Hindawi
Complexity
Volume 2018, Article ID 8679579, 19 pages
https://doi.org/10.1155/2018/8679579

Research Article

Fully Flexible Parallel Merge Sort for Multicore Architectures

Zbigniew Marszałek, Marcin Woźniak, and Dawid Połap

Institute of Mathematics, Silesian University of Technology, Kaszubska 23, 44-100 Gliwice, Poland

Correspondence should be addressed to Marcin Woźniak; marcin.wozniak@polsl.pl

Received 10 June 2018; Revised 30 August 2018; Accepted 23 September 2018; Published 2 December 2018

Guest Editor: Julian Szymanski

The development in multicore architectures gives a new line of processors that can flexibly distribute tasks between their logical cores. These need flexible models of efficient algorithms, both fast and stable. A new line of efficient sorting algorithms can support these systems to efficiently use all available resources. Processes and calculations shall be flexibly distributed between cores to make the performance as high as possible. In this article we present a fully flexible sorting method designed for parallel processing. The idea we describe in this article is based on modified merge sort, which in parallel form is designed for multicore architectures. The novelty of this idea is in particular way of processing. We have developed a fully flexible method that can be implemented for a number of processors. The tasks are flexibly distributed between logical cores to increase the efficiency of sorting. The method preserves separation of concerns; therefore, each of the processors works separately without any cross actions and interruptions. The proposed method was described in theoretical way, examined in tests, and compared to other methods. The results confirm high efficiency and show that with each newly added processor sorting becomes faster and more efficient.

1. Introduction

Multicore architectures support a number of coworking logical processors that receive tasks distributed for parallel processing. Modern computing needs procedures that use advanced programming techniques oriented on performance, which can be achieved by parallelization of algorithms. The algorithms must be implemented in a way that preserves an appropriate separation of concerns which helps to avoid cross actions and interferences between processors. These aspects are important for the efficiency of data base systems and information processing, where a flexible usage of several processors improves performance. Bochenina et al. [1] presented a flexible method designed for parallelization of processing in simulation of stochastic Kronecker graphs. Czarnul et al. [2] proposed special environments for testing parallel applications on large distributed systems. A framework for distributed systems in health care monitoring was proposed by Mora et al. [3]. Esmaeili et al. [4] designed multiagent based algorithm and Stanescu et al. [5] described data base management for distributed systems. Sorting algorithms are used in all types of systems; therefore, improvements in these methods will benefit in many aspects. Research on sorting methods covers many problems from storage to sorting itself. Depending on the idea we can find theoretical or practical advances that directly improve data management, speed of sorting, communication over hardware, storage, etc. Janetschek et al. [6] described how multicore architectures improve runtime environments. The results show that devoted algorithms when run in parallel may significantly improve efficiency. Parallelization of processes and devoted methods are very important for data management, especially for systems where we do not use frames that keep the order of information. De Farias et al. [7] presented devoted method for NoSQL systems based on regression. González-Aparicio et al. [8] described tests on NoSQL key-value data bases. We can see that methods, especially sorting, when parallelized in efficient way are very beneficial for data systems.

Classic versions of various sorting algorithms were presented by Aho and Hopcroft [9] and Knuth [10] among which three have main impact on the development in information processing: quick sort, heap sort, and merge sort. These methods are constantly improved to operate on modern architectures in the most efficient way.

Quick sort was improved by prevention of deadlocks and construction of the new pivot method. In the papers

of Bing-Chao and Knuth [11] faster exchange mechanism with the new pivot was presented. Francis and Pannan [12] discussed how to change partitioning of sorted strings by dynamic assignments, while Rauh and Arce [13] presented an improvement by application of median value for partitioning. Tsigas and Zhang [14] proposed an implementation oriented on efficiency for Sun Microsystems, Inc. The research on improvements for pivot procedure results in the new mechanisms of changing elements in the output string. In the article of Daoud et al. [15] an interesting mechanism for nonquadratic method was proposed, and in the work of Edmondson [16] the research on various pivot possibilities was discussed, while in the paper by Kushagra et al. [17] a multipivot procedure was proposed.

Heap sort benefited from new propositions of multilevel structures to store the data and improvements to the algorithms implemented for changing elements in this structure. Ben-Or [18] presented mathematical assumptions for modeling relations between elements in levels of the heap. Efficiency of reorganizing this structure was presented by Doberkat [19] and Wegner and Teuhola [20], while additional possibilities to boost the method by improved swap methods were proposed by Sumathi et al. [21]. Heap sort was also examined on various computing architectures (Roura [22]) with some possibilities for parallelization (Abrahamson et al. [23]).

Merge sort was improved by new ideas of sublinear methods and various approaches to parallelization. Carlsson et al. [24] discussed a sublinear procedure that was used for composition of sorted substrings. Theoretical background for the first approach to parallel version was presented by Cole [25]. An idea to compose devoted mechanism for partially sorted strings was discussed by Gediga and Düntsch [26]. Results from tests for new implementations were presented by Harris [27] and memory usage on various architectures was proposed by Salzberg [28] and Huang and Langston [29]. Zheng and Larson [30] discussed how to improve input-output operations for faster merging. In the paper by Zhang and Larson [31] dynamic memory assignments were proposed for various capacity of computing architectures. Buffering and reading from input strings were presented by Zhang and Larson [32]. Merge sort was also evaluated in various tests and benchmarks by Vignesh and Pradhan [33], Cheema et al. [34], and Paira et al. [35]. These research results show that merge sort has a very high potential for new improvements.

Quick sort, heap sort, and merge sort were also mixed and derived to present new methods of sorting. Speed of sorting that increased by virtual memory assignments was discussed by Alanko et al. [36] and Larson and Graefe [37]. Cash usage was examined by LaMarca and Ladner [38]. Skewed strings and other input types were examined by Crescenzi et al. [39], while derivatives composed to adapt to the input were presented by Estivill-Castro and Wood [40]. Self-sorting by adaptation of Markov idea for chain rules was proposed by Axtmann et al. [41], while Abdel-Hafeez and Gordon-Ross [42] proposed a free sorting approach. Mohammed et al. [43] proposed an insertion of the elements into the output string by application of bidirectional method of sorting.

1.1. Related Works. During research on improved and new methods of sorting we have proposed improvements to classic versions. Dynamic division of length for quick sort was proposed by Woźniak et al. [44]; this gave an improvement in sorting, prevented deadlocks, and sped up the quick sort of about 10%. Heap structure for faster processing of large data sets was discussed by Woźniak et al. [45]. We have described how to compose and search the structure of the heap in a way that speeds up sorting of about 5% to 10%. The research on improvements for merge sort gave new, faster processing of input string but also improved management of the data during iterations in the algorithm. In [46] we have shown that dynamic rule of merging speeds up the process of about 10%. In [47] we have proved that nonrecursive sorting makes the merge method flexible to various architectures, and in [48] our ideas were reported for Hadoop systems. A theoretical introduction to parallelization of sorting was presented by Cole [25]. The results given there have shown the way to divide the tasks between processors using the idea of binary trees. Uyar [49] presented an approach to first parallelization of merge sort, after which a new method was proposed by Marszałek [50] and Marszałek [51].

The algorithm we would like to discuss here assumes a new concept of sorting by the use of independent merge. Presented in this article is a method that makes the algorithm sorting in a fully flexible way, which means that with each new processor the algorithm gains additional capacity of sorting. The new method we present here is implemented in the way that preserves separate concerns executed on each logical processor. This makes the novelty of one of the most important aspects for this algorithm. The model for this method is using concept of allocation of strings in memory block for the currently merging processor. The idea is based on the PRAM model. The model of PRAM machine is a theoretical assumption of parallel processing; therefore, it does not consider hardware aspects like delay in communication and information exchange between registers in the system. The proposed approach is moving away from the dynamic splitter strings. For the independence of the input and flexible execution of efficient sort, a certain presorted string approach is proposed. This approach allows us to estimate the running time of the algorithm regardless of the input data with a certain processing model. The sorting tasks are divided between processors so that each of them is sorting without any interruptions or cross actions. In benchmark tests the algorithm was about 15% more efficient in comparison to other sorting methods. The proposed parallel sorting has theoretical time complexity $O((\log_2 n)^2)$.

The proposed model of independent and fully flexible sorting was implemented in C# MS Visual 2015 on MS Windows Server 2012. For the research we have used Opteron AMD Processor 8356 8p. The algorithm has been described in theoretical analysis and examined in practical benchmark tests. The results we present in this article show that this algorithm is fast and gives very good improvement when run on multicore architectures.

The concept of faster sorting is one of the main topics for recent advances in architectures and big data processing for complex Internet of things systems. Modern computing

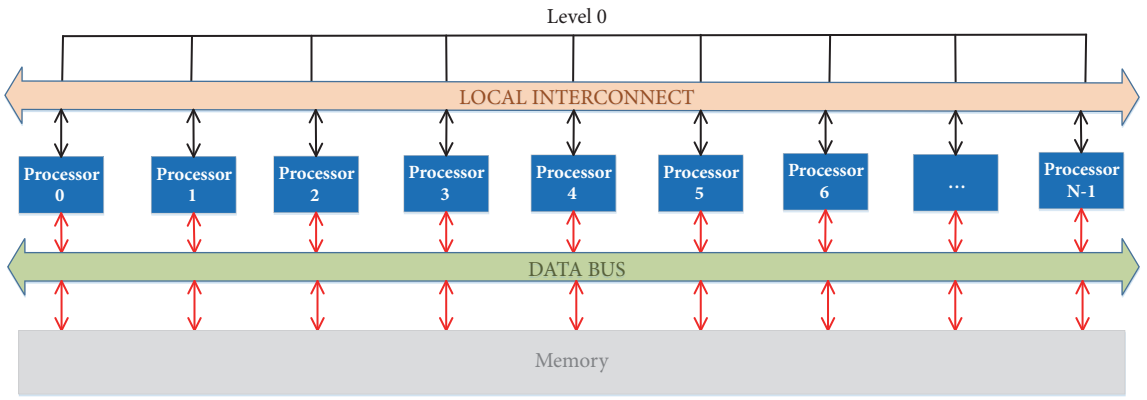

Figure 1: Parallel Random Access Machine.

requires more and more information; therefore, systems devoted to faster processing will give an impact on the final efficiency in data mining easing work for the users. Our solution fits this concept both for theoretical and practical aspects.

2. Data Processing in NoSQL Database and Parallel Sorting Algorithms

A significant problem in theory of computational complexity is the acceleration time of sorting algorithms. Mainly it is improved by dividing the input into parts and executing all tasks concurrently by multiple processors. The new approach we discuss here runs into the limits to show a fully flexible method of this concept. What to do when we can no longer divide inputs into smaller portions, and we would like to make the sorting faster by using multiple processors? A solution to this example can be an independent choice of the operating processor. In this article we present how to implement parallel sorting method with time complexity $O((\log_2 n)^2)$.

