symmetry

MDPI

Article

Parallelization of Modified Merge Sort Algorithm

Zbigniew Marszałek

Institute of Mathematics, Silesian University of Technology, ul. Kaszubska 23, 44-100 Gliwice, Poland; Zbigniew.Marszalek@polsl.pl; Tel.: +48-32-237-1341

Received: 18 August 2017; Accepted: 30 August 2017; Published: 1 September 2017

Abstract: Modern architectures make possible development in new algorithms for large data sets and distributed computing. The newly proposed versions can benefit both from faster computing on the multi core architectures, and intelligent programming techniques that use efficient procedures available in the latest programming studios. Frequently used algorithms to sort arrays of data in NoSQL databases is merge sort, where as NoSQL we understand any database without typical SQL programming interpreter. The author describes how to use the parallelization of the sorting processes for the modified method of sorting by merging for large data sets. The subject of this research is the claim that the parallelization of the sorting method is faster and beneficial for multi-core systems. Presented results show how the number of processors influences the sorting performance. The results are presented in theoretical assumptions and confirmed in practical benchmark tests. The method is compared to other sorting methods like quick sort, heap sort, and merge sort to show potential efficiency.

Keywords: parallel algorithm; data sorting; distributed computing; analysis of computer algorithms

1. Introduction

Computer technology is constantly developed and new architectures are introduced to the market. The design of machines with multiple cores involves programming for many logical processors working independently. The software is oriented on devoted separation of concerns for efficient information processing. These aspects make it possible to manage even more information in various data bases. Algorithms that are very helpful in any data base system are various sorting methods. Classic versions that began research on possible developments were presented in [1,2]. We can distinguish three main types of sorting algorithms: quick sort, heap sort, and merge sort. Along with new processors the new, modified versions of these algorithms were presented. In the developments, we can find some particular modifications for selected data, improved procedures to avoid deadlocks, and new structures that made it possible to increase the speed of sorting.

Quick sort is composed using divisions of the data stack, in which each part is processed to organize elements in relation to a selected divider. Many versions of this method show possible improvements for speeding up the process and preventing deadlocks. A devoted pivot mechanism for faster stack exchange was presented in [3]. Possibilities to use different position of partitioning were discussed in [4]. Also, some derivatives from these methods were introduced. The introduction of a median value for divisions exchange was presented in [5]. This method was tested on various architectures and chipsets, with some results for the Sun Microsystems, Inc. machine were presented in [6]. The quick sort was examined extensively, and the research benefitted in some new versions. A non-quadratic version of this method was proposed in [7]. In [8] was presented some modification of pivot procedure for sort alignments, which was presented as the new version of the quick sort. Multi-pivot version of the quick sort was presented in [9]. Heap sort is using the multilevel structure of data storage, where introduced relations between the following levels influence the speed of sorting. Each change in the structure requires the procedure to insert elements into the heap. Mathematical

models of the relations between levels of the heap were discussed in [10,11]. Discussion of the efficiency of various possibilities for this procedure was presented in [11]. Research on possible changes between levels of the heap was presented in [12], while two swap procedure for arranging elements into the heap was discussed in [13]. Tests on access efficiency were presented in [14], and a parallel version of this method was discussed in [15]. Merge sort is using the "divide and conquer" rule, which suggests the division of the input data into smaller parts that are sorted during following operations of merging into one string. Sublinear merging was presented in [16]. A theoretical approach to composition of the first parallel version of merge sort was presented in [17]. Derivative of this method for partially sorted strings was presented in [18]. Tests on practical implementations were discussed in [19], while tests on memory usage were presented in [20,21]. A derivative for more efficient input-output management was presented in [22], and tests on improved memory management by dynamic division assignment were discussed in [23]. Research on possible improvements in buffering and reading were presented in [24]. Possible enhancements to the merging procedure were discussed in [25]. The research on possible improvements were also based on extensive comparison to other methods. In [26] was presented an analysis of the merge sort in comparison to the bubble sort. In [27] was given an introduction to some new interesting ideas for possible implementation of merging procedures that can boost the algorithm.

These three sorting methods were also mixed and combined to compose some new algorithms for devoted purposes. One of the very important aspects is memory management. In [28,29] it was discussed how the usage of virtual memory influences the speed of sorting. Benchmark tests on cash usage were discussed in [30]. Research on devoted method for skewed strings was presented in [31]. Also, some propositions of adaptive methods were introduced [32]. Tests on possible variations in sorting rules were also discussed in [18]. In [33] it was discussed enhanced information management for big data systems, while in [34] discussion of parallel approaches to sorting for big data systems were proposed. Markov chain rules for adaptive self-sorting were discussed in [35]. In [36] was presented a proposition of the comparison free method for sorting. Also, other sorting methods are permanently developed. Research on the insertion sort method was presented in [37], where instead of classic version was proposed bidirectional insertion method of sorting.

Related Works

Our research on faster and more efficient methods for sorting was started with a proposition for a new method of assigning divided strings for quick sort. In [38] we have shown that the dynamic assignment of position in quick sort can improve sorting by speeding up the method of about 10%. Also, this change prevents deadlocks so often visible for a classic version. The results of the improved composition of the heap structure were presented in [39]. Proposed change of the alignment of the levels in the heap speed up the method of about 5% to 10%. First results of improved merge sort we presented in [40]. We have shown that the dynamic application of "divide and conquer" rule to various parts of the initial string can speed up the process of about 10%. Further research on derivatives of merge sort were discussed in [41]. We have proved that non-recursive version of sorting is more flexible for the implementations on various architectures. Results of our research were examined on Hadoop architectures [42].

In this article I would like to propose a derivative of the newly presented method. In [43] was described a non-recursive fast sort algorithm. This method was proved to have faster sorting from merge sort of about 10% to 15%. Fast sort algorithm was composed as a new method for large data sets. In this work I would like to present application of some idea from fast sort to parallel merge sort algorithm. The novelty of this approach introduces separation of concerns for implementation of sorting on multi core architectures. Some introduction to this idea was presented in [18]. However that was only theoretical proposition of the division of tasks between various processors by the use of binary trees. This article presents practical separation of concerns for parallel merge sort algorithm. In [44] it was discussed how to possibly parallelize the merge sort in a classic version, however that proposition was given for double strings. Proposed in this article a parallelized version of the modified

merge sort algorithm is developed for the dynamically assigned processors. Therefore, discussed here are improvements to make the algorithm scalable to various multiprocessor architectures.

The research presented in this algorithm benefits from a model composed for parallel sorting that is practically realized in C# MS Visual 2015 on MS Windows Server 2012. Proposed in this work algorithm allows the sorting for n elements in time 2n-$\log_2$n-2 using n processors. To analyze the time complexity of this parallel algorithm was used a model of Parallel Random Access Machine (PRAM) that allows an access to read and write in the memory cell for only a single processor. In the same way as in [43], tasks will be divided so that each processor will perform operations on the allocated memory in the most efficient way.

2. Data Processing in NoSQL Database and Parallel Sort Algorithms

Big data describe a wide range of data collected in computer networks. Despite the fact that these data have their own unique features, for the processing is required their unification through the corresponding collecting pipe. Standard data is stored in the collections of data in the form of records of fixed or variable length, but the analysis of the data set is stored in the form of columns. In memory, columnar data is fragmented into smaller units distributed among participating cores, so that parallelization is possible when running a query on the overall data, see Figure 1. Modified parallelization of merge sort algorithm allows to speed up the process of organizing large sets of data. To check the acceleration of this process when using multiple processors, some tests have been made with the measurement of the Central Processing Unit (CPU) clock cycles (clock rate) and sorting time. The tests allows for the evaluation of the efficiency of the process.

