Journal of Universal Computer Science, vol. 24, no. 4 (2018), 488-514
submitted: 14/12/17, accepted:15/4/18, appeared: 28/4/18 © J.UCS

Parallel Fast Sort Algorithm for Secure Multiparty Computation

Zbigniew Marszałek
(Silesian University of Technology, Gliwice, Poland
zbigniew.marszalek@polsl.pl)

Abstract: The use of encryption methods such as secure multiparty computation is an important issue in applications. Applications that use encryption of information require special algorithms of sorting data in order to preserve the secrecy of the information. This proposition is composed for parallel architectures. Presented algorithm works with a number of logical processors. Operations are flexibly distributed among them. Therefore sorting of data sets takes less time. Results of the experimental tests confirm the effectiveness of the proposed flexible division of tasks between logical processors and show that this proposition is a valuable method that can find many practical applications in high performance computing.

Keywords: Privacy, parallel algorithm, algorithm design and analysis, data mining, secure multi-party computation
Categories: F.2, E.1, E.4

1 Introduction

Secure multiparty computational has many practical application such as, for example [Lindell, 09]. The privacy policy may prevent the disclosure of information provided and the source of their origin. The technique for hiding data is secure multi-party computation (MPC). One of the important elements of a safe survival in the information system is to sort the data set. In [Bogdanov, 14], [Laud, 16], [Hamada, 12] describe the use sorting algorithms in secure multi-party computation. In this work, it is proposed to use multiprocessor architecture for cryptographic computations. Continues research on computer technology has allowed the design of machines which are operating on multiple cores, where each of them has a number of logical processors working independently. Powerful computers make possible creation of an appropriate software for more efficient processing of information stored in databases. Modern devices allow the use of intelligent software that helps processing information for better data mining. Data managing techniques and information storage systems need an order in the information to increase efficiency of management and to improve performance. In this case sorting methods are very helpful. Some of the classic propositions which are in the roots of computer algorithms were presented in [Aho, 75], [Knuth, 98]. Initially three types of methods were introduced: quick sort, heap sort and merge sort. Over the years we can find the results of important research on performance of these sorting methods, where the use of various computational approaches and improved computer architectures were reported.

Quick sort is using divisions of the data stack for comparison and sorting of the information in smaller portions. We can find many propositions to divide the information in the most efficient way, what speed up sorting and prevent potential deadlocks. This method in a version with devoted pivot mechanism for better data stack management was presented in [Bing-Chao, 86]. In [Francis, 92] were discussed possible variations in partitioning of the data for quick sort. A derivate of quick sort based on calculation of a median value for each stack division was discussed in [Rauh, 10]. Performance tests for SUN machine were presented in [Tsigas, 03].

Heap sort instead of the classic data structure requires composition of the heap, in which the order of the data depends on relations between subsequent levels. Discussion on the efficiency of insertions of the new elements into this structure was presented in [Doberkat, 83]. Propositions of changes between some heap levels were proposed for external versions of this sorting method in [Lutz, 89], [Wegner, 89]. Mathematical properties of heap structures, which are also called trees were discussed in [Ben-Or, 83], [Doberkat, 83]. A discussion of performance of the digital access to the information stored in heap was presented in [Roura, 01] and parallel algorithms for composition of heap were proposed in [Abrahamson, 87].

Merge sort is based on "divide and conquer" assumption, where we divide the input data into smaller strings which are sorted during subsequent merges into one final string. In [Carlsson, 90] was given a proposition of devoted sublinear merging. Parallel version of merge was discussed in [Cole, 88]. In [Gubias, 06] was presented how to use this method for partially sorted lists. Practical tests and implementations of merge sort were discussed in [Huang, 89], while tests on memory usage for sorting by this algorithm were discussed in [Huang, 89]. An idea of external version for reduced input-output operations was presented in [Zhang, 96], and improvements in memory usage by the dynamic assignments were proposed in [Zhang, 97]. A devoted strategies for improved buffering and faster readings from the stack were presented in [Zhang, 98].

From these classic methods various solutions and approaches for databases and information management systems were developed. Tests on various features of virtual memory assignments for sorting were presented in [Alanko, 84], and some interesting strategies for memory management were discussed in [Larson, 98]. Benchmark tests on method cash and its influence on fast sorting were discussed in [LaMarca, 97]. Data type and alignment are also demanding new methods, in [Cole, 88] was given a proposition for skewed strings. A survey of adaptive sorting methods was presented in [Estivill-Castro, 92]. A quality assessment for sorting rules was presented in [Gedigaa, 02]. Due to permanent growing in the information new technologies are necessary for further development. A study of enhanced information retrieval for big data systems was presented in [Choi, 17]. In [Axtmann, 15] was presented practical version of parallel approach for massive sorting to increase efficiency for big data processing. Sorting and its various application are very important for information management, e.g. an adaptation of chrological big data curation sorting on the network was given in [Choi, 17].

1.1 Related works

Sorting algorithms are an integral part of modern information systems. They also find application in the secure multi-party level security (MPC), and several MPC sorting protocols have been proposed in many works. Analysis of the computational complexity of quick sort and other sorting algorithms in the MPC protocol was described in [Bogdanov, 14], [Hamada, 12]. Moreover, , there is a long list of works to improve sorting in actively SMC private protocol, for example [Laud, 16]. During our research on possible improvements in sorting methods we were working on two aspects: faster sorting and easier data management by applied structures. Our proposition to change the method of divisions in quick sort was presented in [Woźniak, Gubias, 06]. In presented examinations we have shown that dynamic changes of the division position can increase the speed of sorting and prevent the method from deadlocks that are well known for the classic version. Results of our research on various aspects of heap composition used in heap sort algorithm were presented in [Woźniak, 13b]. We have shown that changes in the composition of the levels of the heap can improve management of stored information and positively influence sorting. Our examinations on merge sort were discussed in [Woźniak, 13a]. We have proposed changes in merging to dynamically assign the "divide and conquer" assumption what improved the speed of sorting and results in the implementation of non-recursive merging. In [Marszałek, 15] we have shown some further improvements to gain on speed. An efficient parallelization of designed by us modified merge sort algorithm was discussed in [Marszałek 17]. The results of our research were examined in practical applications designed for Hadoop systems presented in [Czerwiński, 15]. Our research on efficient sorting algorithms gave an introduction to work on the new method based on our previous results. The initial idea for the method which we called Fast Sort Algorithm was presented in [Marszałek, 16].

Research on reducing computational complexity of sorting algorithms are carried out for many years. Initially theoretical works have shown that there is the smallest asymptotic complexity of sorting algorithms, which all the methods may try to approach. Richard Cole [Cole, 88] described merge sort algorithm of complexity $O(\log_2 n)$ using n processors, with very large time constant influencing sorting time. Our article represents the approach for practical design capabilities of parallel algorithms with the lowest computational complexity. In addition, the article [Cole, 88] used binary trees while we use the separation of concerns approach. Sorting methods discussed in the works [Carlsson, 90], [Gubias, 06], [Huang, 89] have some limitations to use in database applications with multi core processing units. We can say that there is a large discrepancy between theoretical works and practical methods in parallel computing processes. The authors of the recent work are trying to fill the gap between theory and practice. In [Marszałek, 16] was described an introduction of the Fast Sort Algorithm, and in [Marszałek 17] was presented a parallel version of our modified merge algorithm. The results of our further research presented in this article show the parallelization of sorting processes and reducing the time complexity $O(n)$ by the use of $n/3$ processors without any cross actions between logical processors. The research on the efficient parallel sorting algorithm benefited in this new model

with a very high performance. The difference between sorting by parallel merging [Marszałek 17] and parallel fast sort algorithm described in this article is in the method of merging of sorted strings. Parallel merge implements the real connection in each iteration and pairs all sorted strings. The parallel fast sort algorithm only links the data in each iteration by the use of the number of logical processors and as a result three sorted strings are composed into one sorted string. This allowed the design of the parallel fast sort algorithm. In each iteration the number of independently operating processors cooperates on sorting what makes the actual size of the task.