The architecture of modern processors allows performing in a parallel way multiple processes. For the analysis of algorithms run on modern computers we use the theoretical multiprocessor machine model PRAM (Parallel Random Access Machine) presented in Figure 1. The efficiency of the algorithm depends on its time complexity and memory usage. The abstract model of the PRAM machine is used to analyze parallel algorithms in terms of complexity. PRAM is defined as a system $< P, I, M >$ composed of processors P_i for $i = 0, \cdots N-1$, instructions I for each of them and memory M accessed during these operations. Due to the access of the processors to read from and write into memory we can talk about four PRAM machines:

(i) Concurrent Read Concurrent Write (CRCW)

(ii) Exclusive Read Concurrent Write (ERCW)

(iii) Concurrent Read Exclusive Write (CREW)

(iv) Exclusive Read Exclusive Write (EREW)

The first two types of machine PRAM are typically theoretical models of computing. The third type allows to access memory using any processor but only one processor can read from the memory and write into the memory cell at the same time. If two processors are simultaneously writing to the same memory cell, the result of the operation is undefined. For the evaluation of our idea we used a type that allows to read and to write on a single processor. Presented in this work was an algorithm developed and implemented using PRAM model to show its theoretical aspects, but the efficiency was verified in practical benchmark tests and compared to other methods.

Definition 1. The time complexity of the algorithm run on the PRAM machine can be defined as the time (number of instructions) of the longest procuring processor during the execution of the algorithm.

Definition 2. The memory complexity is the number of cells used by the PRAM machine to perform the algorithm.

Definition 3. The processor complexity is the maximum number of active processors during computations.

For computational analysis of sorting algorithms, which use comparisons of sorted sequence elements, a simplified calculation model is appropriate. We define the number of comparisons made by the algorithm during sorting and on this basis we determine the complexity of time, processor, and memory for sorting methods on the PRAM machine. The presented calculation model adequately determines the operation time of the sorting algorithm on the computer which performs the entire sorting in the operating memory. For the analysis of external sorting algorithms, other calculation models are used, e.g., No-Remote Memory Access (NoRMA) or Uniform Memory Access (UMA).

The Fully Flexible Parallel Merge Sort method describes how to split the sorting processes between independently working processors. A general scheme of task division between processors is presented in Algorithm 1 and in a flow chart shown in Figure 2. Each merged pair of strings is allocated in memory block in which the currently merging processor can read from and write to memory locations. The algorithm constructed in a classic way has the time complexity $O(n)$. We propose a new way to merge pairs of

1: **for** $t = 0$ **to** $\lceil \log_4 n \rceil$ **do**
2: For current t processor, determine the starting number of the string a_i to locate element,
3: Find the boundary indexes of the string to insert a_i,
4: Find the index of the next element before which the a_i element is inserted,
5: For the computed index insert a_i element into array b,
6: Wait for all processors,
7: For current t processor, determine the starting number of the string b_i to locate element,
8: Find the boundary indexes of the string to insert b_i,
9: Find the index of the next element before which the b_i element is inserted,
10: For the computed index insert b_i element into array a,
11: Wait for all processors,
12: **end for**

Algorithm 1: The algorithm to divide tasks between processors P_i $i = 0, , n$.

sorted strings of $n/2$ elements by n independently working processors. The novelty of our method is in flexibility of implementation. We present a solution using the principle that each processors can read from the memory cells but only one can write to a cell that no other processor is using. Therefore the proposed method has a very high efficiency without cross actions, what results in higher efficiency with each additional processor. The proposed algorithm of combining two sequences using n independently working processors has time complexity $O((\log_2 n)^2)$. This result was proved in theory and examined in benchmark tests, what we discuss in the following sections of the article.

3. Efficient Parallel Sorting

New computers make it possible to implement methods that parallelize processing. Main requirements for these methods are low computational complexity and flexibility to adjust to various multicore architectures. Among possible applications of these methods we can define data base systems and information management systems. This is due to the fact that new machines with a number of processors can divide the tasks between logical processors and therefore the requests are answered in a shorter time. The efficiency of this stage depends on implemented method, which must be flexible for any number of processors and preserve the sorting algorithm without any unnecessary cross actions or interruptions between processors.

3.1. Fully Flexible Parallel Merge Sort Algorithm. A well-known classic merge sort algorithm to merge two sorted strings of n elements allows to merge them into a single sorted sequence by doing so at maximum $2n-1$ comparisons and not less than n comparisons. This algorithm is difficult to perform independently on several processors. We can of course try to parallelize merge by the use of two independently working processors; however, this idea is not efficient. Here we present a novel approach to efficiently merge in parallel way so that the processors do not interfere with each other and the method is flexible for various numbers of processors. We use the possibilities of modern computers and the fact that all processors can simultaneously read from memory cell, but at the same time only one can write into the memory.

Now suppose that we have two sorted strings to merge $n/2$ elements in array $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$. The first half of the original string is stored in $a_0 \leq a_1 \leq \ldots \leq a_{n/2-1}$ and the second half of the original string is stored in $a_{n/2} \leq \ldots \leq a_{n-1}$. The algorithm inserts an element of the array a into the array $b = [b_0, b_1, \ldots, b_{n-1}]$ using processors with indexes $i, 0 \leq i < n$ in simple steps. Each of processors in the loop has a unique index and transmits the information about the dimension of the merged string.

Processor number $i, 0 \leq i < n/2$ computes the index of the element a_t on the second half of the original string before which the element should be inserted to have $a_i, a_{t-1} < a_i \leq a_t$. In case where the insertion must be done after the last element of the array a, the value of the index is n. Processor i inserts the value of the element a_i in the array b under the index $i + t - n/2$. Imagine, for instance, a way to merge two strings stored in the array $a = [2, 5, 7, 9, 0, 2, 4, 8]$ using processors P_0, P_1, P_2, P_3 which insert elements into the array b. The situation is shown in Figure 3.

Each of the processors P_i, $i = 0, 1, 2, 3$ operates independently and determines the index t_i of the element in the second half of the original string, before which the element should be inserted. For example, the processor P_0 inserts the element $a_0 = 2$ prior to $a_5 = 2$. Hence, the index is calculated and inserted element 2 into the array b is equal to the sum of the indexes of elements $a_1 = 2$ and $a_5 = 2$ minus 4 and is 1, see Figure 3. Processor number $i, n/2 \leq i < n$ computes the index of the element a_t on the first half of the original string before which the element should be inserted $a_i, a_{t-1} \leq a_i < a_t$. In case where the insertion must be done after the last element of the array y, the value of the index is $n/2$. Processor i performs insertion of the value of the element a_i into the output string b under the index $i + t - n/2$. Imagine, for instance, a way to merge two strings recorded in array $a = [2, 5, 7, 9, 0, 2, 4, 8]$ using processors P_4, P_5, P_6, P_7 which insert elements into the array b. The situation is shown in Figure 4.

Each of the processors P_i, $i = 4, 5, 6, 7$ operates independently and determines the index t_i of the element in the first half of the array, before which the element should be

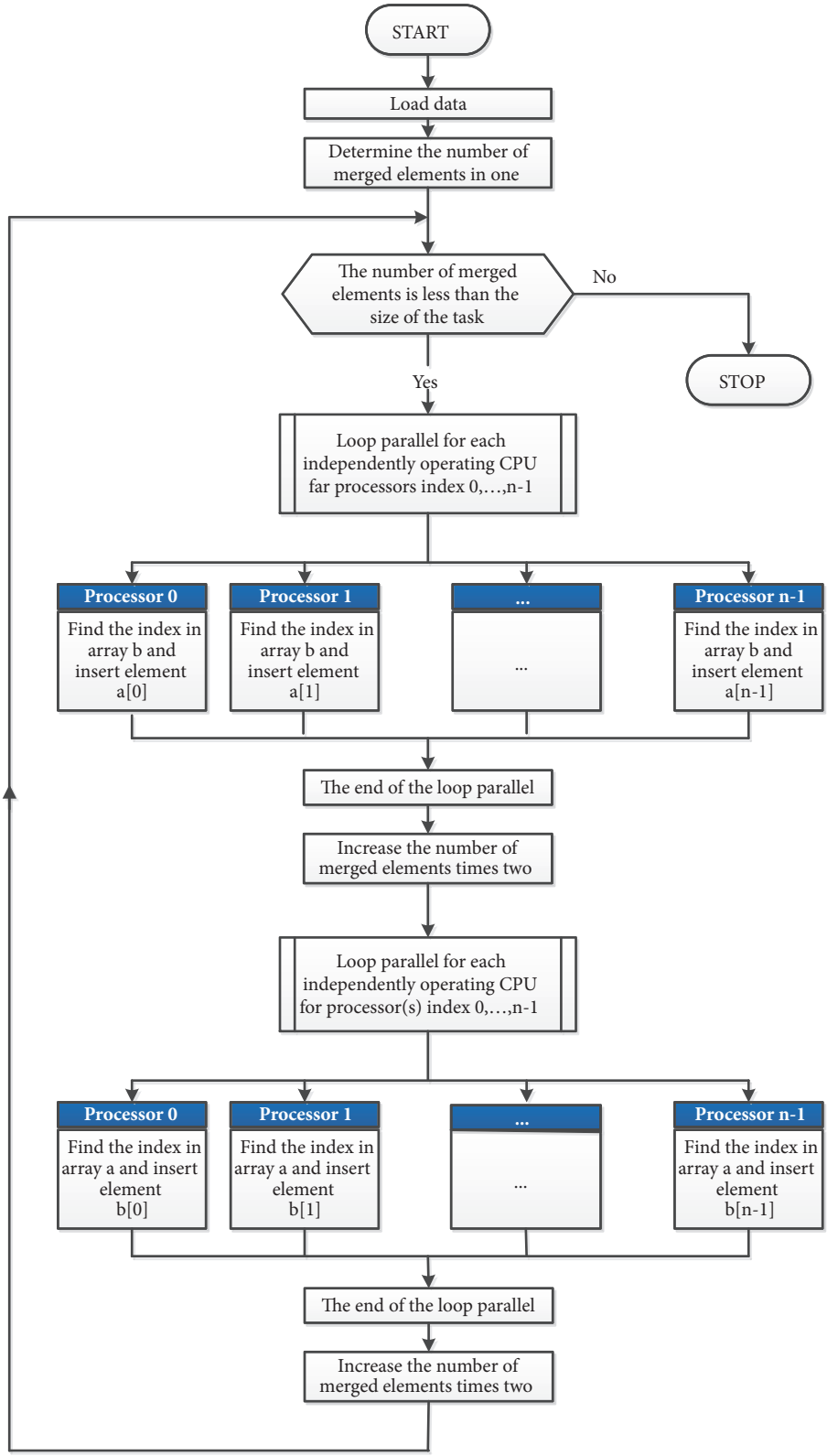

Figure 2: The block diagram of iterative task assignment to the processors while implementing proposed flexible algorithm that adapts to the number of used processors.