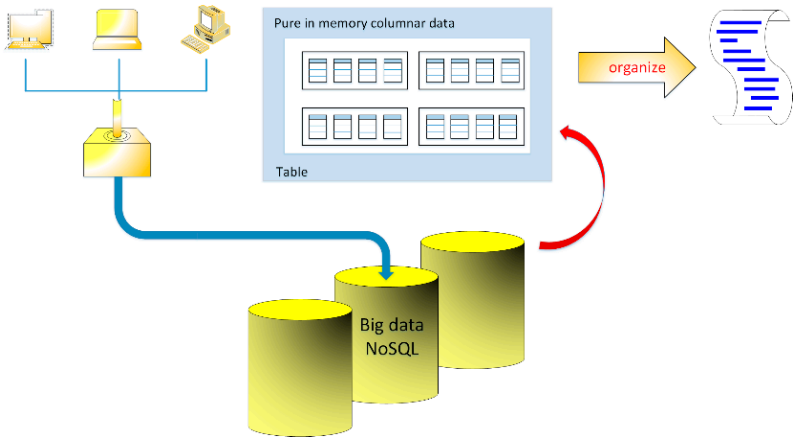

Figure 1. Sample sketch of data processing in NoSQL databases.

Statistical Approach to the Research on Algorithm Performance

For the statistical tests on the performance of this parallel method have been used measures similar to other works [38–41,43]. The arithmetic mean of all of the observed measures for CPU clock, and sorting time can help to estimate performance. Statistically, this measure is equal to the mean value:

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

the standard deviation is defined by the formula:

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

where n is the number of measurements $x_1, x_1, \ldots, x_n$, $\overline{x}$ is the arithmetic mean of the sample. The analysis for sorting time and CPU clock was carried out in 100 benchmark tests for each of the fixed dimensions on the input. The algorithm's stability in a statistical sense is best described on the basis of the coefficient of variation. The coefficient of variation is a measure that allows the determination of the value of diversity in the research. It is determined by the formula:

$$V = \frac{\sigma}{\overline{x}}, \tag{3}$$

where we use arithmetic mean (1) and standard deviation (2). The coefficient of variation reflects the stability of the method in a statistical sense. The study was performed on a collection of data containing from 100 elements up to 100 million elements, increasing the number of elements ten times each new comparison. The results are presented in figures and discussed in the following sections.

3. Parallel Modified Merge Sort Algorithm

Big data require algorithms with low computational complexity with the possibility of division of tasks between multi processors. A special role is played by the sort algorithms used to categorize information. The most frequently used algorithm to sort data in NoSQL databases is merge sort.

The primary issue in NoSQL databases is the ability to quickly organize information, which is necessary for the analysis of the information collected and the compilation of relevant reports. Custom information obtained from equal sources is stored in the form of records of fixed or variable length. Here arises a fundamental problem of data verification, for example, duplication of the same records and the removal of the duplicates. Figure 2 presents a simplified method of collecting information on the disk and searching for the required records. Information entered by the user is subjected to initial verification, based on the already ordered and saved records. To search for the desired information by the user, the records are sorted using a parallel-modified merge sort. Then, proposed modified division search for the location of wanted records and the preparation of a specific report is performed. Let us look at proposed modified merge algorithm for parallel processing. The proposed modification will be that at each step of sorting, we will merge four sorted strings into one ordered sequence of elements, see Figures 3–5.

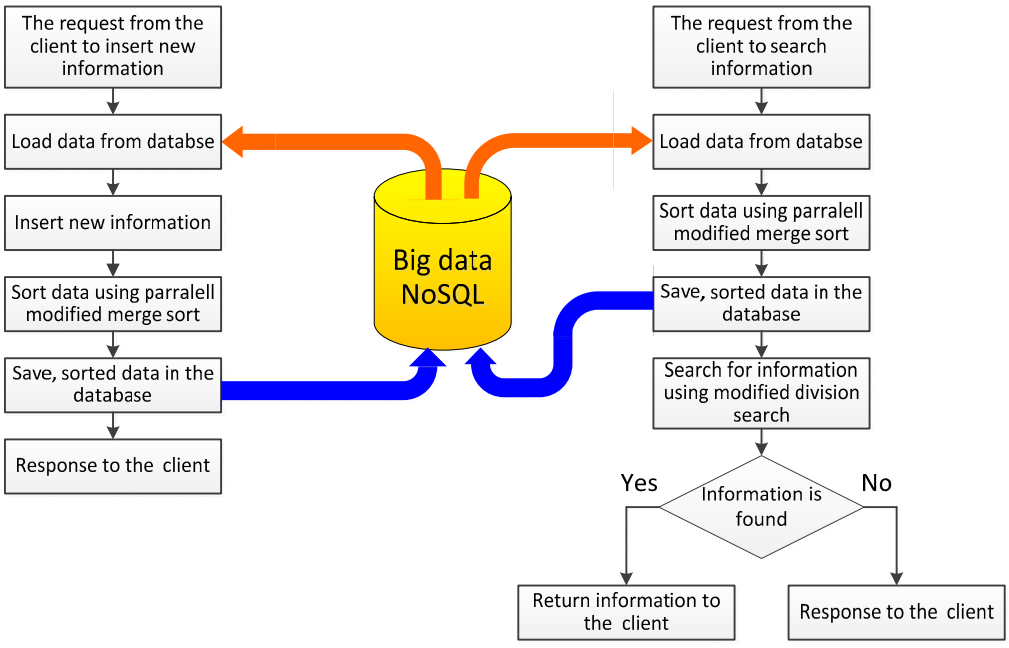

Figure 2. Sample presentation of parallel processing of the request between the user and the server in the NoSQL data bases.

To merge four strings, we use logic indexation of processors. The first processor performs the merge of the first two strings and save the result in temporary array. The index of merged string shall be the same as the index of the first element of the first string. A second processor at the same time starts merging of the third and fourth string. In the same way as the first processor, the second processor will keep the merged sequence of numbers in the temporary array beginning writing at the index of the first element of the third string. All of the processors operate independently from each other and do not share the same memory resources. Separation of concerns for the execution of the parallel merge is established by the use of end-of-cycle assumption located in the parallel for loop. The duration of the process can be defined by the formula:

$$T_{\max} = \max_{0 \le i < p} T_i, \tag{4}$$

where T_i is the sorting time of the i-th processor, and p is the number of processors participating in the parallel merge.

An example of the process in the first stage of the first step of sorting is shown in Figure 3. In this step, the merge use $n/2$ processors for parallel execution of sorting task.

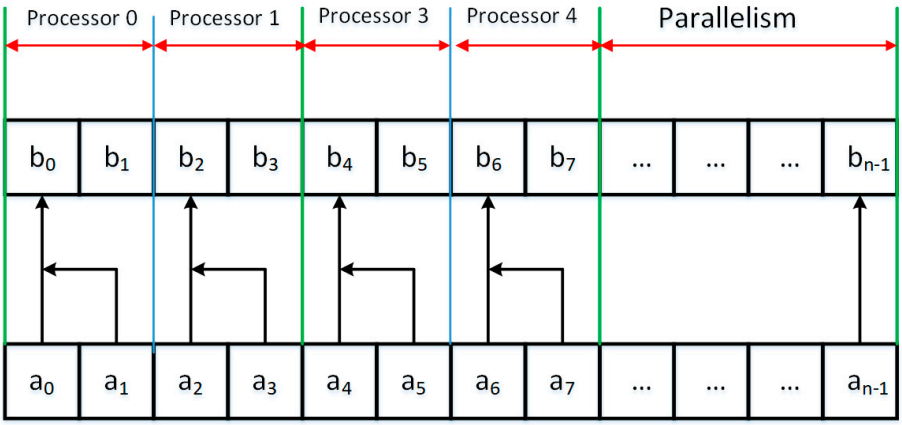

Figure 3. Parallel merge of the first two numeric strings in the first step of modified merge sort.

