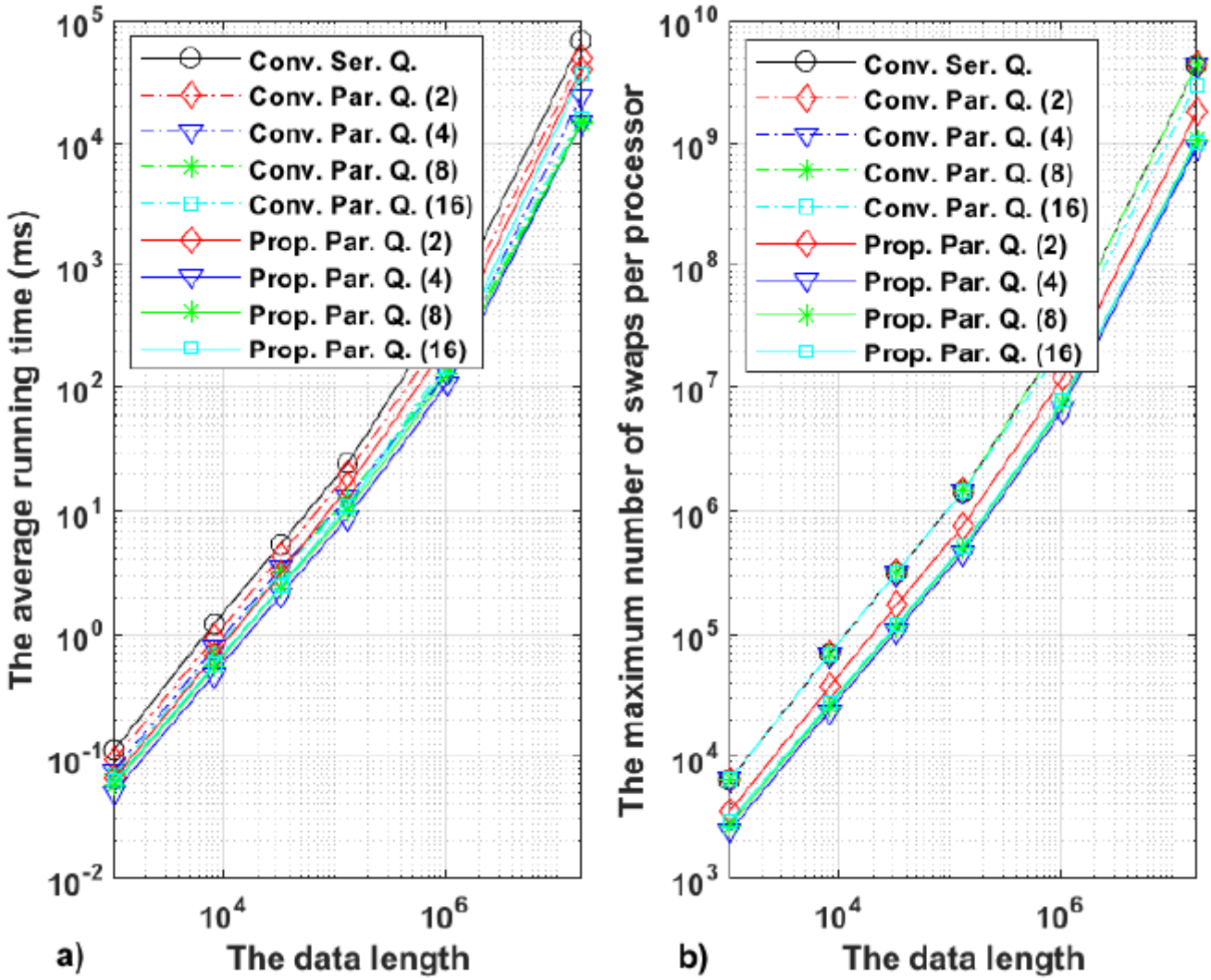

Figure 8

The performance of the proposed and conventional parallel Quick-sort, (a) Average running time, (b) Maximum number of swaps per processor.