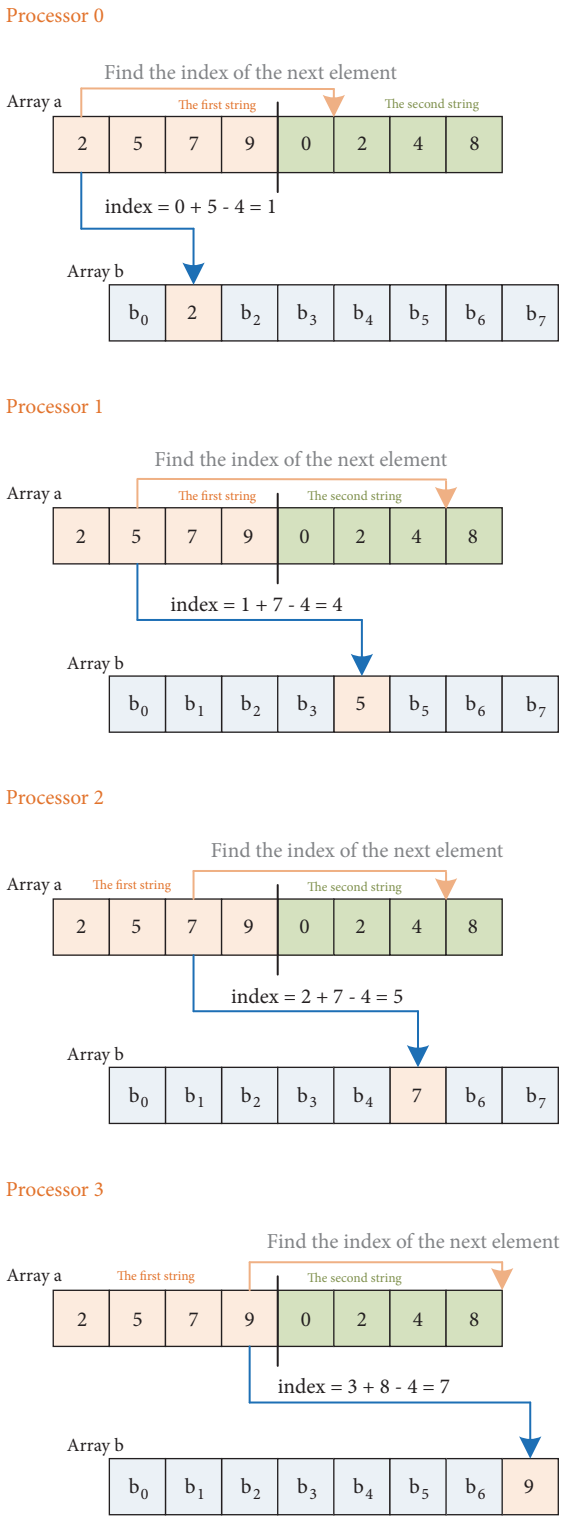

FIGURE 3: Model of the applied flexible sorting for the number of processors for the insertion of elements into the array b: processor P_0 sorts element a_0 into the array b, processor P_1 sorts element a_1 into the array b, processor P_2 sorts element a_2 into the array b, etc.

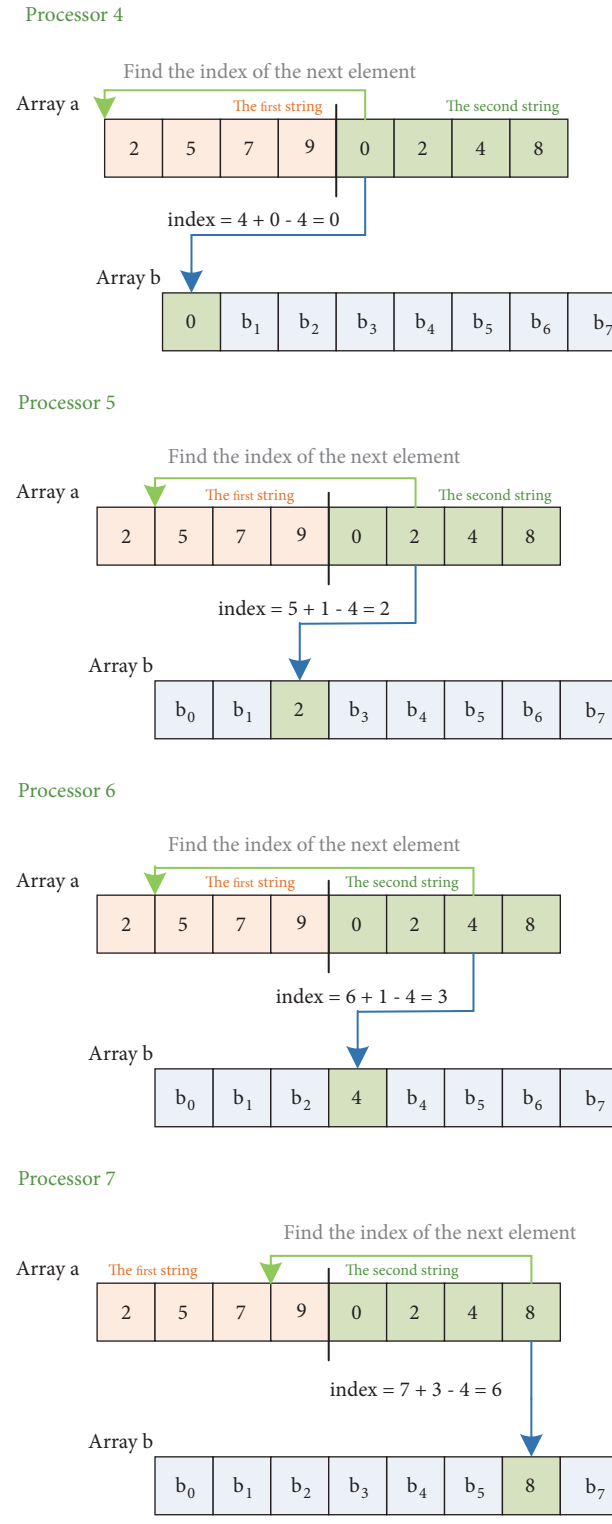

FIGURE 4: Model of the applied flexible sorting for the number of processors for the insertion of elements into the array b: processor P_4 sorts element a_4 into the array b, processor P_5 sorts element a_5 into the array b, processor P_6 sorts element a_6 into the array b, etc.

```
1: The dimension of the array a is n,
2: Create an array b of dimension n,
3: Set options for parallelism to use all processors of the system,
4: t ⟵ 1,
5: while t < n do
6:    m2 ⟵ 2 ∗ t,
7:    m4 ⟵ 4 ∗ t,
8:    parallel for i1 ⟵ 0 to n-1 do
9:   j ⟵ i1/m2,
10:  i ⟵ m2 ∗ j,
11:  iw ⟵ i1%m2,
12:  p1 ⟵ i + t,
13:  if p1 > n then
14:     p1 ⟵ n,
15:  end if
16:  p2 ⟵ i + m2,
17:  if p2 > n then
18:     p2 ⟵ n,
19:  end if
20:  if i1 < p1 then
21:     iz ⟵ IndexRight(a, p1, p2, a[i1]),
22:     b[iz + i + iw] ⟵ a[i1],
23:  else
24:     iz ⟵ IndexLeft(a, i, p1, a[i1]),
25:     b[iz + i1 − t] ⟵ a[i1],
26:  end if
27:  end parallel for
28:  parallel for i1 ⟵ 0 to n-1 do
29:  j ⟵ i1/m4,
30:  i ⟵ m4 ∗ j,
31:  iw ⟵ i1%m4,
32:  p2 ⟵ i + m2,
33:  if p2 > n then
34:     p2 ⟵ n,
35:  end if
36:  p3 ⟵ i + m4,
37:  if p3 > n then
38:     p3 ⟵ n,
39:  end if
40:  if i1 < p2 then
41:     iz ⟵ IndexRight(b, p2, p3, b[i1]),
42:     a[iz + i + iw] ⟵ b[i1],
43:  else
44:     iz ⟵ IndexLeft(b, i, p2, b[i1]),
45:     a[iz + i1 − m2] ⟵ b[i1],
46:  end if
47:  end parallel for
48:  t ⟵ 4 ∗ t,
49: end while
```

Algorithm 2: Proposed flexible parallel merge.

inserted. For example, processor P_4 inserts the element $a_4 = 0$ prior to $a_0 = 2$. Hence, the index is calculated and inserted element 0 into the array b is equal to the sum of the indexes of elements $a_4 = 0$ and $a_0 = 2$ minus 4 and is 0, see Figure 4. A sorting algorithm which can flexibly involve each additional processor is presented in Algorithm 2. Algorithm for finding the index of the element before inserting the new element has time complexity $O(\log_2 n)$.

Because in our method the processors operate independently, we use the machine CREW PRAM model with n processors. The most commonly used model for analysis of parallel algorithms is machine PRAM. In practice, we distinguish between three variants of this model. The first model the Exclusive Read Exclusive Write PRAM gives you the ability to read from and write into memory by only one processor. The second model the Concurrent Read

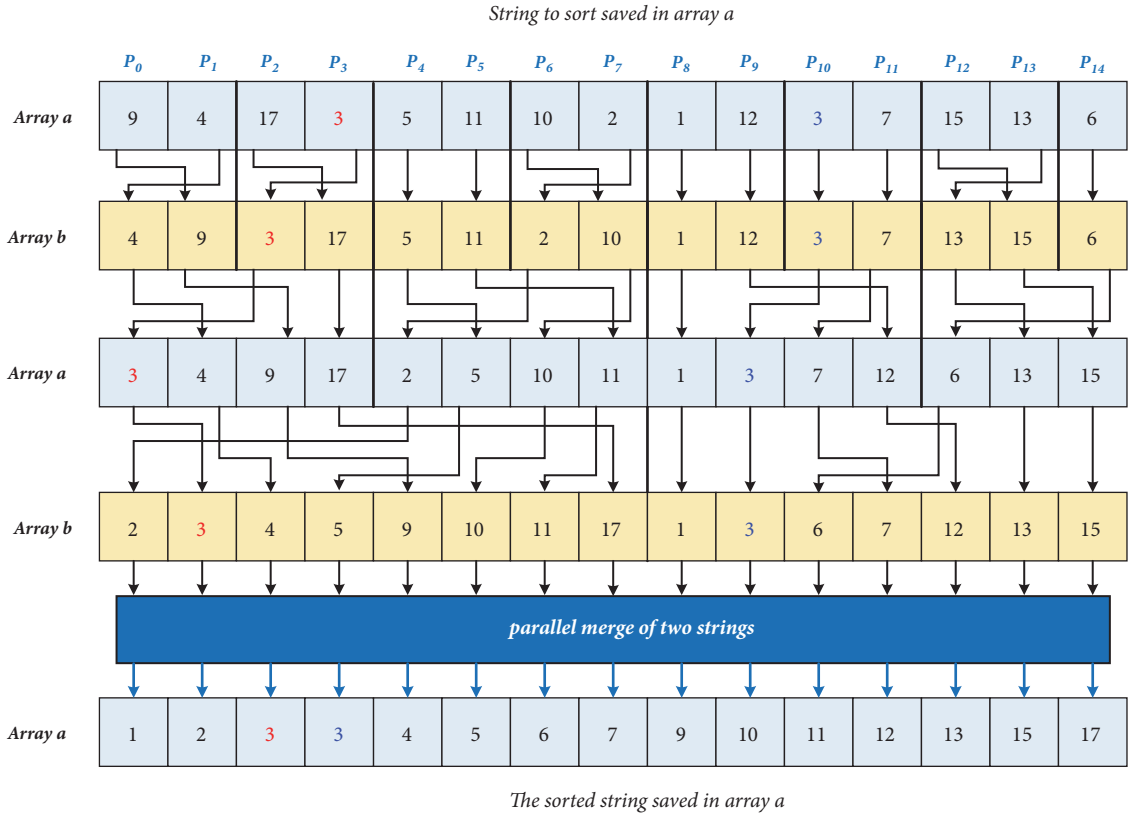

Figure 5: Fully Flexible Parallel Merge Sort Algorithm sorting the input array. The method is a stable method in the sense of keeping the order of arranging elements of the same value in a sorted sequence. In the drawing, this is shown on the element with value 3, which has been marked with different colors. In the sorted sequence, the elements remain in the stacking order, such as that they had in the input of not sorted sequence.