In the next step of the algorithm, stored in the temporary array information is sorted for every two rows and stored again as the merged row in the input array. Parallelization of the process of merging $n/4$ strings is shown in Figure 4. In the next steps of the algorithm we merge in the same way all of the enlarged strings, in each iteration, four times, see Figure 5.

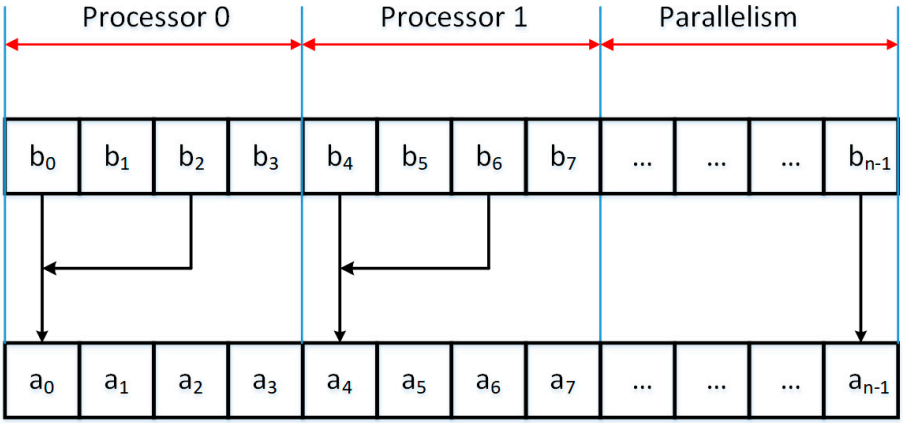

Figure 4. Parallel merge of the two string located in the temporary array.

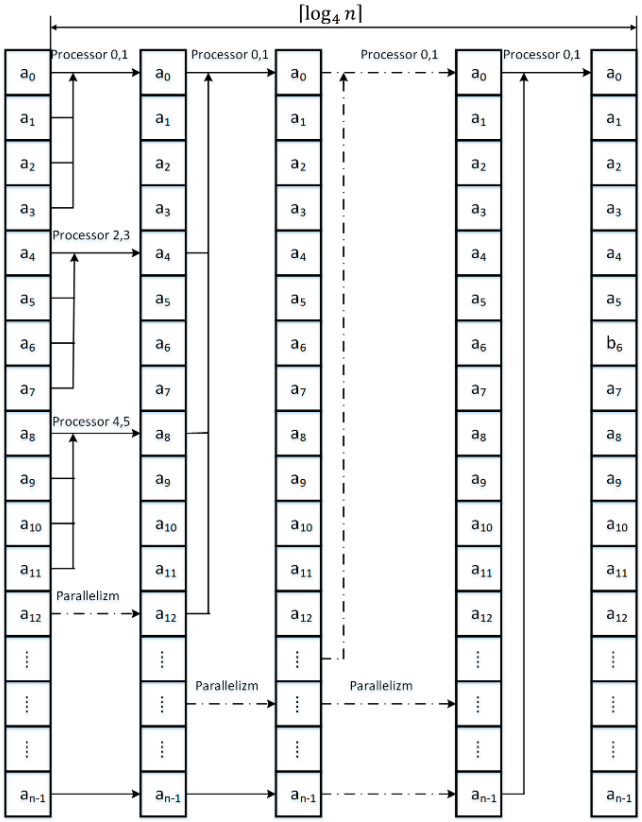

Figure 5. Sample sketch of data processing in NoSQL databases.

Teorem 1. *Parallel Modified Merge Sort Algorithm using n/2 processors has time complexity.*

$$T_{max} = 2n - \log_2 n - 2, \tag{5}$$

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

Let us first notice that the tree sequences $x_1 \leq \ldots \leq x_t$ and $y_1 \leq \ldots \leq y_t$ of t elements, can be merged into one sequence $z_1 \leq \ldots \leq z_{2t}$, using merge algorithm, making no more than $2t - 1$ comparison of the elements of sequences X and Y.

In the first iteration $t = 1$ the first $n/2$ processors perform in a concurrent merge two one element strings by doing no more than $2 \cdot 1 - 1 = 1$ comparisons for each processor. In fact, the time to complete the entire operation is such as the duration of sorting for one processor. Then $n/4$ processors perform in a concurrent merge two one element strings by doing no more than $2 \cdot 2 - 1 = 3$ comparisons for each processor.

Let's now think about the maximum time-of-operation formula, which can be derived by summing the maximum running time of each iteration of the sorting method. In each iteration t, in the first step $n/2^{2t-1}$, and in the second step $n/2^{2t}$, processors perform the integration of four strings 4^{t-1} elements by doing no more than:

$$\left(2 \cdot 2^{2t-2} - 1\right) + \left(2 \cdot 2^{2t-1} - 1\right), \tag{6}$$

comparisons. All operations performed in a simultaneous way we can save in the form of

$$\begin{aligned} \sum_{t=1}^{k} \left[\left(2 \cdot 2^{2t-2} - 1\right) + \left(2 \cdot 2^{2t-1} - 1\right)\right] &= 2 \sum_{t=1}^{k} \left(2^{2k-2} + 2^{2k-1}\right) - 2k \\ = 2\left(1 + 2^1 + 2^2 + \ldots + 2^{2k-1}\right) - 2k &= 2\left(2^{2k} - 1\right) - 2k \end{aligned}$$

what gives that $2k = 2\log_4 n = 2\frac{\log_2 n}{\log_2 4} = \log_2 n$ and $2^{2k} = 4^k = n$, therefore we get:

$$T_{max} = 2n - \log_2 n - 2, \tag{7}$$

which was to prove.

Modified parallel merge algorithm was implemented in C# MS Visual 2015 on MS Windows Server 2012. In the implementation of the algorithm was used the class System.Threading.Tasks, giving the possibility of implementing parallel loops. The parallel loop automatically assigns tasks to the subsequent processors and stops all tasks to complete the loop. It should be remembered that it is not the usual iterative loop, and the indicator of iteration number of the selected processor is performed in the other way. The algorithm was designed to iteratively allocate information to processors by the implemented modified merge. In Algorithm 1 and 2 we can see the code of the methods, while in Figures 6 and 7 we can see block diagrams to explain proposed implementation.

Algorithm 1. Parallelized Modified Merge Sort Algorithm

```
Start
Load table a
Load dimension of table a into n
Create an array of b of dimension n
Start
Load table a
Load dimension of table a into n
Create an array of b of dimension n
Set options for parallelism to use all
processors of the system
Remember 1 in m
While m is less than n then do
Begin
    Remember 2*m in m2
    Remember 4*m in m4
    Remember (n-1)/m2 in it1
    Parallel for each processor at index j greater
    or equal 0 and less than it1 + 1 do
    Begin parallel for
        Remember j*m2 in i
        Remember i in p1
        Remember i+m in p2
        If p2 greater than n then do
        Begin
            Remember n in p2
        End
        Remember n-p1 in c1
        If c1 greater than m then do
        Begin
            Remember m in c1
        End
        Remember n-p2 in c2
        If c2 greater than m then do
        Begin
            Remember m in c2
        End
```

```
        Proceed function the merge algorithm of
        two sorting string merging of array a and
        write in the array b
    End of the parallel for
    Remember (n-1)/m4 in it3
    Parallel for each processor at index j greater
    or equal 0 and less than it3 + 1 do
    Begin parallel for
        Remember j*m4 in i
        Remember i in p1
        Remember i+m2 in p2
        If p2 greater than n then do
        Begin
            Remember n in p2
        End
        Remember n-p1 in c1
        If c1 greater than m2 then do
        Begin
            Remember m2 in c1
        End
        Remember n-p2 in c2
        If c2 greater than m2 then do
        Begin
            Remember m2 in c2
        End
        Proceed function the merge algorithm of
        two sorting string merging of array b and
        write in the array a
    End of the parallel for
Multiply variable m by four
End
Stop
```