In this article we present the parallel fast sort algorithm, applicable to any number of logical processors. The method is composed for n independent processors in that way that during operations all of them work without any cross actions and any interruptions between each other. The proposed design of the parallel fast sort algorithm has time complexity $O[(\log_3 n)^2]$.

2 Parallel Fast Sort Algorithm

For the analysis of parallel sorting algorithm it is convenient to use the parallel machine model - the PRAM (Parallel Random Access Machine) shown in Fig. 1. Depending on the method of access of the processors to the memory, we can specify three types of PRAM machines:

- Exclusive Read Exclusive Write (EREW)
- Concurrent Read Exclusive Write (CREW)
- Concurrent Read Concurrent Write (CRCW)

The first type of PRAM allows to read/write memory using only one processor. The second type provides reading memory by any number of processors, however wringing at the same time can be run only by one processor. The third type allows to access memory using any number of processors. The second one of presented PRAM models reflects the architecture of the modern computer and practically makes it possible to write efficient parallel implementations.

A very important issue for sorting of data sets is the possibility of parallelism between sorting processes. The PRAM machine model can be used to model the division of tasks with low time complexity. This idea was adopted by the EREW PRAM model, which allowed to sort numbers in time $O(n)$. Suppose now we can use the CREW PRAM model, which will be used to describe the parallel sorting algorithm acting on *n* processors.

2.1 Parallel algorithms for merging two sorted strings

Suppose that we have two sorted strings containing *n* elements each that $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$ i $y_0 \leq y_1 \leq \cdots \leq y_{n-1}$. The merge algorithm performs insertions of the elements from an array x or y into the output array $z_0, z_1, \ldots, z_{2n-1}$ by the processor number i, $0 \leq i < 2n$ as follows.

Processor number i, where $0 \leq i < n$ computes the index of the element y_t before which the new element x_i should be inserted to have $y_{t-1} < x_i \leq y_t$. In this case the insertion must be done after the last element of the array y with the value of the index n. Processor i performs insertion of the value of the element x_i in the string z under index $i + t$. Imagine, for instance, this way to merge two strings $x = [1,4,6,7]$, $y = [0,2,5,9]$ using processors P_0, P_1, P_2, P_3. The situation is shown in Fig. 2.

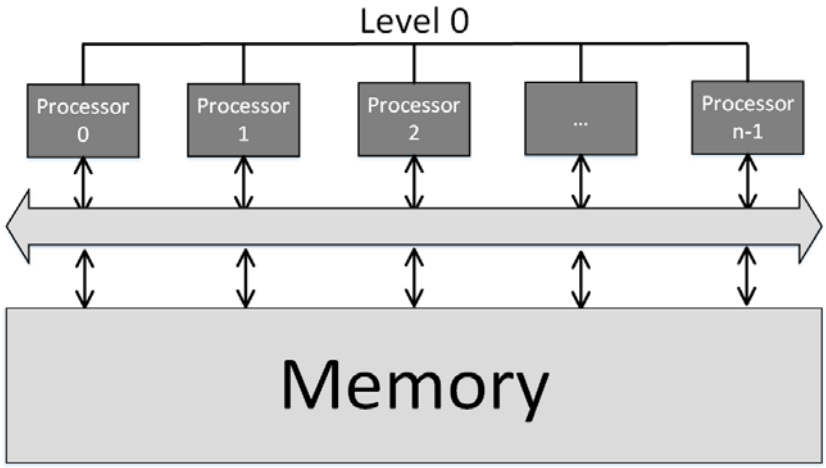

Figure 1: A sample schema of the Parallel Random Access Machine

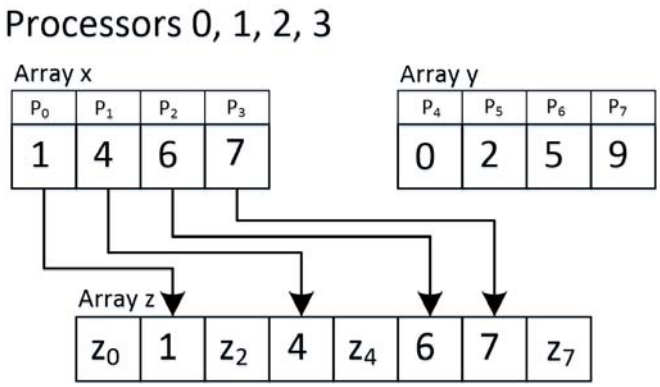

Figure 2: The model of merging array x with array z by applied number of processors

Each of processors $P_i, i = 0,1,2,3$ operates independently and determines the index t_i of the element in the array y, before which the new element should be inserted. For example, for the processor P_1, the element $x_1 = 4$ is inserted prior to $y_2 = 5$. Hence, the index is calculated and the new inserted element is 4, which we put into the array z. This is equal to the sum of the indexes of elements $x_1 = 4$, $y_2 = 5$ which in this case is *3*, see Fig 3.

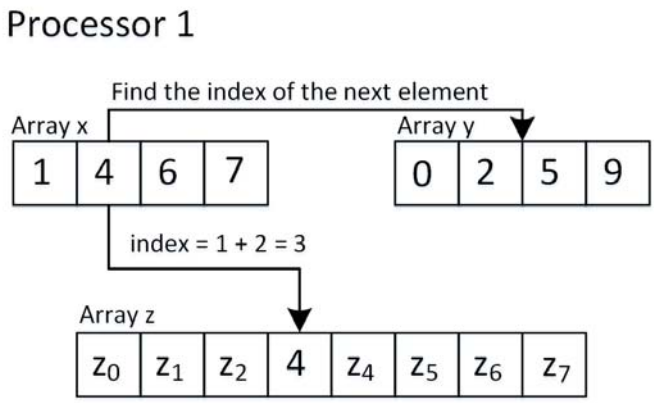

Figure 3: The method of inserting element x_t into the array z by the processor 1

Processor number i, where $n \leq i < 2n$ computes the index of the element x_t before which the new element should be inserted to have y_{t-n}, $x_{t-1} \leq y_{t-n} < x_t$. In this case the insertion must be done after the last element of the array y, with the value of the index n. Processor i performs insertion of the value of the element y_{t-n} into the string z under the index $i-n+t$. Imagine, for instance, this way to merge two strings $x = [1,4,6,7]$, $y = [0,2,5,9]$ using processors P_4, P_5, P_6, P_7. This situation is shown in Fig. 4.

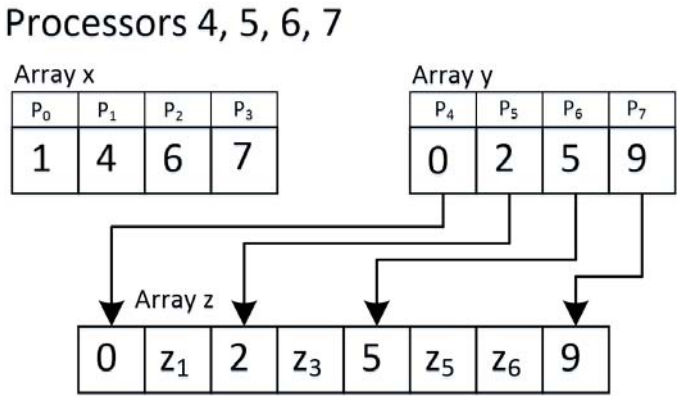

Figure 4: The model of merging array y with array z by applied number of processors

Each of the processors $P_i, i = 4,5,6,7$ operates independently and determines the index t_i of the element in the array x, before which the new element should be inserted. For example, for the processor P_6, the element $y_2 = 5$ is inserted prior to $x_2 = 6$. Hence, the index is calculated and the new inserted element is 5, which we put into the array z. This is equal to the sum of the indexes of elements

$y_{6-4} = y_2 = 5$, $x_2 = 6$ and is 4, see Fig 5. The index of the processor is equal to the sum of the indexes of the inserted element into the array y and the number of elements in the array x.