Exclusive Write gives the right to read from memory by any processor and the right to write to at the same time by only one processor. The third model of the Concurrent Read Concurrent Write PRAM enables simultaneous access of all processors to memory. Therefore in the description and experimental research we use CREW PRAM model as it is the most suitably fitting the operations in modern computer architectures. Proposing fully flexible algorithm on CREW PRAM model for merging two strings will perform in time $O((\log_2 n)^2)$.

In each iteration, the algorithm presented in Figure 5 merges four successive ordered sequences into one ordered numerical sequence. Each iteration is divided into two steps. In the first step, the next two strings from the array a are merged and the result of merging is stored in the table b. In the second step, the algorithm merges the next two ordered sequences from the array b and merged result is written into the array a. So each iteration increases the size of the ordered sequences four times. Because the algorithm uses a parallel version to merge two numeric strings, so in each iteration it can use n processors at every step which makes it flexible to the number of processors. For example, in the first iteration in the first step, two one elemental strings are merged by n processors; therefore, $n/2$ times the algorithm of parallel merging of two numerical sequences is run on independently working processors.

3.2. Theoretical Aspects of Computational Complexity. The Algorithm 2 is a flexible parallel merging of numeric strings without cross actions or interruptions. The presented method is flexible and can be adjusted to a large number of logical processors; therefore, we call it Fully Flexible Parallel Merge Sort (FFPMS). Each step of the merge was divided into two stages. First, the method merges each pair of strings into the temporary array. If at the end of the input array is one element string, it will be rewritten into the temporary array. Then the method merges each pair of sorted strings from the temporary array into the output array. Similarly as in the first stage, if there is one element at the end of the string, it will be rewritten from the temporary array into the output array. The result in the output array is increased four times. To perform the merge was developed the new parallel algorithm that determines the index of each element a_i, $i = 0, \ldots, n-1$. For each of the processors P_i, $i = 0, \ldots, n-1$ we call developed search algorithm to select the index of the element before which the new element a_i should be inserted. Because all processors can simultaneously read the memory and each processor executes the insertion of an element in another cell, the whole process can be run simultaneously without cross actions and interruptions. Figure 6 shows the first step of the first iteration of merging one element strings with the sorted strings of two elements stored in the temporary array. Way to parallelize it further in the process of merging n strings

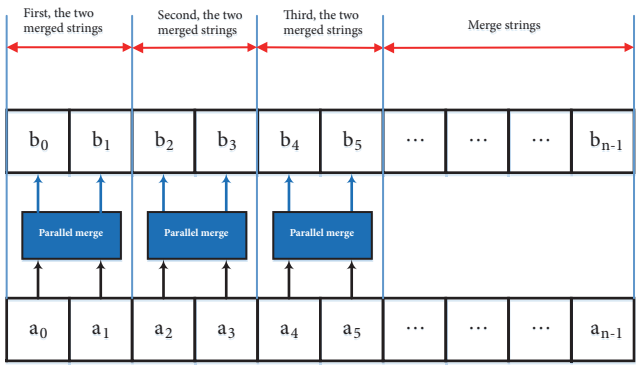

FIGURE 6: The first step of the proposed flexible parallel merge sort, when the algorithm merges elements of the input strings in pairs.

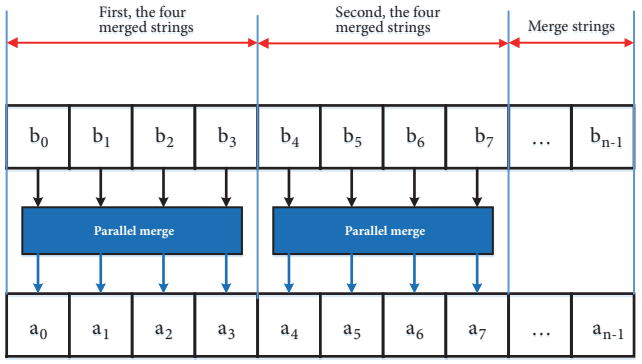

FIGURE 7: The following operations in the proposed flexible parallel merge of the temporary array into the input array.

is shown in Figure 7. In all the steps of the algorithm we merge the data in the same way, strings are enlarged four times and the odd indexed element is rewritten until the final merge, see Figure 8. The method is stable in the sense of keeping the order of arranging elements of the same value in a sorted numerical sequence. In Figure 5 we can see sample sorting, where stability of the order is shown on the elements with value 3, which have been marked with different colors. In the sorted sequence, the elements keep the order of the arrangement, such as they had in the input sequence.

Theorem I. *Proposed Fully Flexible Parallel Merge Sort for n processors has time complexity $O((\log_2 n)^2)$.*

Proof. We are limiting deliberations to $n = 4^k$, where $k = 1, 2, \ldots$.

Let us first notice that sequences $a_0 \leq a_1 \leq \ldots \leq a_{n/2-1}$ and $a_{n/2} \leq \ldots \leq a_{n-1}$ of $n/2$ elements can be merged into one sequence $b_0 \leq \ldots \leq b_{n-1}$ using n processors. Moreover parallel merging of these two sequences makes no more than $\lceil \log_2 n \rceil + C$ comparisons by each of the processors. Thus, time complexity of the parallel algorithm to merge two strings on a CREW PRAM machine model is $O(\log_2 n)$.

At each step $t = 1, \ldots, k = \log_4 n$, in the beginning the algorithm saves into the temporary array two merged halfs of the original string. Next, we merge them from the temporary array and save the result in the array of the sorted elements. Because all processors work independently, thread synchronization happens after each stage. The maximum operating time of each step of the merger of four strings can be written in the form

$$\begin{aligned} T_{\max}(t) &= \left\lceil \log_2 2^{t-1} \right\rceil + \left\lceil \log_2 \left(2 \cdot 2^{t-1}\right) \right\rceil + C_1 \\ &= 2 \left\lceil \log_2 2^{t-1} \right\rceil + C_2 \end{aligned} \tag{1}$$

The first component $\lceil \log_2 2^{t-1} \rceil$ is the maximum time for all 2^{t-1} components to merge. The second component $\lceil \log_2 (2 \cdot 2^{t-1}) \rceil$ is time to merge the double-expanded strings after merging strings in the first stage and writing them in the temporary table. The C_1 constant consists of a fixed number of operations performed both when merging items into temporary tables and a fixed number of operations when merging strings from a temporary table into an array of ordered items. It is independent of step t. By transforming

$$\begin{aligned} \left\lceil \log_2 \left(2 \cdot 2^{t-1}\right) \right\rceil &= \lceil \log_2 2 \rceil + \left\lceil \log_2 2^{t-1} \right\rceil \\ &= 1 + \left\lceil \log_2 2^{t-1} \right\rceil \end{aligned} \tag{2}$$

and inserting into (1) it yields that $2\lceil \log_2 2^{t-1} \rceil + C_2$ equals the maximum time that all processors execute sorting in each step t. Then we aggregate all the sorting times $\log_4 n$ performed by the processors in each step $t = 1, \ldots, k$ and write down the sum of the times as

$$T_{\max} = \sum_{t=1}^{k} T_{\max}(k) = 2 \sum_{t=1}^{k} \left\lceil \log_2 2^{t-1} \right\rceil + C_2 \log_4 n \tag{3}$$

when calculating

$$\begin{aligned} \sum_{t=1}^{k} \left\lceil \log_2 2^{t-1} \right\rceil &= \lceil \log_2 2 \rceil + 2 \lceil \log_2 2 \rceil + \ldots \\ &\quad + (k-1) \lceil \log_2 2 \rceil \\ &= \lceil \log_2 2 \rceil [1 + 2 + \ldots + (k-1)] \\ &= \frac{k(k-1)}{2} \lceil \log_2 2 \rceil \\ &= \frac{\log_4 n (\log_4 n - 1)}{2} \lceil \log_2 2 \rceil \end{aligned} \tag{4}$$

it is assumed that $n = 4^k$ and the result is consistent with our expectation, since the sum of the arithmetical progress of natural numbers is equal $k(k-1)/2$, where $k = \log_4 n$. Intuitively, it can be said that in the subsequent stages of the merge process each of the independent processors searches for the index of the item being rendered by means of a binary search of time complexity $O(t), t = 1, \ldots, \log_2 n$. Since the number of algorithm steps is $\log_2 n$, we obtain the method of time complexity $O[(\log_2 n)^2]$. Therefore by substituting and

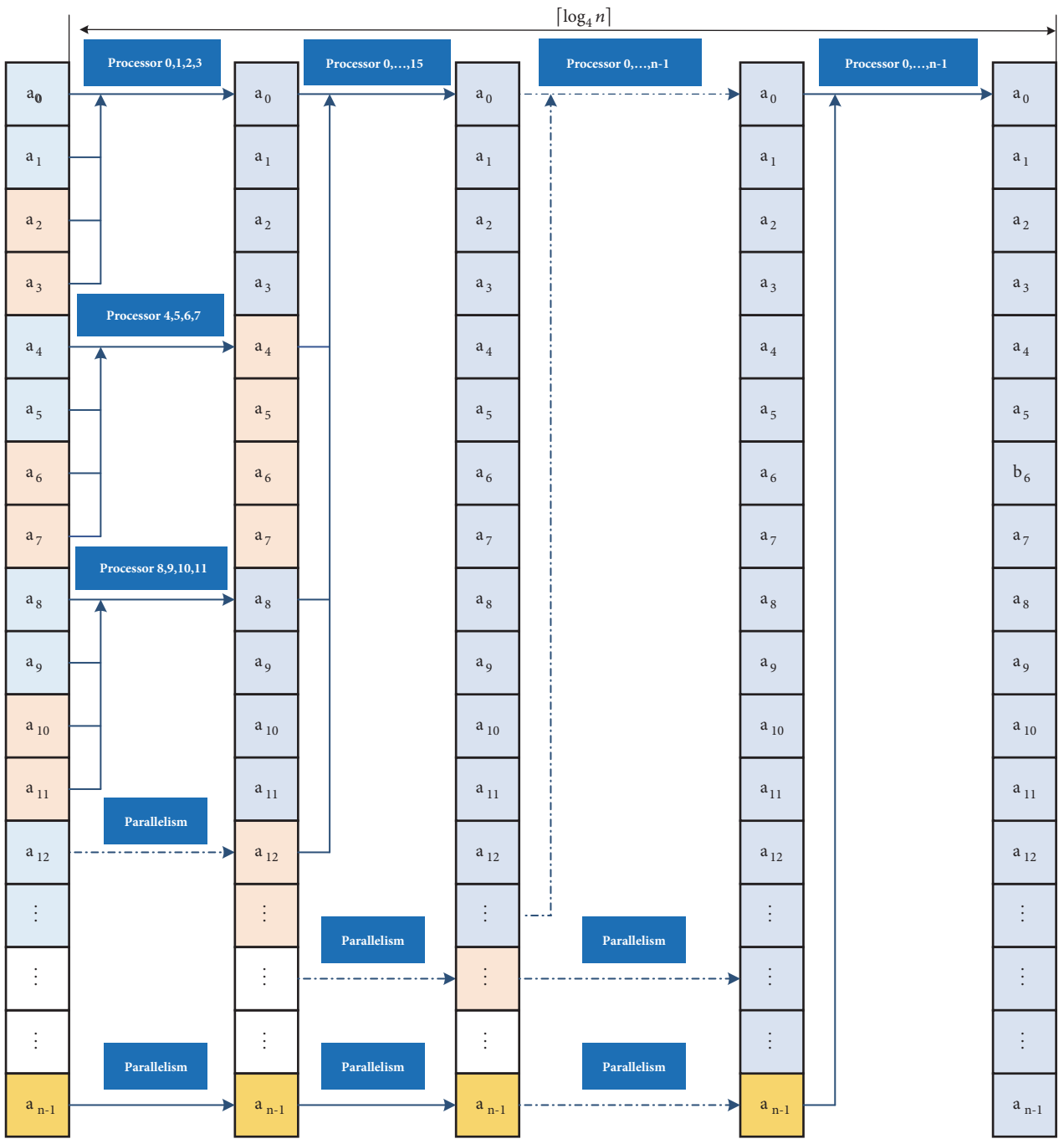

Figure 8: Fully Flexible Parallel Merge Sort Algorithm.

taking into account $\lceil \log_2 2 \rceil = 1$ and $\log_4 n = \log_2 n / \log_2 4 = \log_2 n / 2$, we get

$$T_{\max} = (\log_4 n)^2 + (C_2 - 1)\log_4 n = \frac{(\log_2 n)^2}{4} + \frac{(C_2 - 1)\log_2 n}{2} \quad (5)$$

which was proved. □

The square logarithm of the algorithms time complexity is the result of separating the work of processors in each iteration of the merge strings. Each processor operates independently, but it performs more string comparisons to determine the index of the inserted element. This enables each iteration to use the same number of processors working independently.