Algorithm 2. The Merge Algorithm of Two Sorted Strings

```
Start
Load table a
Load table b
Load index p1
Load variable c1
Load index p2
Load variable c2
Remember p1 in pb
While c1 greater than 0
 and c2 greater than 0 then do
Begin
    If a[p1] less or equal a[p2] then do
     Begin
          Remember a[p1] in b[pb]
          Add to index p1 one
          Add to index pb one
          Subtract from variable c1 one
     End
     Else
```

```
            Remember a[p2] in b[pb]
            Add to index p2 one
            Add to index pb one
            Subtract from variable c2 one
        End
    End
    While c1 greater than 0 then do
    Begin
        Remember a[p1] in b[pb]
        Add to index p1 one
         Add to index pb one
        Subtract from variable c1 one
    End
    While c2 greater than 0 then do
    Begin
        Remember a[p2] in b[pb]
        Add to index p2 one
        Add to index pb one
        Subtract from variable c2 one
    End
    Stop
```

For this article a modified algorithm was used for merging two strings. The proposed modification is that the merged strings are treated as stacks of ordered numbers from the lowest to the highest number, see Figure 6. The algorithm compares the numbers on the stacks when rewriting a lower number to a merged sequence. In the case of equal numbers, the implementation is using the number of the first stack. After emptying one of the stacks it overwrites the number of the stack containing elements.

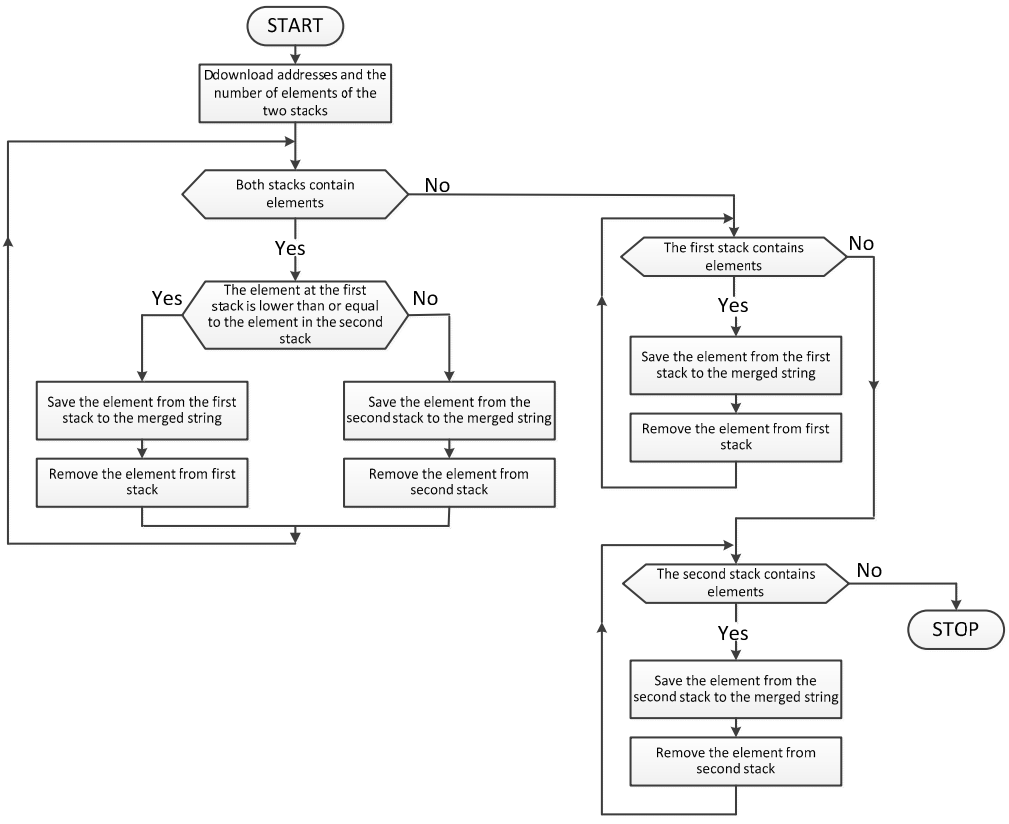

Figure 6. Sample block diagram of the proposed algorithm to merge two sorted strings.

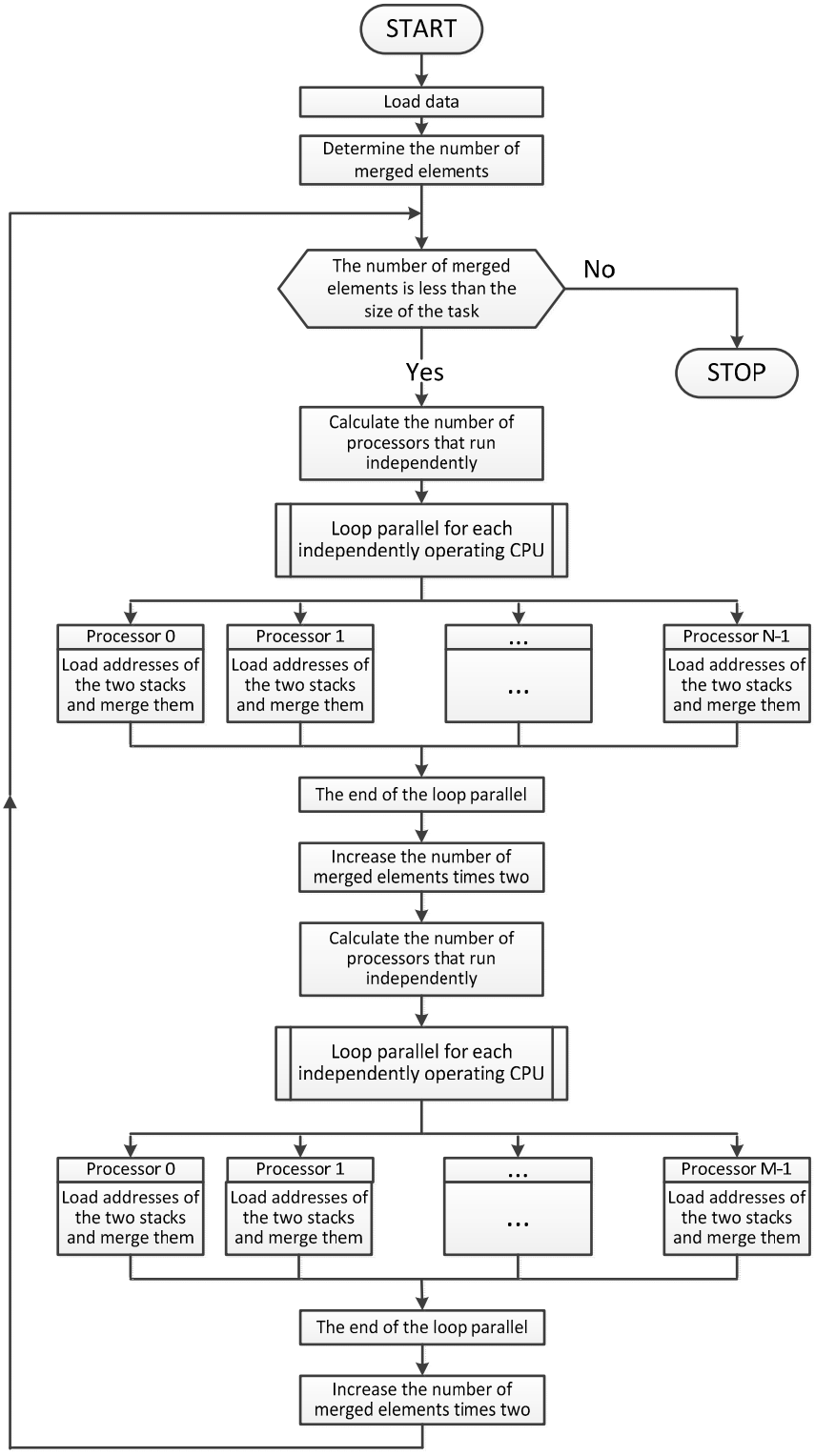

Figure 7. Sample block diagram of the proposed parallel sorting procedure.