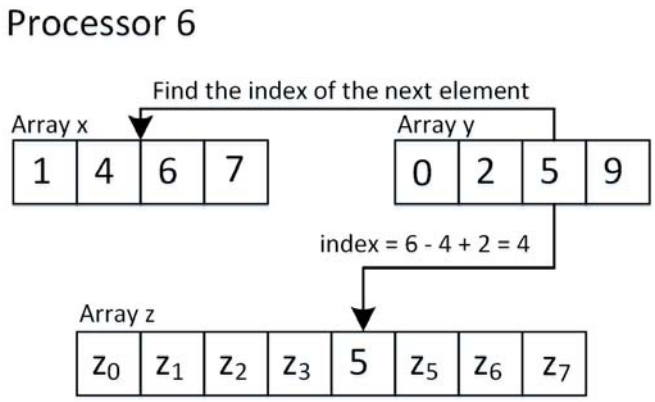

Figure 5: The method of inserting element y_2 into the array z by the processor 6

Similar model of calculations of the index of the element, before which to insert the new element via one of the processors was used in the binary search algorithm [8]. That model had time complexity $O(log_2 n)$. Since we assume that used processors operate independently in the machine model CREW PRAM with $2n$ processors, the whole algorithm for parallel merging of two strings will perform in much shorter time.

In the next section we describe the newly proposed parallel fast sort algorithm, which only links the data in each iteration using the number of available independently working logical processors. All of them cooperate on the input information to sort it without crossing or interruptions, and as a result the processed strings are composed into one sorted output. The new method is developed for *n* independent processors, what makes it more powerful with each new available processor.

2.2 Improved parallel algorithm without core-crossing actions

Proposed Parallel Fast Sort Algorithm (PFSA) makes possible the division of tasks between independently working logical processors without cross actions. To develop it we have used the model of the machine CREW PRAM. We use a temporary array to merge the first two strings. For the efficiency of the processing, the third string remains to be rewritten into the temporary array. Due to applied model of the machine CREW PRAM all the processors can read the data but at the same time write into the cell which is not currently being reorganized by any other processor. The initial method proposed in [23] for faster sorting 3 strings could use only one processor. Therefore the time complexity was $O(n)$, since the tasks could not be divided between processors. Proposed PFSA method allows to independently use *n* processors and therefore lower the time complexity to $O[(\log_3 n)^2]$.

An example of the first stage in proposed PFSA of two lines is shown in Fig. 6. The PFSA merges the strings stored in a temporary array to the third string located in the input array. The result of merge is stored in the input array. A sample schema of parallelized process of merging n/3 strings is shown in Fig. 7.

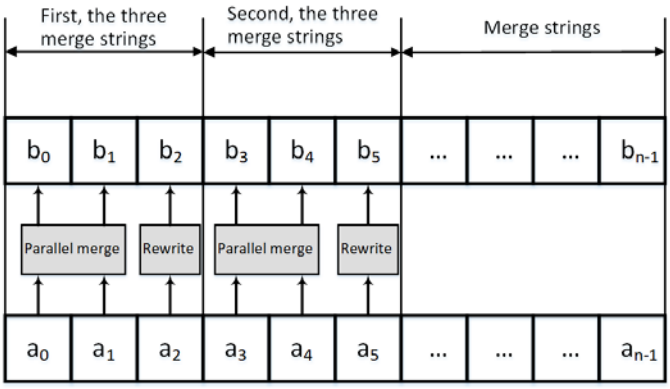

Figure 6: Parallel merge of the first two numeric strings in the first step of the proposed PFSA

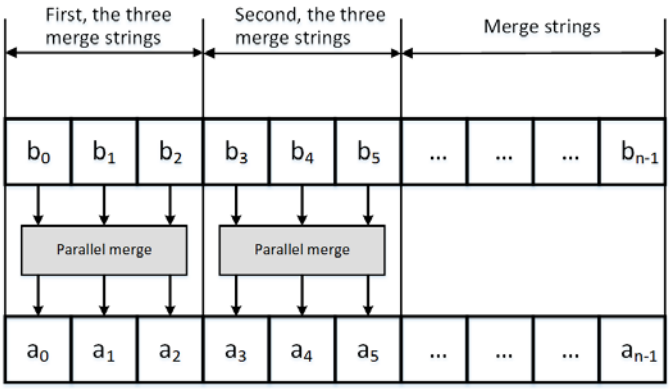

Figure 7: Parallel merge of the temporary array into the third string located in the input array of the proposed PFSA

In the next steps of the PFSA we merge strings enlarged each time three times, see Fig. 8.

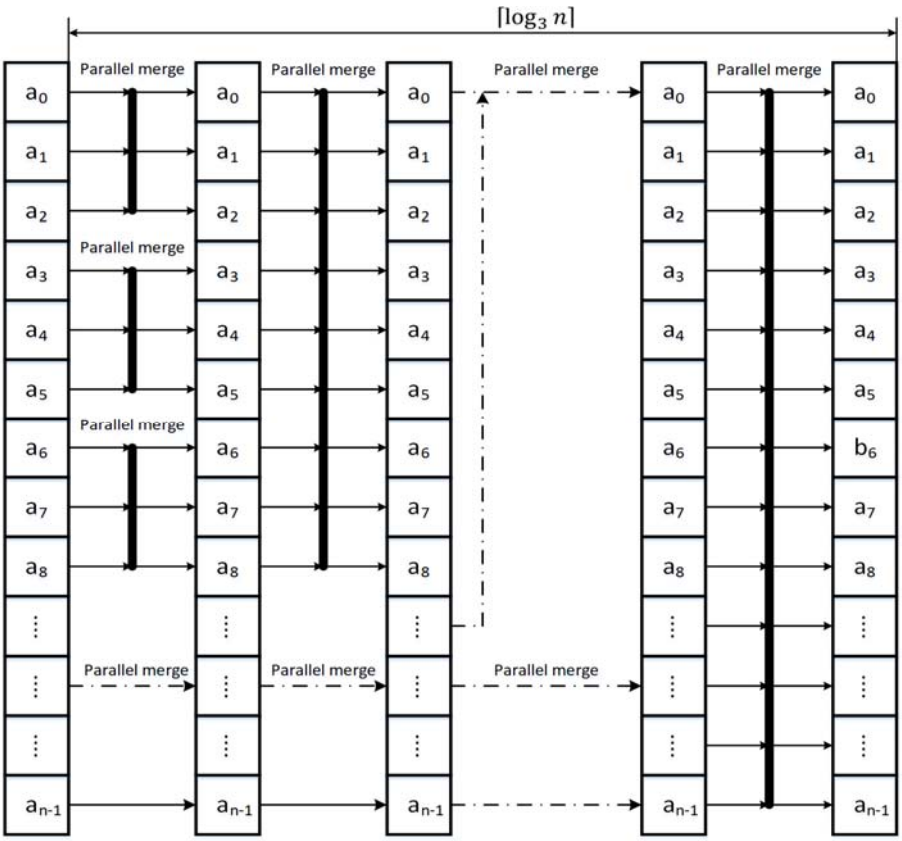

Figure 8: Sample schema of the Parallel Fast Sort Algorithm

THEOREM I. Presented improved version of the Parallel Fast Sort Algorithm using n processors working independently on the machine CREW PRAM has time complexity $O[(\log_3 n)^2]$.

Proof. We are limiting our deliberations to $n = 3^k$, where $k = 1,2,\ldots$. Let us first notice that sequences $x_1 \leq \cdots \leq x_m$ and $y_1 \leq \cdots \leq y_m$ of t elements we can merge into one sequence $z_1 \leq \cdots \leq z_{2m}$ using n processors. Therefore parallel merging of these two sequences will make no more than $\lceil \log_2 n \rceil + C$ comparisons of the elements in sequences X and Y. Thus, the time complexity of the parallel merge algorithm for two strings on a CREW PRAM machine is $O(\log_2 m)$.