Theorem II. *By using k processors for the parallel sorting method on CREW PRAM, we can lower the time complexity to $O(n(\log_2 n)^2/k)$.*

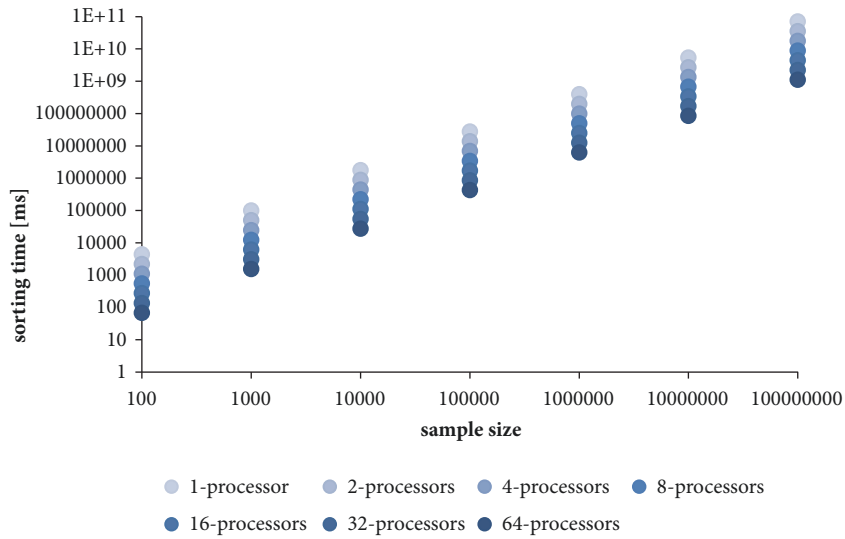

Figure 9: Chart of the theoretical complexity for the proposed method showing the potential growth in efficiency of the proposed approach for multicore architectures up to 64 cores.

Proof. The proof comes as natural derivation from the proof of Theorem I. □

If we assume that the time increment for an algorithm with the increase in the task dimension is a unit of time, then for parallel sorting method by merging we obtain the following graph of the algorithm's theoretical run time, see Figure 9. This theoretical result of $O(n(\log_2 n)^2/k)$, where k is the number of processors, is an asymptotic result which does not take into account the delays that may occur when data is transmitted on the data bus and other operations by the system as a result of the implementation. Hence, the difference in the speed of the algorithm is only visible for a sufficiently large task dimension. Analyzing Figure 9 we see that with each new added processor the method should gain additional advantage in sorting big data sets. The chart shows theoretical results for architectures of up to 64 logical processors. We can conclude that in theoretical way from 1 to 64 logical processors the method should gain about 10% to 15% on efficiency which is very important for big data sets. The proposed algorithm is fully scalable and can be executed for a veritable dimension of the sorted sequence of numbers using the specified number of processors without any problems.

For benchmark tests the method was implemented in C# Visual Studio Enterprise 2015. To assign tasks and to facilitate processors loop *Parallel For* available in C# language has been used, which enables the usage of the maximum number of processors available in the system. In the method we have two additional functions targeted at the delivery of the index of the element before inserting the new ones. The first function returns an index to the next element in the string on the right side, see Figure 10. The second function returns an index of the next element in the string on the left side, see Figure 11.

4. The Study of the Algorithm

Performance analysis is based on benchmark tests for the algorithms implemented in C# in Visual Studio 2015 Enterprise on MS Windows Server 2012, namely: quick sort, heap sort, and classic merge, and presented here is flexible parallel merge. For testing were used 100 samples generated at random for the task size from 100 to 100 000 000 elements, increasing the size of sorted array ten times in the following experiments. In addition, for each set of 100 samples generated randomly for a given dimension size were added samples consisting numbers ascending and descending, as well as samples containing numbers which compose a critical situation for sorting algorithms (Woźniak et al. [44]). The number of samples was chosen as 100 since it is a standard statistical number to examine proposed methods in benchmark tests.

4.1. Benchmark Tests. Let us now show practical verification of the computational complexity and comparisons to other methods. We are interested in presenting how the method works, but we do not assume delays from hardware, bus of the motherboard, hard disk connectors, etc. The tests we present here are focused on measuring efficiency of applied processors. We assume that tests are free of other delays and compare the results to evaluate how the sorting methods are run on processors. In comparisons we have used statistical measures. Each of the experiments was verified for sorting time measures in [ms] and Central Processing Unit (CPU) operations measured in [ti]. The arithmetic mean is equal to the mean value of the measurements from the experiments

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{6}$$

The standard deviation from the expected value is computed as

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}} \tag{7}$$

where n represents the number of measurements $x_1, x_1, \ldots, x_n$, and $\overline{x}$ is the arithmetic mean (6). The

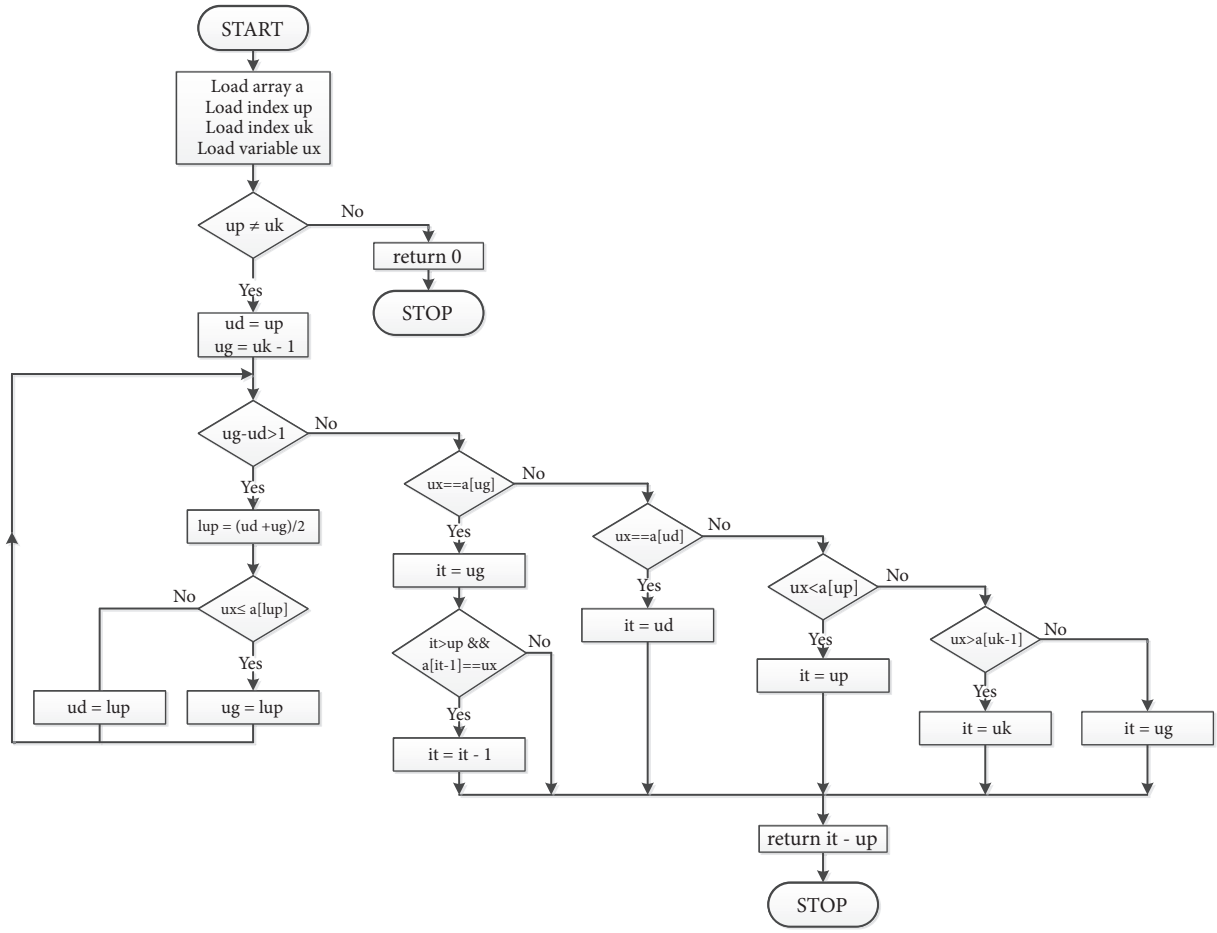

Figure 10: The block diagram of the function which returns index located in the right string.

coefficient of variation presents possible diversity between the results and is computed as

$$V = \frac{\sigma}{\overline{x}} \tag{8}$$

where the symbols are the arithmetic mean (6) and the standard deviation 7. Each sorting operation was measured in time [ms] and CPU (Central Processing Unit) usage represented in tics of CPU clock [ti]. Tests were carried out on quad core amd opteron processor 8356 8p. These results are averaged for 100 experiments to show comparison presented in Figures 12 and 13. The results of the newly proposed method are presented in Table 1 for time and Table 2 for computing operations. Comparison of coefficient of variation is presented in Tables 3 and 4.

Analyzing Tables 1 and 2 we see that with each new processing core the efficiency is extended and the sorting is done faster. While from Tables 3 and 4 we see that the algorithm for any number of CPUs used in the research has almost the same stability for large data sets. Some variations in stability of the algorithm for small inputs are due to the fact that the system exceeds the sorting capability automatically, what has an influence on the performance.