The proposed parallel sorting algorithm is using merge in each step. The four strings of numbers are merged into one ordered string of numbers, see Figure 7. Each step of the algorithm is divided into two stages. First, in the iteration is merge of all subsequent pairs of numeric strings to the temporary array. Then, there is the repetition of the merging of all subsequent pairs of numeric strings in the input array. The merge process has been parallelized in every step in such a way that the first stage of

the merge is used to write information to the temporary array, from which it will be executed by the specified number of processors.

Also, the second stage may be performed simultaneously for a fixed number of processors. Proposed paralleled merge processes four numeric strings to use more CPUs than simple selection algorithm. The largest element of four strings is simply rewritten to output string. The practical parallel calculation process uses a loop Parallel For in C# Visual Studio. Which is assigning iteratively tasks for further processors. In the implemented algorithm available processors are numbered from zero to the total number of allocated processors minus one. In this way, each processor can calculate indexes from all the stacks and the number of elements on the stacks.

4. The Study of the Parallelized Modified Merge Sort

A comparative study of the speed of the presented method was performed on MS Windows Server 2012 with Opteron Advanced Micro Devices (AMD). Processor 8356 8p produced by AMD, Inc. The algorithm was implemented in C# in Visual Studio Ultimate 2015. Statistical analysis for 100 samples generated randomly was conducted for each dimension of input, starting from the input set of 100 elements and increasing the size of the task 10 times each test to 100 million elements. Among the examined layouts of the elements were various combinations. Similarly to [9] were examined permutations of the random elements of the integers from 1 to n, randomized selections of the elements from 1 to sqrt(n), decreasing strings of integers from n to 1, increasing strings of integers 1 to n, same n equal integers in a string, but also special layouts that are hard to sort. A deeper analysis on these special examples of critical strings for quick sort was presented and discussed in our previous research [38].

Each sorting operation by an examined method was measured in time [ms] and CPU usage represented in tics [ti] of the CPU clock. These results are averaged for 100 sorting samples. Benchmark comparison is described in Tables 1 and 2, Figures 8 and 9. In the tables are presented only mean times. These comparison show the analysis of the sorting time. Worst Case Execution Time (WCET) analysis is not necessary since the given and proved theorem clearly show WCET, which is when all the comparison instructions are done for the sequence of integers sorted inversely. The proposed method does not suffer from any critical strings as quick sort [38]. The best sorting time will be for the ordered sequence. The method will then carry out only half of the comparisons of those that are performed in the worst case. This is due to the proposed implementation of the algorithm that merges two numbers. At maximum, the proposed algorithm performs $2n$-1 comparisons for merging two n element strings, but must have at least n comparisons to merge them. Therefore, we have the upper and lower limit of the sorting time of the algorithm, and in this case, it is justified to perform statistical surveys on the mean time analysis, as to show how the algorithm behaves in the meantime.

Table 1. The results of parallelized modified merge sort in [ms].

Method—Average Time Sorting for 100 Samples in [ms]				
Elements	**1—Processor**	**2—Processors**	**4—Processors**	**8—Processors**
100	1	1	1	1
1000	1	1	1	1
10,000	6	4	3	3
100,000	46	27	20	17
1,000,000	499	287	203	173
10,000,000	5745	3256	2317	1938
100,000,000	65,730	37,542	26,382	23,186

Table 2. The results of parallelized modified merge sort in [ti].

Method—Average Time Sorting for 100 Samples in [ti]				
Elements	**1—Processor**	**2—Processors**	**4—Processors**	**8—Processors**
100	362	324	264	256
1000	757	647	524	510
10,000	13,672	7993	5490	4647
100,000	72,291	41,473	31,437	25,748
1,000,000	777,903	437,706	315,924	269,424
10,000,000	8,954,448	5,230,931	3,798,070	3,238,545
100,000,000	102,449,603	58,073,015	41,119,351	35,138,937

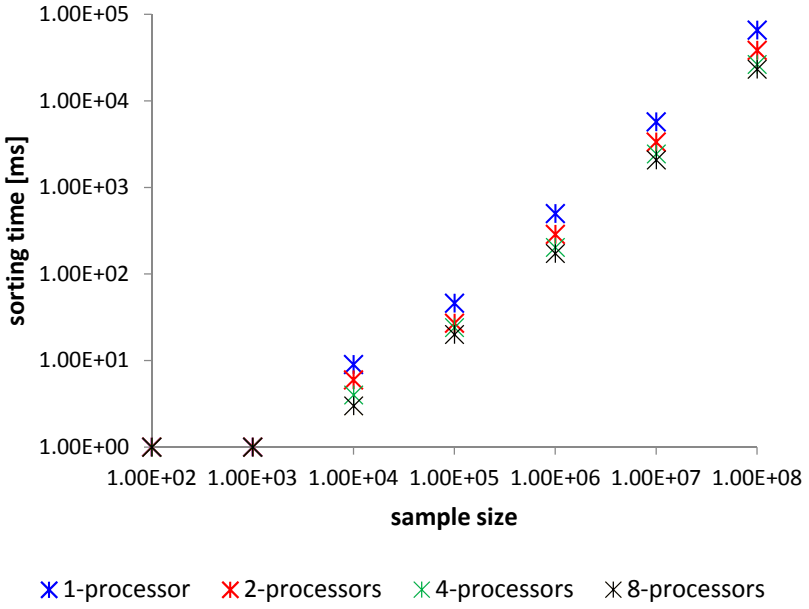

Figure 8. Comparison of benchmark time [ms].

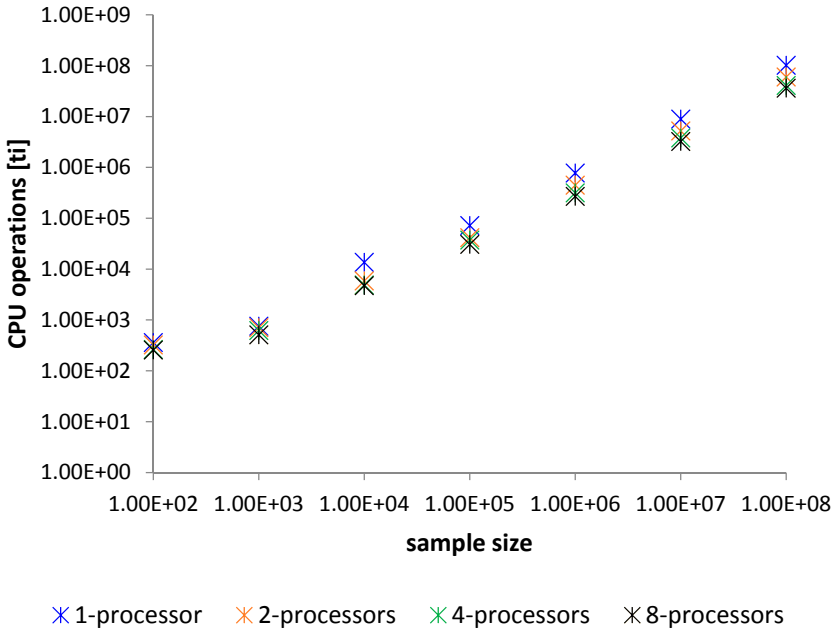

Figure 9. Comparison of benchmark processing [ti].