At each step $t = 1,\ldots,k = \log_3 n$, in the beginning of the sorting algorithm we save in the temporary array two strings and additional one as the third in a row. Next, we merge these two strings of the temporary array and save united strings in the array of the sorted elements. Because all processors work independently the thread synchronization happens after each stage of merging of two strings from the

temporary array. Therefore the maximum operating time of each step of the merge process of three strings is

$$T_{max}(t) = \lceil \log_2 3^{t-1} \rceil + \lceil \log_2 (2 \cdot 3^{t-1}) \rceil + C_1 = 2\lceil \log_2 3^{t-1} \rceil + C_2 \quad (1)$$

again, for the processing time of each step, we get

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

therefore when calculating

$$\sum_{t=1}^{k} \lceil \log_2 3^{t-1} \rceil = \lceil \log_2 3 \rceil + 2 \lceil \log_2 3 \rceil + \cdots + (k-1) \lceil \log_2 3 \rceil =$$

$$= \lceil \log_2 3 \rceil [1 + 2 + \cdots + (k-1)] = \frac{k(k-1)}{2} \lceil \log_2 3 \rceil =$$

$$= \frac{\log_2 n (\log_2 n - 1)}{2} \lceil \log_2 3 \rceil \quad (3)$$

so by substituting and taking into account $\lceil \log_2 3 \rceil = 2$, we get

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

which was to prove.

This is an estimation of the indexes of the items being inserted. First, two strings are merged and written to the auxiliary table and the third one is just rewritten as shown in Fig 6. As stated above, the processing time of the longest operation is $\lceil \log_2 3^{t-1} \rceil + C$. Then all the processors are waiting for the insertion of elements to be finished, and start inserting new elements from this array into the array of output ordered elements. The maximum running time for each processor is $\lceil \log_2 (2 \cdot 3^{t-1}) \rceil + C$. After completing the insertion, they proceed to the next step of sorting and merge three times longer arrays. The time of each sort step by n processors is estimated. If the merged string is divided among k processors, then we get n/k merged substrings processed by the number of used processors. Each subsequence according to the Theorem I is merged in time $O[(\log_2 n)^2]$. All the processors work independently and are equally efficient. Hence as the conclusion we can get the final theorem.

THEOREM II. By using k processors for proposed PFSA method on CREW PRAM machine, we can lower the time complexity to $O\left(\frac{n(\log_2 n)^2}{k}\right)$.

Proof. The proof comes as a natural derivation from the proof of the Theorem 1.

2.3 Implementation of the method

Presented PFSA method was implemented in C# Visual Studio Enterprise 2015. The algorithm uses a parallel loop, which takes as arguments the start index, the number of iterations, and the action object for (Action object<int>) Loop Parallel. This construction efficiently reduces created program code, because there is no need to create separate tasks, run them and wait for them to be finished. The algorithm presented in Algorithm 1 uses maximum number of processors available in the system. Due to preservation of the order the insertion of the elements is integrated within the loop. Therefore two functions are targeted to deliver the index of the element before which we are about to insert the new element. The first function returns an index to the next element in the string on the right side, see Algorithm 2. The second function returns an index to the next element in the string on the left side, see Algorithm 3. Sorting function uses the possibility of targeted delivery of an index to the next element in correctly merged two strings. The block diagrams of implemented algorithms are presented in Fig. 9 and Fig. 10.

```
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 t
While t is less than n then do
Begin
    Remember 2*tt in t_0
    Remember 3*tt in t_1
    Parallel for each processor at index i_1 greater
    or equal 0 and less than n do
    Begin parallel for
        Remember i_1 / t_1 in j
        Remember t_1 * j in i
        Remember i_1 % t_1 in iw
        Remember i + t in p_1
        If p_1 greater than n then do
        Begin
             Remember n in p_1
        End
        Remember i + t_0 in p_2
        If p_2 greater than n then do
        Begin
             Remember n in p_2
        End
        If i_1 less than p_1 then do
        Begin
             Proceed  function index located in the right sting in array a and
remember found index in iz
             Remember a[i_1] in b[iz + i + iw]
        End
         Else
```

```
        If i_1 less than p_2 then do
        Begin
            Proceed  function index located in the left sting in array a and remember
found index in iz
            Remember a[i_1] in b[iz + i_1 - t]
        End
        Else
        Begin
             Remember a[i_1] in b[i_1]
        End
     End of the parallel for
    Parallel for each processor at index i_1 greater or equal 0 and
less than n do
    Begin parallel for
        Remember i_1 / t_1 in j
        Remember t_1 * j in i
        Remember i_1 % t_1 in iw

        Remember i + t_0 in p_2
        If p_2 greater than n then do
        Begin
             Remember n in p_2
        End
        Remember i + t_1 in p_3
        If p_3 greater than n then do
        Begin
             Remember n in p_3
        End
        If i_1 less than p_2 then do
        Begin
             Proceed  function index located in the right sting in array b and
remember found index in iz
             Remember b[i_1] in a[iz + i + iw]
        End
         Else
        If i_1 less than p_2 then do
        Begin
         Proceed  function index located in the left sting in array b and remember
found index in iz
         Remember b[i_1] in a[iz + i_1 - t_0]
        End
     End of the parallel for
     Multiply variable t by three
End
Stop
```

Algorithm 1 Implementation code of the Parallel Fast Sort Algorithm

```
Start
Load table a
Load dimension of table a into n
Load index up
Load index uk
Load variable ux
If up equals uk then do
Begin
    Return 0
End
Remember up in ud
Remember uk -1 in ug
While ug - ud greater than 1 then do
Begin
    Remember (ud + ug) / 2 in lup
    If ux less or equals a[lup] then do
    Begin
        Remember lup in ug
    End
    Else
    Begin
        Remember lup in ud
    End
End
If ux equals a[ug] then do
Begin
      Remember ug in it
      If it greater than up and a[it - 1] equals ux
    then do
    Begin
        Subtract one from it
    End
End
Else
If ux equals a[ud] then do
Begin
    Remember ud in it
End
Else
If ux less than a[up] then do
Begin
    Remember up in it
End
Else
If ux greater than a[uk - 1] then do
Begin
```

```
    Remember uk in it
End
Else
Begin
    Remember ug in it
End
Return it – up
Stop
```

Algorithm 2 The function code which returns the index of the element located in the right string

```
Start
Load table a
Load dimension of table a into n
Load index vp
Load index vk
Load variable vx
Remember vp in vd
Remember vk -1 in vg
While vg - vd greater than 1 then do
Begin
    Remember (vd + vg) / 2 in lvp
    If vx less than a[lvp] then do
    Begin
        Remember lvp in vg
    End
    Else
    Begin
        Remember lvp in vd
    End
End
If vx equals a[vg] then do
Begin
      Remember vg in it
      If it+1 less than vk and a[it + 1] equals vx
    then do
    Begin
        Add one to it
    End
    If it equals vk -1 than do
    Begin
        Add one to it
    End
End
Else
```

```
If vx equals a[vd] then do
Begin
    Remember vd in it
    If it+1 less than vk and a[it] equals vx
    then do
    Begin
        Add one to it
    End
End
Else
If vx less than a[vp] then do
Begin
    Remember vp in it
End
Else
If vx greater than a[vk - 1] then do
Begin
    Remember vk in it
End
Else
Begin
    Remember vg in it
End
Return it – vp
Stop
```

Algorithm 3 The function code which returns the index of the element located in the left string

2.4 Secure Multiparty Computation

With the growth of information technology, there is a need to protect data privacy and the need to disseminate information without compromising privacy. Many area may be given of public life in which the distributed information should be encrypted in order to preserve privacy, e.g. Medicine, economics, etc. Secure multi-party computation (MPC) is a technique that enables the creation of such secure systems [Bogdanov, 14], [Laud, 16]. In MPC protocols, n parties $\mathcal{P}_1,\ldots,\mathcal{P}_n$ evaluate a function $f(x_1,\ldots,x_n)=(y_1,\ldots,y_n)$, where input x_i and output y_i values are in secret-shared form. Accept the assumption that input and output values for the MPC protocols belong to the field $\mathbb{Z}_p$ and $[\![s]\!]_{\mathcal{P}_i}$ denote a share for $\mathcal{P}_i$ where a secret value is $s \in \mathbb{Z}_p$. Let $\mathbb{Q}$ be a coalition of parties and $[\![s]\!]_{\mathbb{Q}}$ denote a set of shares $\{[\![s]\!]_{\mathcal{P}_i} : \mathcal{P}_i \in \mathbb{Q}\}$. When $\mathbb{U}$ represents all parties, a share value $[\![s]\!]_{\mathbb{U}}$ for convenience denote as $[\![s]\!]$.