4.2. Analysis and Comparison. Let us compare the algorithms assuming that duration of the method using only one processor is a base line and let us examine if the duration is shorter when using multiple processors. The results are shown in Figures 14 and 15. Comparing the results in Figures 14 and 15 we can see that each new processor gives additional boost to sorting by decreasing sorting time and necessary operations and therefore makes the method more efficient. The most spectacular difference is between one and two processors. Figure 14 shows that time of processing can be really shorter for large collections. For collections above 10 000 elements we can see the boost in sorting time and the differences by the use of additional processors become visible. For collections above 1 000 000 elements the sorting time becomes shorter with each newly added processor of about 10%-15%. The dashed lines representing trends confirm this and show that additional processors seriously boost the proposed method. Figure 15 shows that the application of further processors improves the method of about 40% if we switch from 1 processor to 2 processors. Additionally for adding the next 2 processors we can get another 20% of efficiency. Next processors give about 10% to 15% rise in the efficiency with each new logical core used in the method. The

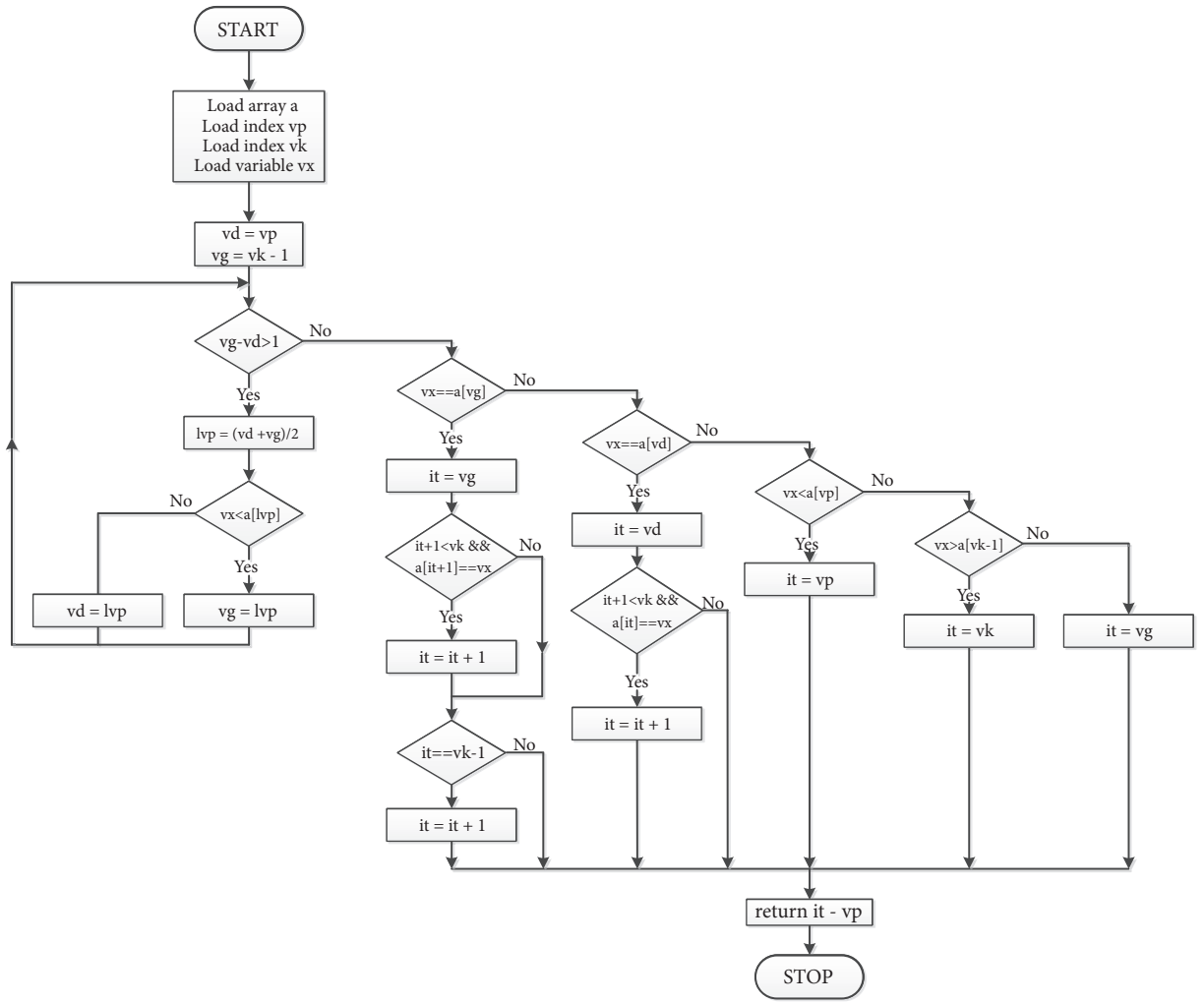

Figure 11: The block diagram of the function which returns index located in the left string.

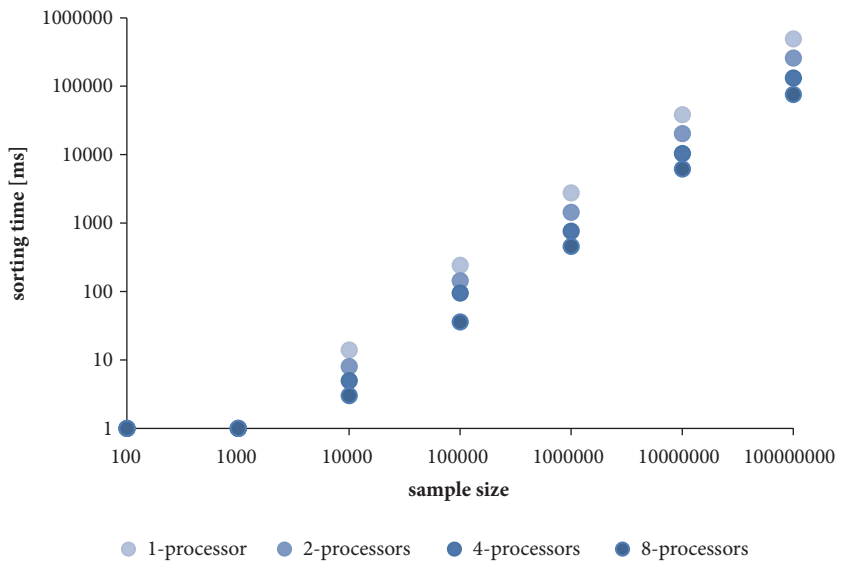

Figure 12: Comparison of benchmark time [ms].

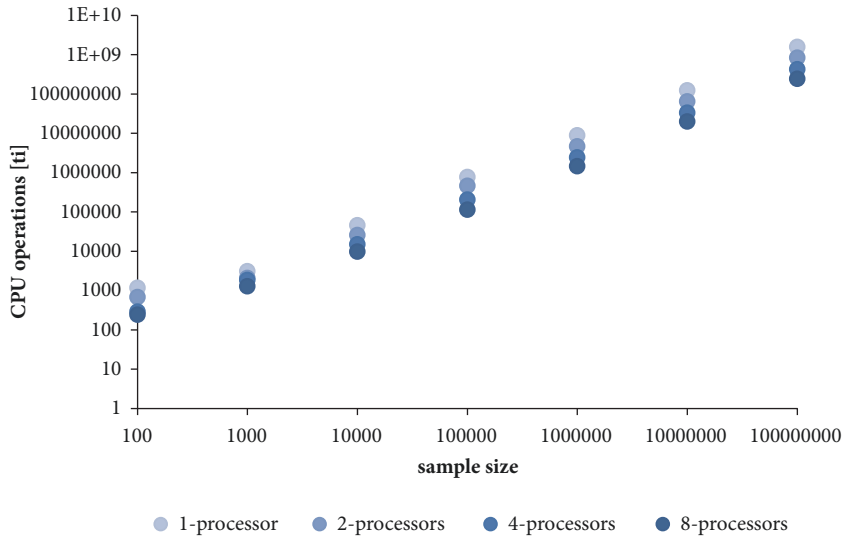

Figure 13: Comparison of benchmark operations [ti].

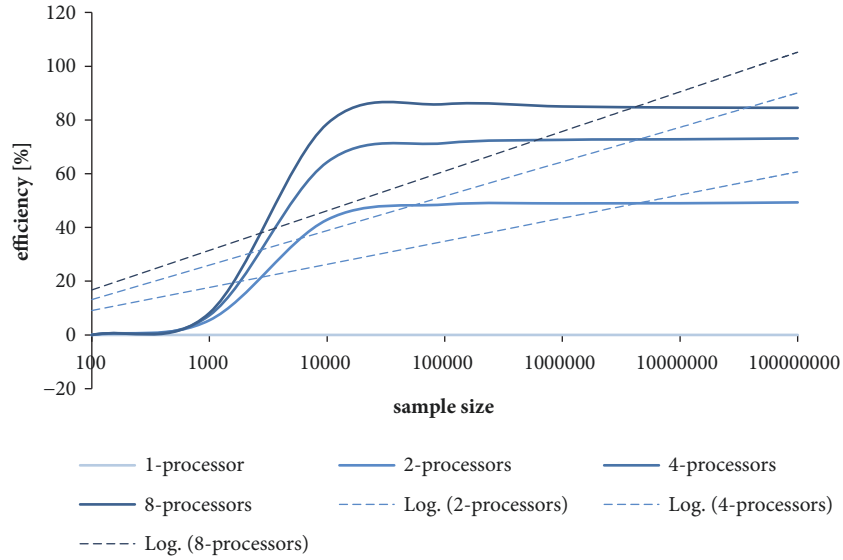

Figure 14: Comparison of the benchmark efficiency using multiple processors in terms of operation time [ms].

dashed trend lines confirm the stability of usage of computing powers by the proposed method. That shows possibility to improve sorting on multicore architectures. Due to proposed implementation the method is fully flexible; therefore, we can add a large number of processors to speed up sorting. This is very important for new computer architectures. In Figure 16 we can see a comparison of the proposed method to other sorting algorithms. As a baseline for comparisons was selected heap sort algorithm. We can see that FFPMS is performing much faster in comparison to other methods. The usage of additional processors makes improvements for large data sets. We can see a difference between using 8 and 12 processors in FFPMS, which is visible for sets above 1 000 000 elements. Other methods are much less efficient and might have deadlocks. For small sets quick sort (Woźniak et al. [44]) and classic merge (Woźniak et al. [46]) work very similar. Proposed method is flexible, what means we can use a various number of processors. However it does not mean that the method is boosted the same with each new processor. There are some limitations visible also in our research. There exists significant improvement from 4 processors to 8 processors; however, from 8 processors to 12 processors this improvement is lower and in the above comparisons is visible for collections above 10 000 000 elements. Dashed lines for trends confirm numerical results. We can see that trend lines for FFPMS are growing wich means that the proposed method shall be more efficient for big data sets. Similar grow in trend line is visible for classic merge; however, this method is about 15% less efficient. Trend line for quick sort is decreasing what shows that quick sort will be losing efficiency for big data sets. These benchmark tests confirm high potential of the proposed flexible parallel method.