Analyzing results we can see that each new processor can give some additional power of computing. Most visible changes in operations are between 2 and 4 processors. With the usage of each new processor, the number of operations in the system is lower. Similarly, the operation time is shorter. These confirm the theoretical assumptions proven in Theorem I.

Comparison of coefficient of variation for parallelized modified merge sort is presented in Tables 3 and 4.

Table 3. Comparison of the coefficient of variation for [ms].

Coefficient of Variation [ms]				
Elements	**1—Processor**	**2—Processors**	**4—Processors**	**8—Processors**
100	0.3821615	0.43268856	0.42831802	0.44107100
1000	0.3643562	0.40356701	0.41257718	0.36173402
10,000	0.2743689	0.36596252	0.21821789	0.34069257
100,000	0.1661708	0.17707090	0.17747680	0.14015297
1,000,000	0.1919563	0.21309771	0.14367983	0.15528751
10,000,000	0.2029680	0.21077261	0.16998284	0.17756993
100,000,000	0.2089429	0.20919211	0.20167919	0.16278364

Table 4. Comparison of the coefficient of variation for [ti].

Coefficient of Variation [ti]				
Elements	**1—Processor**	**2—Processors**	**4—Processors**	**8—Processors**
100	0.40505154	0.26224710	0.15192519	0.17000328
1000	0.36725132	0.28016626	0.20075293	0.254527489
10,000	0.28111613	0.31917443	0.23101516	0.212133586
100,000	0.16516246	0.17583523	0.12684148	0.132691392
1,000,000	0.19185991	0.21235997	0.15981887	0.155664995
10,000,000	0.20295364	0.21077453	0.16344274	0.177647028
100,000,000	0.20894325	0.20918928	0.15855142	0.162786764

Studies have shown the statistical stability of the sort algorithm for large data collections. Some changes in the coefficients for small size inputs stemmed from the fact that the system automatically exceed operations what caused longer sorting time in this cardinality.

4.1. Comparison and Analysis

Time analysis of sorting is an important element in identifying the effectiveness of the methods of sorting large data sets and NoSQL data bases. Let us compare the algorithm. For this we can assume that the duration of the method will be compared in a respect to one processor. Let us examine if the duration of sorting is shorter for the method using multiple processors. The results are shown in Figures 10 and 11.

We can see that with the introduction of each new processing unit it is possible to efficiently increase the performance of the method. Just two processors can shorten the time of sorting of about 20% to 30%. Each new processor gives additional possibility to decrease sorting time. This can be very important for multi-core architecture with several processors. This result is very important for large data sets.

If we compare the usage of processing power similar conclusions can be drawn. Just two processors can shorten time of sorting by about 20% to 30%. Each new processor gives additional possibility to divide tasks and therefore speed up the sorting. In Figure 12 is a comparison of sorting times for various methods. The results are presented for a quick sort presented in [38], heap sort presented in [39], and merge sort presented in [40]. As a version for comparison was selected proposed

parallel modified merge sort on 8 processors (PMMS8). All the methods were presented in relation to the heap sort. The difference in presented in percentage change of sorting time in [ms].

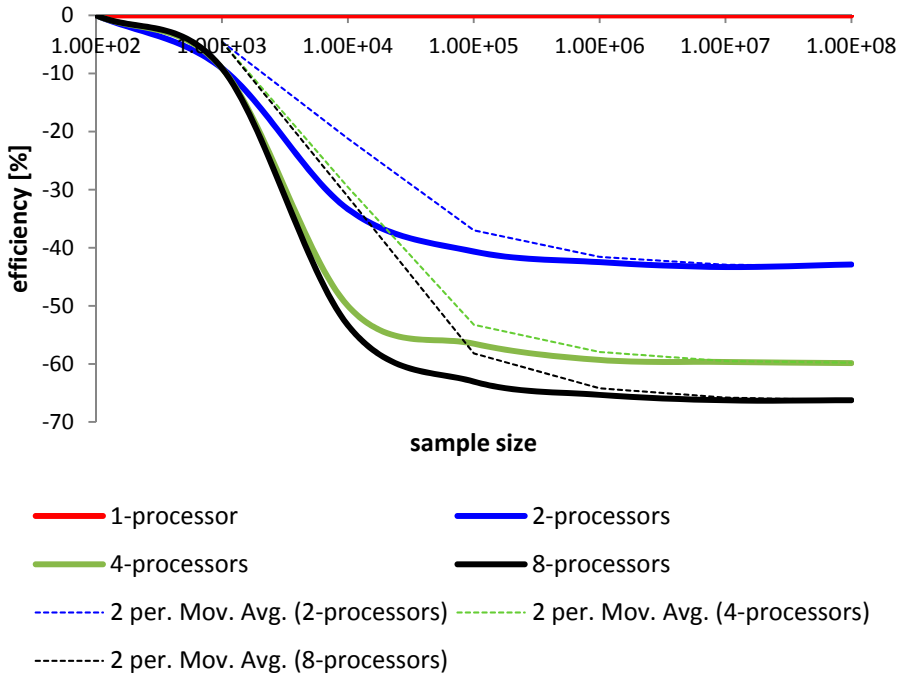

Figure 10. Comparison of the method using multiple processors in terms of operational time [ms].

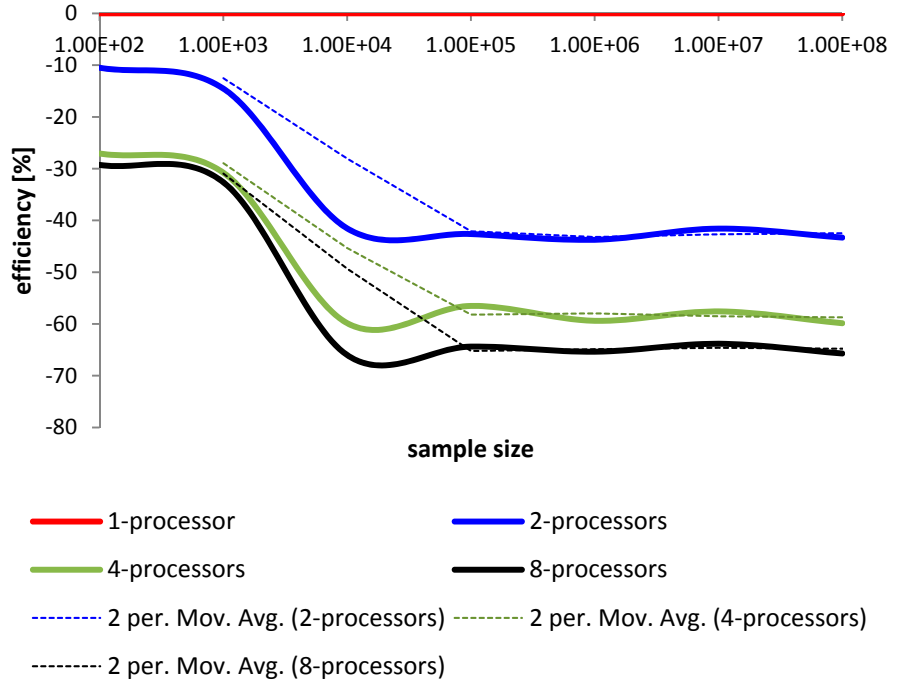

Figure 11. Comparison of the method using multiple processors in terms of operational time [ti].