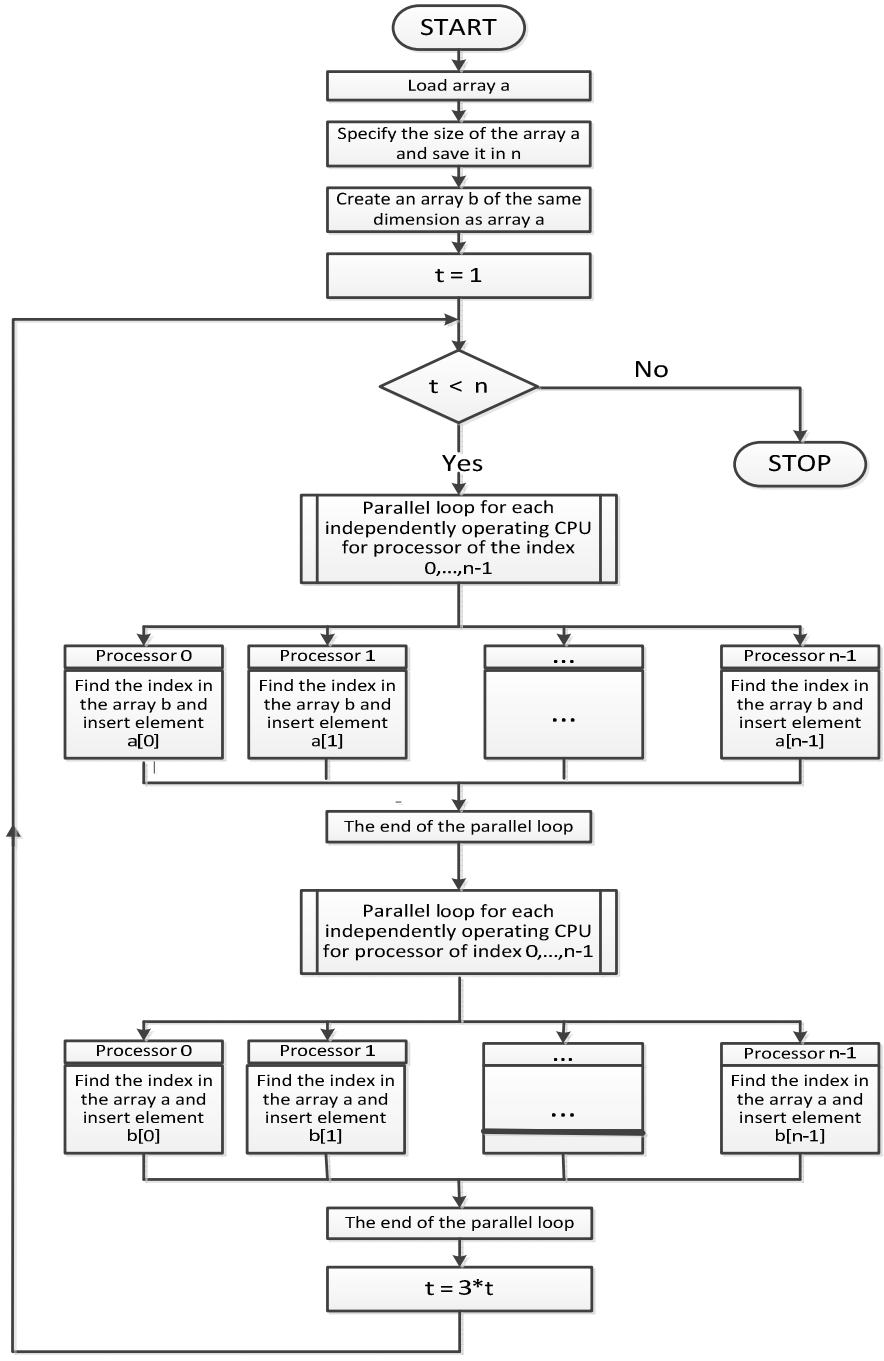

Figure 9: The block diagram of the proposed Parallel Fast Sort Algorithm

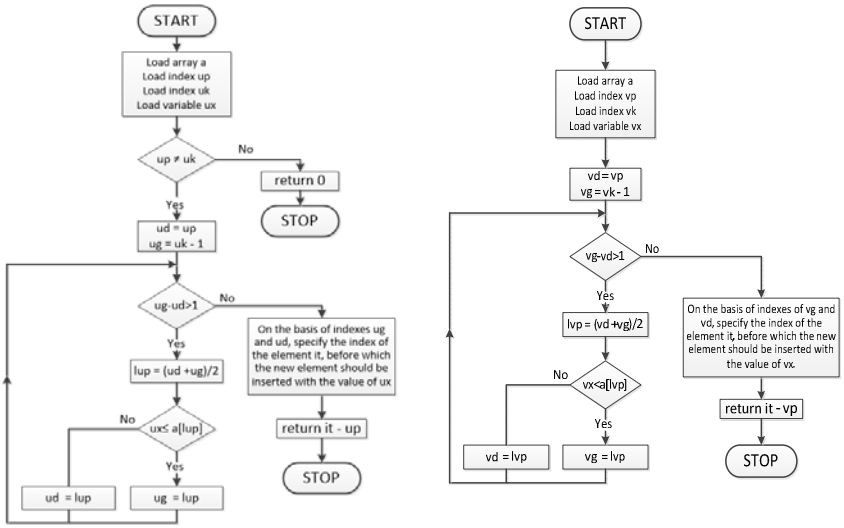

Figure 10: The block diagrams of two functions used in PFSA code implementation to calculate exact position of the insertion, from the left side and from the right side

Another important issue is the security model of information transfer. There is a possibility that the Protocol will be partially damaged at the input and output. We can say that the Protocol is secure if you can identify the broken part of the Protocol at the input and the output, i.e. . $\mathbb{I} = \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_r}\}$ denote the parties that are corrupted. The formal definition of security model can be found [Shamir, 79].

In accordance with the Protocol MPC secret values becomes available to all participants . The secret-shared values is input for each party and output in secret-shared form. In the Protocol MPC, the following protocols are define:

1. Comparison Protocol – outputs a shared value of the comparison result of the inputs two shared values [41]. Formally defining, the comparison protocol takes as input two arguments $[\![a]\!]_{\mathcal{P}_t}$ and $[\![b]\!]_{\mathcal{P}_t}$ for each $\mathcal{P}_t \in \mathbb{U}$ and returning a compression result to the output $[\![c]\!]_{\mathcal{P}_t}$ for each $\mathcal{P}_t \in \mathbb{U}$ [Hamada, 12].
2. Shuffling Protocol – performs uniform random permutation from the input shared values []. Protocol execution is dented as
   $Shuffle([\![a_1]\!], \ldots, [\![a_m]\!]) \rightarrow [\![b_1]\!], \ldots, [\![b_m]\!]$
3. Reveal Protocol – the reveal algorithm in a multi-party setting. Protocol execution is dented as
   $Reveal([\![x]\!]) \rightarrow x$

As always in processing the information, the same way in the processing of secret sharing schemes play an important role sorting algorithms. Appropriately direct use of the known sorting algorithms such as quick sort, heap sort and merge sort is impossible, as the effect of these algorithms is based on comparison operations for elements to be sorted. This problem can be solved in a simple way through execution

shuffling before sorting and application of the method compared the sorted data. Application of the method compared the sorted data, not leaking information, as the ordinal information is randomized by the shuffling. In addition, the method that compares the elements being sorted can be secured so that it is not possible to be a leak in the information outside (in accordance with the principles of object-oriented programming). The solution to the problem of applying known algorithms sorting protocols MPC is simple and effective.

In this paper, it was proposed the method of parallel fast sorting. Parallel Fast Sort Algorithm is a stable method behaved the order of the order of ascending sorted string i.e. The original order of repeated values of the sorted string is preserved. The stability of the method in the order of the duplicate values is a very important feature in the MPC protocol sorting applications.