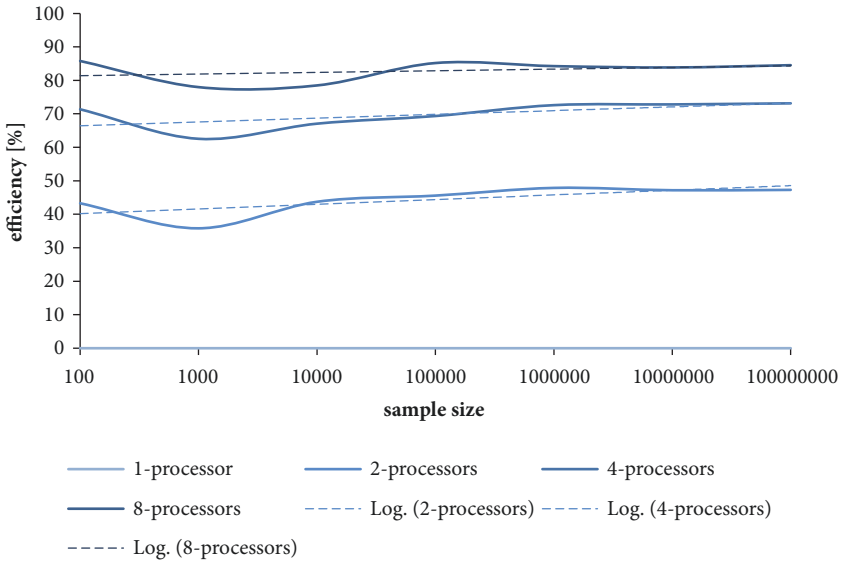

Figure 15: Comparison of the benchmark efficiency using multiple processors in terms of operations [ti].

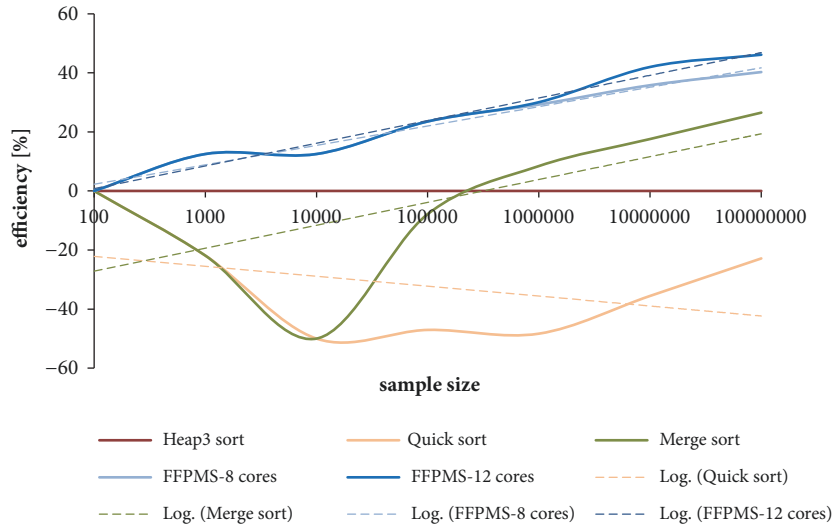

Figure 16: Benchmark comparison of the proposed Fully Flexible Parallel Merge Sort on 8 (light blue) and 12 (blue) cores to other sorting algorithms: heap sort with 3 divisions (brown), quick sort (cream), and classic merge sort (green) presented with logarithmic trend lines (dashed).

Table 1: Average benchmark sorting time for 100 samples in [ms].

Elements	1 - processor	2 - processors	4 - processors	8 - processors
100	1	1	1	1
1 000	1	1	1	1
10 000	14	8	5	3
100 000	241	124	69	36
1 000 000	2770	1414	759	415
10 000 000	38341	19552	10423	5892
100 000 000	491229	248920	131960	75985

Table 2: Average benchmark sorting operations for 100 samples in [ti].

Elements	1 - processor	2 - processors	4 - processors	8 - processors
100	1379	782	395	196
1 000	3097	1988	1160	682
10 000	45845	25795	15099	9863
100 000	775806	422189	237789	115017
1 000 000	8923418	4650899	2445417	1405403
10 000 000	123523898	65244951	33581034	19949470
100 000 000	1582604411	834171066	425138053	244802699

Table 3: Coefficient of variation for sorting time [ms].

Elements	1 - processor	2 - processors	4 - processors	8 - processors
100	0.3413121	0.2931151	0.3012821	0.3043211
1 000	0.2257182	0.1152201	0.1923175	0.1831273
10 000	0.1549049	0.1118317	0.2267786	0.1781741
100 000	0.1092983	0.1260794	0.2274779	0.0921129
1000 000	0.0809068	0.0836661	0.0826251	0.0717478
10000 000	0.0707954	0.0789099	0.0739682	0.0664342
100000 000	0.0687242	0.0737327	0.0721888	0.0632200

Table 4: Coefficient of variation for sorting operations [ti].

Elements	1 - processor	2 - processors	4 - processors	8 - processors
100	0.2490578	0.4343373	0.3066603	0.2368111
1 000	0.1389146	0.0654149	0.1422623	0.1645332
10 000	0.0856474	0.0501013	0.0694751	0.0769935
100 000	0.0903991	0.0959194	0.0853802	0.0600897
1000 000	0.0808740	0.0837383	0.0827343	0.0717861
10000 000	0.0707980	0.0789032	0.0739785	0.0664350
100000 000	0.0687246	0.0737334	0.0721900	0.0632179

4.3. Conclusions. The study shows that FFPMS operates in shorter time measuring tasks above 1 000 000. The proposed method is effective when using a large number of processors available in modern chip-sets. Its theoretical complexity is $O(n(\log_2 n)^2/k)$, where k is the number of logical processors that are used in calculations. Tests conducted on a limited number of threads fully confirm the theoretical results of the research, showing a clear improvement with each new additional processor.

Reduced sorting time is a big advantage for large data sets. Moreover the method does not have deadlocks. Proposed implementation gives a separation of concerns which makes it possible to freely enlarge the number of processors used for sorting. Other methods are less efficient. During tests merge sort, quick sort and heap sort gave results about 10% to 20% worse. From the research results we can conclude that merge sort is the only algorithm that can compete with FFPMS, but still these results are about 10% worse. At every step of the algorithm, the transformation of the arrays is mutually univocal and unique. Therefore, it will never happen that an element is inserted by the processor outside the last element of the array. In addition, each processor inserts exactly to one memory cell and no other processor inserts into the same memory cell.

Due to the fact that the FFPMS algorithm is particularly effective in the case of merging large numerical sequences using a large number of processors, it seems to be purposeful to additionally combine it with the algorithm described in Marszałek [51]. In a hybrid sorting method designed in this way, the initial merging steps would be performed using Marszałek [51], and the final steps of merging long numerical sequences would be performed using the algorithm presented in this paper. The method constructed in such a way would enable the use of a large number of processors in entire sorting process in even more efficient manner. This will be one of the directions in our future research.

The complexity of the square from the logarithm of the dimension of the sorting task is asymptotically smaller than the element from the task's dimension. The comparison is presented in Table 5. That is why, even on one processor, we obtain sorting results for the 100 000 000 dimension in a very short time, and you cannot receive such results in real time for other sorting methods. In Table 5 we have comparison to other algorithms. We can see that Grover algorithm

Table 5: Comparison of complexity for sorting operations.

Elements	$\lceil \sqrt{n} \rceil$	$\lceil \log_2 n \rceil$	$\lceil \log_2^2 n \rceil$
100	10	7	49
1 000	32	10	100
10 000	100	14	196
100 000	317	17	289
1000 000	1 000	20	400
10000 000	3163	24	576
100000 000	10 000	27	729

on quantum computer (represented in $\lceil \sqrt{n} \rceil$ complexity) is much slower, and similarly we see comparison to binary search algorithm complexity (represented in $\lceil \log_2 n \rceil$ complexity). Our method (represented in $\lceil \log_2^2 n \rceil$ complexity) is very efficient, and let us sort inputs even faster than on quantum computer. Comparing to some other classic methods like bubble sorting method (complexity n^2) we see that the proposed algorithm shows very high boost.

In the implementation of the presented sorting algorithm was used the C# class *System.Threading.Tasks*. This class has a parallel for loop, which is not an iterative in classic understanding, but an iteration assigns tasks to the processes which are computed for each of the processors. Subsequent processes receive the identifier of consecutive natural numbers starting with zero and ending with one less than the number of processes. According to the object-oriented programming principle, variables declared inside this loop are only available for a given thread, and arrays and variables declared outside the parallel loop are available for all processes. Synchronization of all processes occurs after the parallel loop ends. A simple rule applies here: all processes can read the arrays and global variables, but only one of them can write to the same memory location. While transferring data between memory and processes the cache compartment is done automatically and the programmer is not affected. Therefore proposed implementation is very complex and makes the program oriented on maximum efficiency for the parallelization of sorting processes.

The novelty of the presented algorithm is the way of dividing tasks so that the processes do not interfere with each other. Each of the processes inserts the specified item into the merged string only on the basis of the information, and there are no parallel loop insertions in the same memory cell. This division of work in processes increases the computational complexity of the algorithm with a small number of processors. At the same time it increases the sorting performance of servers with a large number of processors and large operating memory (such as Oracle SPARC M8-8). The SPARC M8-8 is equipped with 256 cores with 8 interleaves, a total of 2048 logical processors and 76TB of memory. For this type of computer the proposed method will be a perfect solution, which can independently operate on each of the processors without cross actions. Therefore the efficiency of our method will be much higher. However our method also fits Microsoft software processors manufacturer like Intel and AMD, which were used in presented tests. The method strives to match any number of processors and can be implemented actually in any programming language. In our tests, according to the available licenses for the software and hardware testing, the algorithm was performed in C# language under Visual Studio. There should be no problem to rewrite the implementation of the proposed algorithm for Oracle SPARC M8 server in Java. Oracles Java is an object-oriented language which has rich library for handling multithreaded algorithms and applications. The results of statistical studies presented in this paper guarantee the reproducibility of the obtained results on other servers.

From the research have arisen some interesting open questions. In the tests we tried to present the method working on various numbers of processors. We assumed that there were no delays from hardware, bus of the motherboard, hard disk connectors, etc. However these aspects shall be verified in our future research. We also plan to concentrate on some more sophisticated data structures which can additionally support data management in the system. It will be also interesting to develop devoted versions of the presented algorithm for specialized management systems used in medicine, geoscience, financial systems, etc.

The algorithm presented in the paper has been designed for a typical architecture that corresponds to the modern computers equipped with multithreaded cores with a fast cooperative memory. This corresponds to the Uniform Memory Access (UMA) model, and in principle there is no difference in performance relative to the CREW PRAM model. The computer network on the PRAM model is similar in actions to the data bus through which the processors have access to the entire memory. Of course, it is possible to distribute calculations on many computers over the Internet which is done in the UMA model. However, this would require additional research on the possibility of applying the presented idea to distributed systems. The No-Remote Memory Access (NoRMA) architecture model is very complex and does not correspond to the processing of data on modern computers, so it was not considered in this work. The question of the possibility of using the proposed method for GPUs remains open. General-Purpose computing on Graphics Processing Unit (GPGPU) shows the computing power used by graphic cards where the processor cores work together. However, it is important to note that the processors used there are low in computing power and that the graphic card's performance is decisive in principle. An additional aspect is the operating memory that the graphic chip-set has. For modern architectures we can talk about very efficient memory in terms of data capacity and access speed. In the case of graphic cards, the processors and memory used in them are much less efficient. Thus, for large data sets, there may be situations in which such systems will not sufficiently cope with processing input data, and thus the sorting methods may lose a significant portion of performance. Therefore, in the assumption of the developed method we use the concept of the main processor from motherboard as appropriate for the presented method of computing.