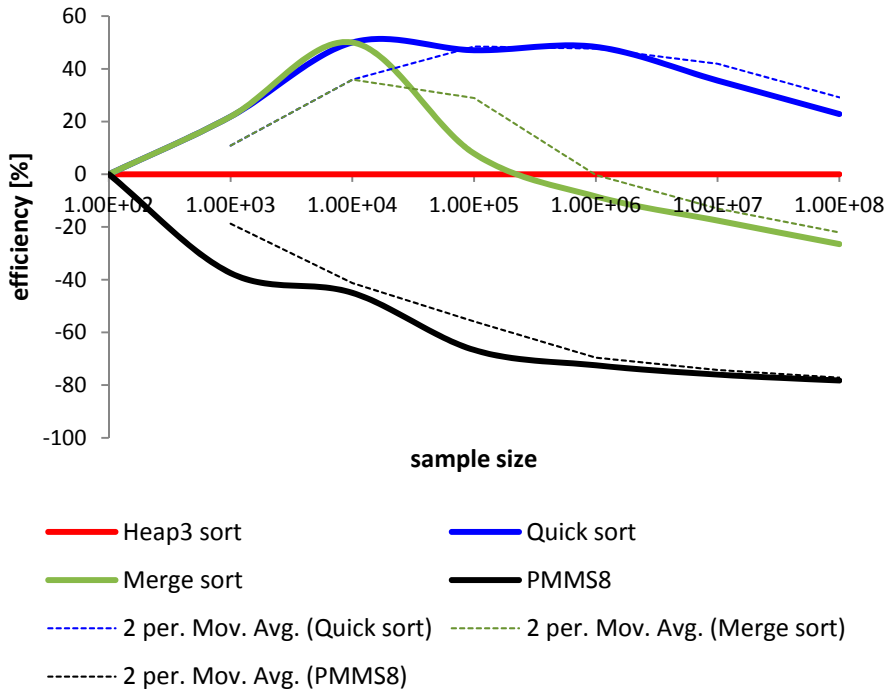

Figure 12. Comparison of sorting time for heap sort, quick sort, merge sort and proposed in this article parallel modified merge sort on 8 processors.

Comparing results from Figure 12 we can see that the proposed PMMS8 is much more efficient for sorting large data sets than other methods. The results have shown that quick sort algorithm is the less efficient among examined algorithms. An additional disadvantage of using the quick sort algorithm is the possibility of deadlocks, as shown in [38]. Merge sort, however of up to 100,000 elements sorts inputs in a very similar way to quick sort, above this number becomes more efficient and above 1,000,000 elements is faster than heap sort algorithm. Additional merge sort does not have deadlocks, which is what makes this method more efficient. Proposed in this article is that a parallel version of merge sort is the most efficient algorithm. The results have shown that this algorithm can sort large collections of data about 40% faster if we use just 8 processors. An additional advantage is that there is no possibility of deadlocks, as this method comes as a parallel derivative from the modified merge presented in [40]. We theoretically estimate the results for additional processors, from which we can conclude the bounds to execution time with each new processor. Sorting time for one processor is $T_{max} = n \log_2 n$. In the theorem we have proved that the maximum for $n/2$ processors is $T_{max} = 2n - \log_2 n - 2$. Therefore, we are not able to decrease sorting time freely with each new processor. There is always a question of the computing power of the machine used for sorting operations.

4.2. Conclusions

Studies have shown the effectiveness of the presented method for large data sets. Reductions of sorting time are clearly visible in the yield from under ten thousand elements, making it easier to sort the data sets in NoSQL databases. An additional advantage of the proposed method is no deadlocks, as presented in this article method is a parallel version of modified merge sort for which it was proved to have no deadlocks. Introduction of any additional processor gives a big advantage. As we have seen from the results, each additional processor can boost the method, which is what is very important for large data sets. Since in the proposed application was used, a separation of concerns the method can be implemented on architectures with multiple cores. Therefore, its practical efficiency will be visible for cloud computing, big data sets, and NoSQL system, etc.

Presented results from the research show that with new processors the algorithm performs better. Compared to other sorting methods, the proposed PMMS from 0% to 80% faster in each extended cardinality. Unfortunately, we are not able to decrease sorting time freely. There is always a question about the performance of the machine. Also, another important matter is the operating system, programming studio, compiler, and programming language itself. These of all matters influence the algorithm. In the research, MS Windows Server 2012 was used with Opteron AMD Processor 8356 8p, and the PMMS algorithm was implemented in C# in Visual Studio Ultimate 2015. These software issues were chosen as most commonly used at the market. Computer architecture used for the research was the most powerful, as it was within the author's research capabilities.

In the proposed analysis, creating a probabilistic model and counting the regression curve was not performed for a special reason. Because the division of the information into aggregated blocks and allocations of processors is predetermined and does not subjected to any changes during the operation of the algorithm, we do not have to discuss a classic statistical analysis of the research. In our case presented discussion it is enough to draw conclusions. The proposed method is independent of input cast, unlike in the quick sort method for which deeper statistical analysis is necessary due to the various deadlocks and sequences that are hard to sort. In our case, as proven in the theorem, for any number of input sequences and available number of processors there is a top and bottom estimation of the algorithm's running time. This is confirmed by the study as the coefficient of variation stabilizes and is about 20%, which is in line with the theory of computational complexity for the numerical sequences algorithm.

5. Final Remarks

The article presents Parallelized Modified Merge Sort algorithm for rapid sorting of large data sets. A proposed method is based on a model of PRAM (Parallel Random Access Machine) that allows the efficient access to read and write information in the memory cell for each single processor. Therefore, the implementation was done using a separation of concerns to increase the efficiency and enable the dynamic association of processors to performed operations. The method was implemented for a parallel processing of information on machines with many processors. Practical realization of the PMMS was done in C# MS Visual 2015 on MS Windows Server 2012.

In the article was presented a theoretical analysis of the efficiency and practical verification. The proposed algorithm was proved to sort n elements in the maximum time $2n-\log_2 n-2$ using n processors. Comparison tests have shown that the method is more efficient than other sorting methods, especially for big data sets. In the study, it was shown that the statistical stability of the proposed method is on a very good level. The results of benchmark tests confirmed theoretical computational complexity. Presented parallelized sorting algorithm can be successfully used in database applications, especially in situations where a number of processors can be used for speeding up the sorting process.

In the future research is planned a further development in sorting performance. The research will involve the parallelization of the developed versions of heap sort and quick sort algorithms.

Acknowledgments: The author would like to acknowledgment contribution to this research from the Rector of Silesian University of Technology under grant RGH 2017 for prospective professors, which was received for covering the costs of this publication in open access.

Author Contributions: Zbigniew Marszałek designed the method, performed experiments and wrote the paper.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Aho, I.; Hopcroft, J.; Ullman, J. *The Design and Analysis of Computer Algorithms*; Addison-Wesley Publishing Company: Boston, MA, USA, 1975.
2. Knuth, D. *The Art of Computer Programming Vol.3: Sorting and Searching*; Addison-Wesley: Boston, MA, USA, 1998.