2.5 Performance Experimental study of the proposed Parallel Fast Sort Algorithm

Performance analysis of the presented method is based on benchmark tests for the algorithm implemented in C# in Visual Studio 2015 Enterprise on MS Windows Server 2012. For tests were used 100 samples generated at random task size from 100 to 100000000 elements, increasing the size of sorted array by ten times each time. Each sorting operation by examined method 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. For the statistical tests and comparisons were used methods as in [23]. We have measured statistical average of *n* elements set of samples $a_1, \ldots, a_n$ defined by the formula

$$\bar{a} = \frac{1}{n}(a_1 + \cdots + a_n). \tag{5}$$

The standard deviation defined by the formula

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(a_i - \bar{a})^2}{n-1}}, \tag{6}$$

where *n* is the number of elements in sample, $a_1, a_1, \ldots, a_n$ of random value variables, $\bar{a}$ is the arithmetic mean of the sample.

In order to compare sorting algorithms we have done an analysis for large sets of data. The analysis for sorting time was carried out in 100 benchmark tests for each of the fixed dimensions of the task. Algorithm stability is described on basis of the coefficient of variation. The coefficient of variation is a measure that allows determining value of diversity in examined sample. It is determined by the formula

$$V = \frac{\sigma}{\bar{a}} \tag{7}$$

where we use arithmetic mean (5) and standard deviation (6). The coefficient of variation reflects the stability of the method in a statistical sense. Benchmark tests of the newly proposed PFSA method were taken for 100, 1000, 10000, 100000, 1000000, 10000000 and 100000000 elements on the input. The results are presented in tables and discussed in the following figures. The purpose of the analysis and comparison is to verify how the newly proposed parallel processing can speed up sorting of data sets. Presented results are averaged for 100 sorting samples. Benchmark tests for PFSA are described in Tab. 1 - Tab. 4 and comparison of the results to other sorting methods is presented in Tab. 5. For these comparisons we have used three methods: quick sort [Woźniak, 15], heap sort [Woźniak, 13] and merge sort [Marszałek, 15].

Elements	1 – processor	2 – processors	4 – processors	8 – processors
100	1	1	1	1
1 000	1	1	1	1
10 000	16	10	6	5
100 000	235	154	116	52
1 000 000	2999	1568	829	516
10 000 000	41766	21560	11300	6734
100 000 000	429462	278583	143647	82079

Table 1: The result of sorting for parallel fast sort algorithm in [ms]

From Tab. 1 we can see that sorting time is increasing with the number of sorted elements. The results are presented in Fig. 11. However the time is lower with each new processor used for sorting. The most visible difference is for the data sets above 10 000 000 elements. That shows a positive influence of proposed parallel processing, since with the increasing number of used processing cores the PFSA method is able to efficiently sort huge amounts of information.

Analyzing Tab. 2 we see how the number of calculations on each applied processing architecture changes with additional cores. The results are presented in Fig. 12. We can see the same situtation as for time. With increasing number of used cores the number of calculations is lower. Since all the processors are working independently the PFSA method can be very efficient even for large data sets. The efficiency depends on the number of used processing units. Comparison of coefficient of variation for PFSA is presented in Tab. 3 and Tab. 4, for time and calculations respectively. Analyzing these values we can see that with increasing number of processing units the proposed method is more stable in a statistical way. That shows a positive impact of the proposed PFSA implementation with no cross-actions between

processors. The algorithm for any number of CPUs used in sorting has very similar stability in each class of input cardinality. Some variations in stability of the algorithm are visible only for small inputs. These are due to the fact that operating system automatically exceeds the sorting algorithm, what can cause some additional operations.

Elements	1 – processor	2 – processors	4 – processors	8 – processors
100	3251	2523	2383	1397
1 000	3627	2752	2270	1958
10 000	52787	30685	20260	14290
100 000	755410	496632	371700	167045
1 000 000	9663086	5053206	2670337	1662674
10 000 000	134558726	69459912	36406762	21695164
100 000 000	1705779646	897515824	462790092	264436121

Table 2: The results of sorting for parallel fast sort algorithm in [ti]

Elements	1 – processor	2 – processors	4 – processors	8 – processors
100	1.1024162	1.1034171	1.0012263	1.1043141
1 000	0.7559289	0.9258200	0.9225771	0.9817180
10 000	0.1549049	0.1647508	0.1886751	0.2267786
100 000	0.1399457	0.1415474	0.1619496	0.1079664
1 000 000	0.0641322	0.0599950	0.0672706	0.0559623
10 000 000	0.0586108	0.0605951	0.0625390	0.0541402
100 000 000	0.0532901	0.0609835	0.0564864	0.0516798

Table 3: Coefficient of variation for parallel fast sort algorithm in [ms]

Elements	1 – processor	2 – processors	4 – processors	8 – processors
100	1.4543798	1.3870293	1.2285883	1.1669990
1 000	0.2131656	0.1766220	0.1969925	0.1027110
10 000	0.1480054	0.1158681	0.1775589	0.1358741
100 000	0.1399730	0.1410735	0.1650359	0.1191470
1 000 000	0.0641393	0.0598660	0.1751595	0.0674605
10 000 000	0.0586089	0.0605900	0.1769479	0.0625266
100 000 000	0.0532901	0.0609837	0.1939433	0.0564883

Table 4: Coefficient of variation for parallel fast sort algorithm in [ti]

The results of comparison are presented in Tab. 5. In Fig. 13 and Fig. 14 we can see a comparison of sorting time and operations, respectively. The proposed method becomes more efficient with each new processing core that can is used for sorting. We can say that approximately each new core is able to speed up the process of sorting of about 5% to 10%. This possibility is very important for large data sets, where new computer architectures can efficiently support operations. This result is possible due to the proposed implementation of the method. We have very efficient separation of concerns for each applied processor, what results in a fact that we have no cross-actions between processors.

Elements	QS	3HS	MS	PFSA
100	1	1	1	1
1 000	1.95	1.6	1.95	1.95
10 000	6	4	6	4
100 000	75	51	55	47
1 000 000	933	629	576	368
10 000 000	10962	8087	6665	4795
100 000 000	125400	102071	75007	58957

Table 5: Comparisons of sorting time for examined methods in [ms]

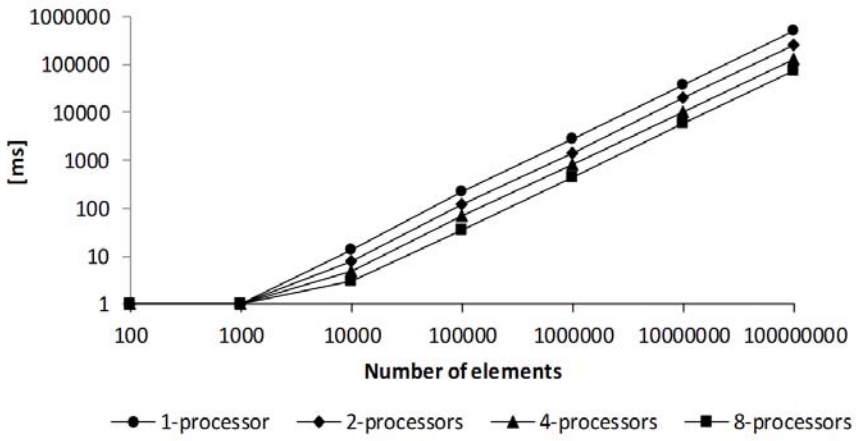

Figure 11: Comparison of benchmark sorting time [ms] for various number of used processors