An interesting research conclusion would be also in relation to quantum computing. The current state of knowledge

indicates that the best results for quantum computing are achieved only theoretically and we still have no real quantum machines. The reason is unstable energy allocation in quantum machines. According to our knowledge there is no such algorithm for current computers, which can show comparable results to quantum ones. However, in this work we have shown that our algorithm is able to match quantum efficiency (which exists only in theory at the present time), but as shown here it works in reality on real multicore architectures. This is very important feature of our method.

5. Final Remarks

The article presents a Fully Flexible Parallel Merge Sort Algorithm which is composed to preserve high performance in sorting at the lowest possible computational complexity.

Presented in this article is a method for an effective way to organize large amounts of data using a number of processors. The method was developed using separation of concerns; therefore, there are no cross actions or interferences between processors. The proposed model is fully flexible for various number of processors. The tests confirmed the theoretical computational complexity and the stability of the algorithm. Moreover, the achievement of an operational time equals to $O(\sqrt{N})$, which is an interesting innovation considering that no better result was yet achieved by any sort algorithm on classic computers as well as in quantum theory. An example is the Grover algorithm, which is considered to be the best one because it can execute $O(\sqrt{N})$ queries, while our proposition has similar result. It is worth noting that the current hardware state does not give the possibility to implement the Grover algorithm on the dedicated machine, which makes it still only theoretical proposition. Unlike the method presented in this work, it is implemented and flexible to a growing number of operational cores.

Data Availability

The article presents a method which can be used to manage data in computer systems so we do not have any special data set to report.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

Authors acknowledge contribution to this project from the Rector of the Silesian University of Technology under grant for perspective professors no. 09/010/RGH18/0035.

References

[1] K. Bochenina, S. Kesarev, and A. Boukhanovsky, "Scalable parallel simulation of dynamical processes on large stochastic Kronecker graphs," *Future Generation Computer Systems*, vol. 78, pp. 502–515, 2018.

[2] P. Czarnul, J. Kuchta, M. Matuszek et al., "MERPSYS: An environment for simulation of parallel application execution on large scale HPC systems," *Simulation Modelling Practice and Theory*, vol. 77, pp. 124–140, 2017.

[3] H. Mora, D. Gil, R. M. Terol, J. Azorín, and J. Szymanski, "An IoT-Based Computational Framework for Healthcare Monitoring in Mobile Environments," *Sensors*, vol. 17, no. 10, p. 2302, 2017.

[4] A. Esmaeili, N. Mozayani, M. R. Jahed Motlagh, and E. T. Matson, "A socially-based distributed self-organizing algorithm for holonic multi-agent systems: Case study in a task environment," *Cognitive Systems Research*, vol. 43, pp. 21–44, 2017.

[5] L. Stanescu, M. Brezovan, and D. D. Burdescu, "Automatic mapping of MySQL databases to NoSQL MongoDB," in *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016*, pp. 837–840, Poland, September 2016.

[6] M. Janetschek, R. Prodan, and S. Benedict, "A workflow runtime environment for manycore parallel architectures," *Future Generation Computer Systems*, vol. 75, pp. 330–347, 2017.

[7] V. A. E. Farias, F. R. C. Sousa, J. G. R. Maia, J. P. P. Gomes, and J. C. Machado, "Regression based performance modeling and provisioning for NoSQL cloud databases," *Future Generation Computer Systems*, vol. 79, pp. 72–81, 2018.

[8] M. T. González-Aparicio, M. Younas, J. Tuya, and R. Casado, "Testing of transactional services in NoSQL key-value databases," *Future Generation Computer Systems*, vol. 80, pp. 384–399, 2018.

[9] A. V. Aho and J. E. Hopcroft, *The design and analysis of computer algorithms*, Pearson Education India, 1974.

[10] D. E. Knuth, *The art of computer programming. Vol. 3*, Addison-Wesley, Reading, MA, 1998.

[11] H. Bing-Chao and D. E. Knuth, "A one-way, stackless quicksort algorithm," *BIT. Nordisk Tidskrift for Informationsbehandling (BIT)*, vol. 26, no. 1, pp. 127–130, 1986.

[12] R. S. Francis and L. J. H. Pannan, "A parallel partition for enhanced parallel QuickSort," *Parallel Computing*, vol. 18, no. 5, pp. 543–550, 1992.

[13] A. Rauh and G. R. Arce, "A fast weighted median algorithm based on Quickselect," in *Proceedings of the 2010 17th IEEE International Conference on Image Processing, ICIP 2010*, pp. 105–108, Hong Kong, September 2010.

[14] P. Tsigas and Y. Zhang, "A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN Enterprise 10000," in *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Euro-PDP 2003*, pp. 372–381, Italy, February 2003.

[15] A. M. Daoud, H. Abdel-Jaber, and J. Ababneh, "Efficient non-quadratic quick sort (NQQuickSort)," *Communications in Computer and Information Science*, vol. 194, pp. 667–675, 2011.

[16] J. Edmondson, "M pivot sort–replacing quick sort presenter," in *Proceedings of the James edmondson conference: Amcs05–2005 world congress in applied computing*, 2005.

[17] S. Kushagra, A. López-Ortiz, J. I. Munro, and A. Qiao, "Multi-pivot quicksort: Theory and experiments in," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 47–60, 2014.

[18] M. Ben-Or, "lower bounds for algebraic computation trees," in *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 80–86.

[19] E.-E. Doberkat, "Inserting a new element into a heap," *BIT: Nordisk Tidsskrift for Informationsbehandling*, vol. 21, no. 3, pp. 255–269, 1981.

[20] L. M. Wegner and J. I. Teuhola, "The External Heapsort," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 917–925, 1989.

[21] S. Sumathi, A. M. Prasad, and V. Suma, "Optimized Heap Sort Technique (OHS) to Enhance the Performance of the Heap Sort by Using Two-Swap Method," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, vol. 327 of *Advances in Intelligent Systems and Computing*, pp. 693–700, Springer International Publishing, Cham, 2015.

[22] S. Roura, "Digital access to comparison-based tree data structures and algorithms," *Journal of Algorithms in Cognition, Informatics and Logic*, vol. 40, no. 1, pp. 1–23, 2001.

[23] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, "A simple parallel tree contraction algorithm," *Journal of Algorithms in Cognition, Informatics and Logic*, vol. 10, no. 2, pp. 287–302, 1989.

[24] S. Carlsson, C. Levcopoulos, and O. Petersson, "Sublinear merging and natural mergesort," *Algorithmica. An International Journal in Computer Science*, vol. 9, no. 6, pp. 629–648, 1993.

[25] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[26] G. Gediga and I. Düntsch, "Approximation quality for sorting rules," *Computational Statistics & Data Analysis*, vol. 40, no. 3, pp. 499–526, 2002.

[27] J. D. Harris, "Sorting unsorted and partially sorted lists using the natural merge sort," *Software: Practice and Experience*, vol. 11, no. 12, pp. 1339-1340, 1981.

[28] B. Salzberg, "Merging sorted runs using large main memory," *Acta Informatica*, vol. 27, no. 3, pp. 195–215, 1989.

[29] B.-C. Huang and M. A. Langston, "Practical In-Place Merging," *Communications of the ACM*, vol. 31, no. 3, pp. 348–352, 1988.

[30] L. Zheng and P.-Å. Larson, "Speeding up external mergesort," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 2, pp. 322–332, 1996.

[31] W. Zhang and P. A. Larson, "Dynamic memory adjustment for external mergesort," in *VLDB, Citeseer*, pp. 25–29, 1997.

[32] W. Zhang and P. A. Larson, "Buffering and read-ahead strategies for external mergesort," in *The VLDB Journal*, pp. 523–533, 1998.

[33] R. Vignesh and T. Pradhan, "Merge sort enhanced in place sorting algorithm," in *Proceedings of the 2016 International Conference on Advanced Communication Control and Computing Technologies, ICACCCT 2016*, pp. 698–704, India, May 2016.

[34] S. M. Cheema, N. Sarwar, and F. Yousaf, "Contrastive analysis of bubble & merge sort proposing hybrid approach," in *Proceedings of the 6th International Conference on Innovative Computing Technology, INTECH 2016*, pp. 371–375, Ireland, August 2016.

[35] S. Paira, S. Chandra, and S. K. S. Alam, "Enhanced Merge Sort-A New Approach to the Merging Process," in *Proceedings of the 6th International Conference On Advances In Computing and Communications, ICACC 2016*, pp. 982–987, India, September 2016.

[36] T. O. Alanko, H. H. A. Erkio, and I. J. Haikala, "Virtual Memory Behavior of Some Sorting Algorithms," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 422–431, 1984.

[37] P.-Å. Larson and G. Graefe, "Memory management during run generation in external sorting," *SIGMOD Record*, vol. 27, no. 2, pp. 472–483, 1998.

[38] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of sorting," *Journal of Algorithms in Cognition, Informatics and Logic*, vol. 31, no. 1, pp. 66–104, 1999.

[39] P. Crescenzi, R. Grossi, and G. F. Italiano, "Search data structures for skewed strings," in *Experimental and efficient algorithms*, vol. 2647 of *Lecture Notes in Comput. Sci.*, pp. 81–96, Springer, Berlin, 2003.

[40] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys*, vol. 24, no. 4, pp. 441–476, 1992.

[41] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, "Practical massively parallel sorting," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2015*, pp. 13–23, USA, June 2015.

[42] S. Abdel-Hafeez and A. Gordon-Ross, "An efficient O(N) comparison-free sorting algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1930–1942, 2017.

[43] A. S. Mohammed, Ş. E. Amrahov, and F. V. Çelebi, "Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort," *Future Generation Computer Systems*, vol. 71, pp. 102–112, 2017.

[44] M. Woźniak, Z. Marszałek, M. Gabryel, and R. K. Nowicki, "Preprocessing Large Data Sets by the Use of Quick Sort Algorithm," in *Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions*, vol. 364 of *Advances in Intelligent Systems and Computing*, pp. 111–121, Springer International Publishing, Cham, 2016.

[45] M. Woźniak, Z. Marszałek, M. Gabryel, and R. Nowicki, "Triple heap sort algorithm for large data sets," *Looking into the Future of Creativity and Decision Support Systems*, pp. 657–665, 2013.

[46] M. Woźniak, Z. Marszałek, M. Gabryel, and R. K. Nowicki, "Modified Merge Sort Algorithm for Large Scale Data Sets," in *Artificial Intelligence and Soft Computing*, vol. 7895 of *Lecture Notes in Computer Science*, pp. 612–622, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[47] Z. Marszałek, M. Woźniak, G. Borowik et al., "Benchmark Tests on Improved Merge for Big Data Processing," in *Proceedings of the Asia-Pacific Conference on Computer-Aided System Engineering, APCASE 2015*, pp. 96–101, Ecuador, July 2015.

[48] D. Czerwinski, "Digital Filter Implementation in Hadoop Data Mining System," in *Computer Networks*, vol. 522 of *Communications in Computer and Information Science*, pp. 410–420, Springer International Publishing, Cham, 2015.

[49] A. Uyar, "Parallel merge sort with double merging," in *Proceedings of the 8th IEEE International Conference on Application of Information and Communication Technologies, AICT 2014*, Kazakhstan, October 2014.

[50] Z. Marszałek, "Novel Recursive Fast Sort Algorithm," in *Information and Software Technologies*, vol. 639 of *Communications in Computer and Information Science*, pp. 344–355, Springer International Publishing, Cham, 2016.

[51] Z. Marszałek, "Parallelization of modified merge sort algorithm," *Symmetry*, vol. 9, no. 9, 2017.