3. Bing-Chao, H.; Knuth, D. A one-way, stack less quick sort algorithm. *BIT* **1986**, *26*, 127–130. [CrossRef]
4. Francis, R.; Pannan, L. A parallel partition for enhanced parallel quick sort. *Parallel Comput.* **1992**, *18*, 543–550. [CrossRef]
5. Rauh, A.; Arce, G. A fast weighted median algorithm based on quick select. In Proceedings of the IEEE 17th International Conference on Image Processing, Hong Kong, China, 26–29 September 2010; pp. 105–108.
6. Tsigas, P.; Zhang, Y. A simple, fast parallel implementation of quick sort and its performance evaluation on SUN enterprise 10,000. In Proceedings of the Euromicro Workshop on Parallel, Distributed and Network-Based Processing, Genova, Italy, 5–7 February 2003; pp. 372–381.
7. Daoud, A.; Abdel-Jaber, H.; Ababneh, J. Efficient non-quadratic quick sort (NQQuickSort). In Proceedings of the International Conference on Digital Enterprise and Information Systems, London, UK, 20–22 July 2011.
8. Edmondson, J. Pivot sort—Replacing quick sort. In Proceedings of the 2005 International Conference on Algorithmic Mathematics and Computer Science, Las Vegas, NV, USA, 20–23 June 2005; pp. 47–53.
9. Kushagra, S.; López-Ortiz, A.; Munro, J.; Qiao, A. Multi-pivot quicksort: Theory and experiments. In Proceedings of the Meeting on Algorithm Engineering & Experiments. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 10–12 January 2016; pp. 47–60.
10. Ben-Or, M. Lower bounds for algebraic computation trees. In Proceedings of the 15th ACM Symposium on Theory of Computing (STOC), Boston, MA, USA, 25–27 April 1983; pp. 80–86.
11. Doberkat, E. Inserting a new element into a heap. *BIT Numer. Math.* **1983**, *21*, 255–269. [CrossRef]
12. Lutz, M.; Wegner, L.; Teuhola, J. The external heap sort. *IEEE Trans. Softw. Eng.* **1989**, *15*, 917–925.
13. Sumathi, S.; Prasad, A.; Suma, V. 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), Bhubaneswar, India, 14–15 November 2014; pp. 693–700.
14. Roura, S. Digital access to comparison-based tree data structures and algorithms. *J. Algorithms* **2001**, *40*, 123–133. [CrossRef]
15. Abrahamson, K.; Dadoun, N.; Kirkpatrick, D.; Przytycka, T. A simple parallel tree construction algorithm. *J. Algorithms* **1987**, *10*, 287–302. [CrossRef]
16. Carlsson, S.; Levcopoulos, C.; Petersson, O. Sublinear merging and natural merge sort. *Algorithms* **1990**, *450*, 251–260. [CrossRef]
17. Cole, R. Parallel merge sort. *SIAM J. Comput.* **1988**, *17*, 770–785. [CrossRef]
18. Gedigaa, G.; Duntschb, I. Approximation quality for sorting rules. *Comput. Stat. Data Anal.* **2002**, *40*, 499–526. [CrossRef]
19. Gubias, L. Sorting unsorted and partially sorted lists using the natural merge sort. *Softw. Pract. Exp.* **2006**, *11*, 1339–1340. [CrossRef]
20. Huang, B.; Langston, M. Merging sorted runs using main memory. *Acta Inform.* **1989**, *27*, 195–215. [CrossRef]
21. Huang, B.; Langston, M. Practical in-place merging. *Commun. ACM* **1989**, *31*, 348–352. [CrossRef]
22. Zhang, W.; Larson, P. Speeding up external merge sort. *IEEE Trans. Knowl. Data Eng.* **1996**, *8*, 322–332. [CrossRef]
23. Zhang, W.; Larson, P. Dynamic memory adjustment for external merge sort. In Proceedings of the Very Large Data Bases Conference, San Francisco, CA, USA, 25–29 August 1997; pp. 376–385.
24. Zhang, W.; Larson, P. Buffering and read-ahead strategies for external merge sort. In Proceedings of the Very Large Data Bases Conference, New York, NY, USA, 24–27 August 1998; pp. 523–533.
25. Vignesh, R.; Pradhan, T. Merge sort enhanced in place sorting algorithm. In Proceedings of the 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), Ramanathapuram, India, 25–27 May 2016; pp. 698–704.
26. Cheema, S.; Sarwar, N.; Yousaf, F. Contrastive analysis of bubble & merge sort proposing hybrid approach. In Proceedings of the 2016 International Conference on Innovative Computing Technology (INTECH), Dublin, Ireland, 24–26 August 2016; pp. 371–375.
27. Smita, P.; Sourabh, C.; Safikul, A.S. Enhanced merge sort—A new approach to the merging process. *Procedia Comput. Sci.* **2016**, *93*, 982–987.
28. Alanko, T.; Erkio, H.; Haikala, I. Virtual memory behavior of some sorting algorithm. *IEEE Trans. Softw. Eng.* **1984**, *10*, 422–431. [CrossRef]

29. Larson, P.-Å.; Graefe, G. Memory management during run generation in External Sorting. In Proceedings of the SIGMOD, Seattle, WA, USA, 2–4 June 1998; pp. 472–483.
30. LaMarca, A.; Ladner, R. The influence of caches on the performance of sorting. *J. Algorithms* **1999**, *31*, 66–104. [CrossRef]
31. Crescenzi, P.; Grossi, R.; Italiano, G.F. Search data structures for skewed strings. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2647, pp. 81–96.
32. Estivill-Castro, V.; Wood, D. A survey of adaptive sorting algorithms. *Comput. Surv.* **1992**, *24*, 441–476. [CrossRef]
33. Choi, S.; Seo, J.; Kim, M.; Kang, S.; Han, S. Chrological big data curation: A study on the enhanced information retrieval system. *IEEE Access* **2017**, *5*, 11269–11277. [CrossRef]
34. Axtmann, M.; Bigmann, T.; Schulz, C.; Sanders, P. Practical massively parallel sorting. In Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15), Portland, OR, USA, 13–15 June 2015; pp. 13–23.
35. Shen, Z.; Zhang, X.; Zhang, M.; Li, W.; Yang, D. Self-sorting-based MAC protocol for high-density vehicular Ad Hot networks. *IEEE Access* **2017**, *5*, 7350–7361. [CrossRef]
36. Abdel-Hafeez, S.; Gordon-Ross, A. An efficient O(*N*) comparison-free sorting algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* **2017**, *25*, 1930–1942. [CrossRef]
37. Saher, M.A.; Emrah, A.S.; Celebi, F. Bidirectional conditional insertion sort algorithm; An efficient progress on the classical insertion sort. *Future Gener. Comput. Syst. Int. J. ESci.* **2017**, *71*, 102–112.
38. Woźniak, M.; Marszałek, Z.; Gabryel, M.; Nowicki, R. Preprocessing large data sets by the use of quick sort algorithm. *Adv. Intell. Syst. Comput. KICSS'2013* **2015**, *364*, 111–121. [CrossRef]
39. Woźniak, M.; Marszałek, Z.; Gabryel, M.; Nowicki, R. Triple heap sort algorithm for large data sets. In *Looking into the Future of Creativity and Decision Support Systems*; Skulimowski, A.M.J., Ed.; Progress & Business Publishers: Cracow, Poland, 2015; pp. 657–665.
40. Woźniak, M.; Marszałek, Z.; Gabryel, M.; Nowicki, R. Modified merge sort algorithm for large scale data sets. In Proceedings of the International Conference on Artificial Intelligence and Soft Computing, Zakopane, Poland, 9–13 June 2013; pp. 612–622.
41. Marszałek, Z.; Woźniak, M.; Borowik, G.; Wazirali, R.; Napoli, C.; Pappalardo, G.; Tramontana, E. Benchmark tests on improved merge for big data processing. In Proceedings of the Asia-Pacific Conference on Computer Aided System Engineering 9APCASE'2015), Quito, Ecuador, 14–16 July 2015; pp. 96–101.
42. Czerwiński, D. Digital filter implementation in Hadoop data mining system. In Proceedings of the International Conference on Computer Networks, Brunow, Poland, 16–19 June 2015; pp. 410–420.
43. Marszałek, Z. Novel recursive fast sort algorithm. In Proceedings of the 22nd International Conference on Information and Software Technologies (ICIST), Druskininkai, Lithuania, 13–15 October 2016; pp. 344–355.
44. Uyar, A. Parallel merge sort with double merging. In Proceedings of the 2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT), Astana, Kazakhstan, 15–17 October 2014; pp. 490–494, Book Series: International Conference on Application of Information and Communication Technologies.