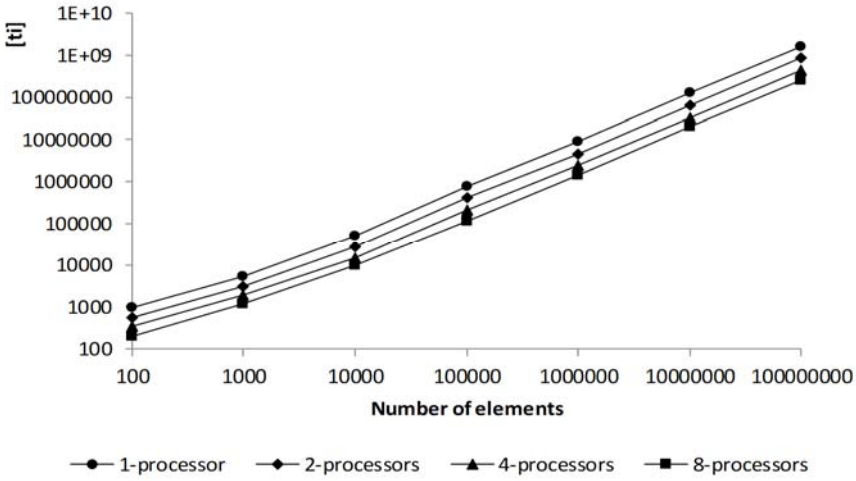

Figure 12: Comparison of benchmark sorting operations [ti] for various number of used processors

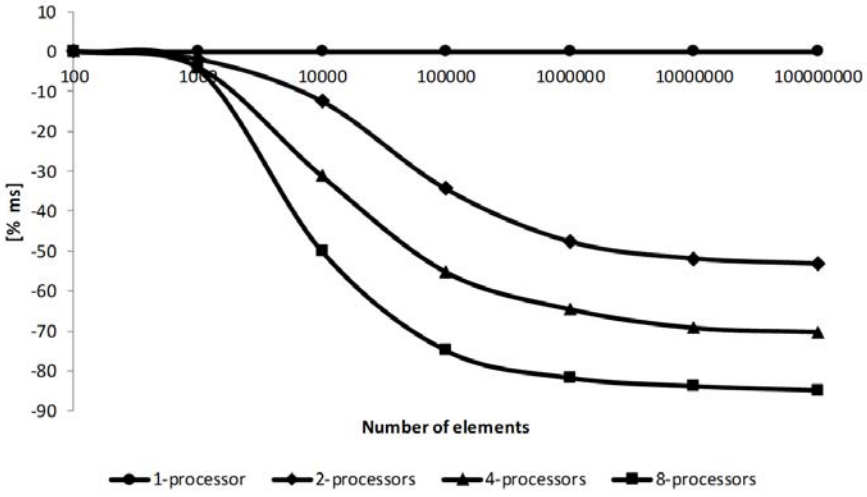

Figure 13: Comparison of the method efficiency for various number of used processors in terms of sorting time [ms]

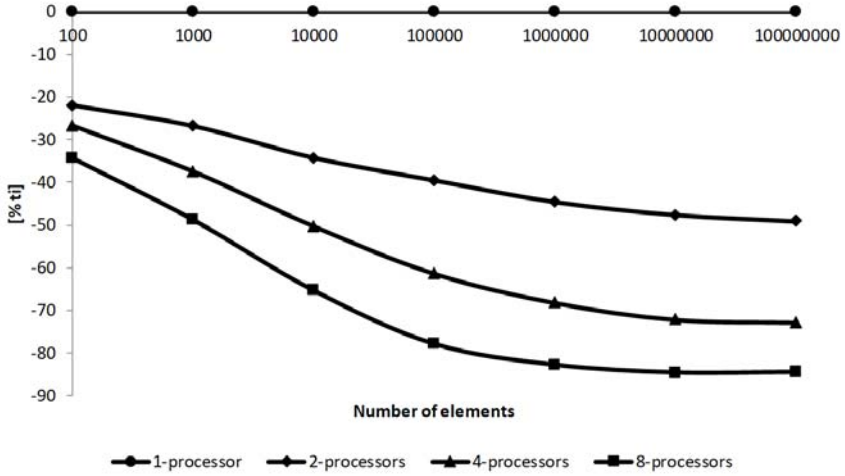

Figure 14: Comparison of the method efficiency for various number of used processors in terms of sorting operations [ti]

3 Analysis and comparison of sorting time

Analysis and comparison will help to estimate the PFSA efficiency for sorting. To present the proposed method we have compared it with Quick Sort algorithm (QS) presented in [29], Heap Sort algorithm for three divisions on each node in the levels

of the heap (3HS) presented in [Woźniak, 13] and Merge Sort algorithm (MS) presented in [Marszałek, 15]. Sample results are presented in Tab. 5 and comparison of the results is visible in Fig. 15. The proposed method can be less efficient for data sets of up to 1 000 000 elements. We can say that in this cardinality other methods can show about 10% higher performance. But at the same time the study shows that PFSA operates in shorter time measuring tasks for above 1 000 000 elements. This is a very promising result. The method becomes more efficient with each new applied core and it's performance becomes more visible for large data sets. The proposed method is effective when using a large number of processors available in modern chipsets. It's theoretical complexity is $O\left(\frac{n(\log_2 n)^2}{k}\right)$, where k is the number of logical processors that are used in calculations. As we can see from this analysis our tests conducted on a limited number of processing units fully confirm the theoretical results.

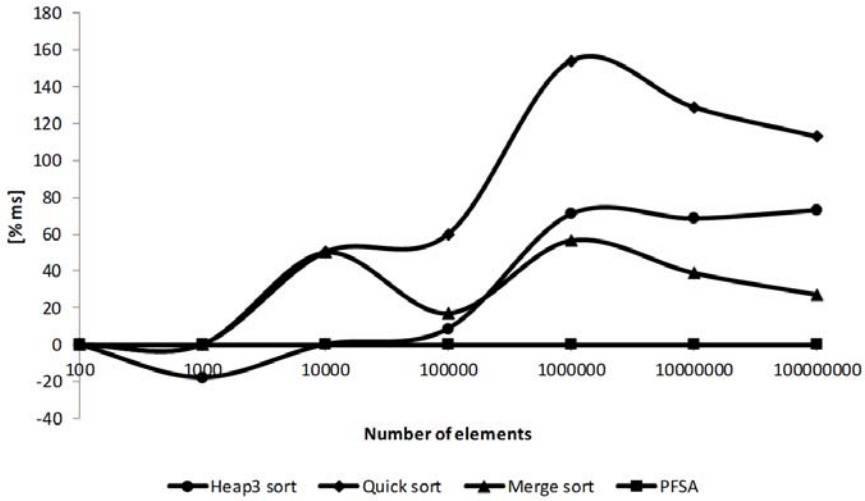

Figure 15: Comparison of the tested sorting methods in terms of sorting time [ms]

4 Final Remarks

The article presents a new method of distribution of tasks among processors in a parallel method on the principle that all processors can read memory cells but only one processor can write to the same memory cell. The implementation of the method makes it work without any cross-actions, therefore all the processors are working totally independently. Due to the design of the new method the sorting is performed faster. The PFSA algorithm uses interaction features of modern processors and leads to method working in time $O[(\log_2 n)^2]$ when using *n* logical processors.

Presented method of parallel sorting for large data sets may find practical application in NoSQL databases but also in chipsets with large number of processing units. The stability of the algorithm and it's theoretical time complexity have been

confirmed during testing. As we have seen from the benchmark analysis, the effectiveness of the proposed division of tasks between processors is displayed when using a large number of logical processors available in modern computers. The method performs better with increasing number of cores, that gives very promising results for modern powerful chipsets. Further work will be done on the parallel computing in the secure multi-party level security. In particular, security research will be carried out on the sorting and active private protocol algorithms to ensure the full security of sensitive applications.

References

[Abrahamson, 87] Abrahamson, K., Dadoun, N., Kirkpatrick, D., Przytycka, T.: A simple parallel tree construction algorithm, Journal Algorithms, no.10, 1987, pp. 287–302.

[Aho, 75] Aho, I., Hopcroft, J., Ullman, J.: The design and analysis of computer algorithms, Addison-Wesley Publishing Company, USA, 1975.

[Alanko, 84] Alanko, T., Erkio, H., Haikala I.: Virtual memory behavior of some sorting algorithm, IEEE Transactions on Software Engineering, vol.10, no.4, 1984, pp. 422–431.

[Axtmann, 15] Axtmann, M., Bigmann, T., Schulz, C., Sanders, P.: Practical massively parallel sorting, SPAA '15 Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures, Portland, Oregon, USA — June 13 - 15, 2015, pp. 13-23, DOI: 10.1145/2755573.2755595

[Ben-Or, 83] Ben-Or, M.: Lower bounds for algebraic computation trees, Proceedings of 15th ACM Symp. Theory of Computing, ACM Press, 1983, pp. 80–86.

[Bing-Chao, 86] Bing-Chao, H., Knuth, D.: A one-way, stack less quick sort algorithm, BIT, vol. 26, pp. 127–130, 1986.

[Bogdanov, 14] Bogdanov, D., Laur, S., & Talviste, R.: A practical analysis of oblivious sorting algorithms for secure multi-party computation. In Nordic Conference on Secure IT Systems (pp. 59-74). Springer, Cham (2014, October).

[Carlsson, 90] Carlsson, S., Levcopoulos, C., Petersson O.: Sublinear merging and natural merge sort, Lecture Notes on Computer Science - SIGAL'1990, vol. 450, pp. 251–260, 1990. DOI: 10.1007/3-540-52921-7_74

[Cole, 88] Cole R.: Parallel merge sort, SIAM Journal on Computing vol. 17, no. 4, pp. 770–785, 1988. DOI: 10.1137/0217049.

[Crescenzi, 03] Crescenzi, P., Grossi, R., Italiano, G.F.: Search data structures for skewed strings, Lecture Notes in Computer Science, vol. 2647, Springer-Verlag Berlin Heidelberg, 2003, pp. 81–96.

[Czerwiński, 15] Czerwiński, D.: Digital filter implementation in Hadoop data mining system, Communications in Computer and Information Sciences - CN'2015, vol. 522, pp. 410–420, 2015. DOI: 10.1007/978-3-319-19419-6 39.

[Choi, 17] Choi, S., Seo, J., Kim, M., Kang, S., Han, S.: Chrological Big data Curation: A Study on the Enhanced Information Retrieval System, IEEE ACCESS, vol. 5, pp. 11269-11277, 2017. DOI: 10.1109/ACCESS.2016.2642979

[Doberkat, 83] Doberkat, E.: Inserting a new element into a heap, BIT Numerical Mathematics, vol.21, 1983, pp. 255–269.

[Estivill-Castro, 92] Estivill-Castro, V., Wood, D.: A survey of adaptive sorting algorithms, Computing Surveys, vol.24, no.4, 1992, pp. 441–476.

[Francis, 92] Francis, R., Pannan, L.: A parallel partition for enhanced parallel quick sort, Parallel Comput, vol. 18, no. 5, pp. 543–550, 1992.

[Gubias, 06] Gubias, L.: Sorting unsorted and partially sorted lists using the natural merge sort, Software Practice and Experience, vol. 11, no. 12, pp. 1339–1340, 2006. DOI: 10.1002/spe.4380111211.

[Gedigaa, 02] Gedigaa, G., Duntschb, I.: Approximation quality for sorting rules", Comput. Stat. Data Anal., vol 40, pp. 499–526, 2002.

[Hamada, 12] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In International Conference on Information Security and Cryptology (pp. 202-216). Springer, Berlin, Heidelberg (2012, November).

[Huang, 89] Huang, B., Langston, M.: Merging sorted runs using main memory, Acta Informatica, vol. 27, no. 3, pp. 195–215, 1989. DOI: 10.1007/BF00572988.

[Huang, 89] Huang, B., Langston, M.: Practical in-place merging, Communications of ACM, vol. 31, no. 3, pp. 348–352, 1989. DOI: 10.1002/spe.4380111211.

[Knuth, 98] Knuth, D.: The art of computer programming Vol.3: Sorting and Searching, Addison-Wesley, USA, 1998.

[LaMarca, 97] LaMarca, A., Ladner, R.: The influence of caches on the performance of sorting, Proceedings of ACM-SIAM Symposium on Discrete Algorithms, pp. 370–379, 1997.

[Larson, 98] Larson, P., Graefe, G.: Memory management during run generation. External Sorting, Proceedings of SIGMOD, 1998, pp. 472–483.

[Laud, 16] Laud, P., Pettai, M.: Secure multiparty sorting protocols with covert privacy. In Nordic Conference on Secure IT Systems (pp. 216-231). Springer International Publishing (2016, November).

[Lindell, 09] Lindell, Y., Pinkas, B.: Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, *1*(1), 5 (2009).

[Lutz, 89] Lutz, M., Wegner, L., Teuhola, J.: The external heap sort, IEEE Transactions on Software Engineering, vol. 15, no. 7, pp. 917–925, 1989. DOI: 0098-5589/89/0700-0917.

[Marszałek, 16] Marszałek, Z.: Novel recursive fast sort algorithm, Communications in Computer and Information Science ¬ ICIST'2016, vol. 639, pp. 344-355, 2016. DOI: 10.1007/978-3-319-46254-7_27

[Marszałek, 15] Marszałek, Z., Woźniak, M., Borowik, G., Wazirali, R., Napoli, C., Pappalardo, G., Tramontana, E.: Benchmark tests on improved merge for big data processing, Asia-Pacific Conference on Computer Aided System Engineering APCASE'2015, IEEE, 14-16 July, Quito, Ecuador, pp. 96–101. DOI: 10.1109/APCASE.2015.24.

[Marszałek 17] Marszałek, Z.: Parallelization of Modified Merge Sort Algorithm, Symmetry, vol. 9, no. 9, pp. 176:1-176:18. DOI: 10.3390/sym9090176

[Nishide, 07] Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 343–360. Springer, Heidelberg (2007)

[Rauh, 10] Rauh, A., Arce, G.: A fast weighted median algorithm based on quick select, Proceedings of the IEEE 17th International Conference on Image Processing, IEEE, 26-29 September, 2010, Hong Kong, 2010, pp. 105–108.

[Roura, 01] Roura, S.: Digital access to comparison-based tree data structures and algorithms, Journal Algorithms, vol.40, no.1, 2001, pp. 123–133.

[Shamir, 79] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)

[Shen, 17] Shen, Z., Zhang, X., Zhang, M., Li, W., Yang, D.: Self-Sorting-Based MAC Protocol for High-Density Vehicular Ad Hoc Networks, IEEE ACCESS, vol. 5, pp. 7350-7361, 2017. DOI: 10.1109/ACCESS.2017.2692254

[Wegner, 89] Wegner, L., Teuhola, J.: The external heap sort, IEEE Transactions on Software Engineering, vol.15, no.7, 1989, pp. 917–925.

[Woźniak, 15] Woźniak, M., Marszałek, Z., Gabryel, M., Nowicki, R.: Preprocessing large data sets by the use of quick sort algorithm, Advances in Intelligent Systems and Computing - KICSS'2013, vol. 364, pp. 111–121, 2015. DOI: 10.1007/978-3-319-19090-7 9.

[Woźniak, 13a] Woźniak, M., Marszałek, Z., Gabryel, M., Nowicki, R.: Modified merge sort algorithm for large scale data sets, Lecture Notes in Artificial Intelligence - ICAISC'2013, vol. 7895, 612–622, 2013. DOI: 10.1007/978-3642-38610-7 56

[Woźniak, 13b] Woźniak, M., Marszałek, Z., Gabryel, M., Nowicki, R.: Triple heap sort algorithm for large data sets, in A. M. J. Skulimowski (Eds.), Looking into the Future of Creativity and Decision Support Systems, Progress & Business Publishers, 7-9, November, Cracow, Poland, pp. 657–665

[Tsigas, 03] Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quick sort and its performance evaluation on SUN enterprise 10000. Proceedings of Euromicro Workshop on Parallel, Distributed and Network-Based Processing, pp. 372–381, 2003.

[Zhang, 96] Zhang, W., Larson, P.: Speeding up external merge sort, IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 2, pp. 322–332, 1996. DOI: 10.1109/69.494169.

[Zhang, 97] Zhang, W., Larson, P.: Dynamic memory adjustment for external merge sort, Proceedings of Very Large Data Bases Conference, 1997, pp. 376–385.

[Zhang, 98] Zhang, W., Larson, P.: Buffering and read-ahead strategies for external merge sort, Proceedings of Very Large Data Bases Conference, Morgan Kaufmann Publishers, New York, August 24-27, 1998, pp. 523–533.