DE GRUYTER OPEN

Print ISSN: 1312-2622; Online ISSN: 2367-5357
DOI: 10.1515/itc-2018-0009

Sorting without Moves by Counting

N. Vasilev, A. Bosakova-Ardenska

Key Words: *Sorting without moves; parallel sorting; Bubble sort; Insertion sort; Quick sort.*

Abstract. *This paper examines one algorithm for sorting a row without moves by counting. It is proposed a modification of this algorithm, which decreases the number of increments. The modification is named SWM. On the base of SWM it is proposed a parallel algorithm for sorting rows. The SWM algorithm is compared to Bubble sort, Insertion sort, Selection sort and Quick sort algorithms. The results show that proposed algorithm is faster than Bubble sort, Insertion sort and Selection sort but slower than Quick sort.*

1. Introduction

It is considered that near 25% of the work of the computer systems is sorting information [1]. This means that the searching of good methods and algorithms for sorting is very important. A lot of books and papers discuss advantages and applications of sorting algorithms [1,2,3,4,5,6]. They could be implemented iteratively or recursively. Most of sorting algorithms use iterative approach for sorting in place by execution of swap operations in different conditions [7,8,9,10,11,12,13]. The recursive approach is preferred for implementation of sorting algorithms which use divide-and-conquer technics, such as Quick sort and Merge sort [2,6,10,14,15]. According to the use of additional structures (arrays and others), sorting algorithms could be grouped into two major groups. In the first group fall algorithms which sort data in place, i.e. without any additional data structure, such as Bubble sort, Insertion sort, Selection sort, Quick sort and other. Some modifications of sorting algorithms which work in-place in order to decrease memory operation and energy consumption are developed last years [16,17]. The second group of algorithms includes sorting algorithms which use additional data structure. The benefit of these algorithms is that they minimize memory operations (swap operations). In this category are sorting algorithms such as Bucket sort, Radix sort, algorithms which use tables of left or/and right inversions and etc. [1,2,18,19,20].

The aims of this paper are:

– to propose algorithms for sorting rows without moves by counting;

– to implement the proposed algorithms;

– to evaluate and compare experimentally these algorithms with other algorithms.

2. Algorithm for Sorting of Rows without Moves by Counting

The proposed algorithm is as follows.

The array a[n] contains the initial data (elements).

The number of position of every element in a sorted row will be recorded in array p[n].

All initial values of the elements of the array p[n] are 1.

Each element a_i, i = 1, 2,…, n, is compared with the elements of the row on the right side of it.

If $a_i > a_k$, k = i+1, i+2,…, n, the value in p_j is increased with 1 (p_i++).

If $a_i \leq a_k$, k = i+1, i+2,…, n, the value in p_k is increased with 1 (p_k++).

The values p_j, j = 1, 2,…, n, show the position of the element aj in the ascending order.

Example 1

Sort in ascending order the row with 10 elements: 60, 80, 40, 30, 20, 90, 50, 100, 10, 70.

All initial values of the elements of the array p[10] are 1.

The first element a_0 = 60 is bigger than five elements on the right side of it: 40, 30, 20, 50 and 10. Then 5 times the value in p_0 is increased with 1. Thus the endmost value in p_0 is 1+5 = 6. This is the position of the first element in the sorted row.

The first element a_0 = 60 is less than 4 elements on the right side of it: 80, 90, 100 and 70. The values in the positions of these elements are increased with 1. So the value in p_1, p_5, p_7 and p_9 is 1+1 = 2.

The second element a_1 = 80 is bigger than 6 elements on the right side of it: 40, 30, 20, 50, 10 and 70. Then 6 times the value in p_1 is increased with 1. Thus the endmost value in p_1 is 2+6 = 8. This is the position of the second element in the sorted row.

The second element a_1 = 80 is less than 2 elements on the right side of it: 90 and 100. The values in the positions of these elements are increased with 1. So the value in p_5 and p_7 is 2+1 = 3.

The third element a_2 = 40 is bigger than three elements on the right side of it: 30, 20 and 10. Then 3 times the value in p_2 is increased with 1. Thus the endmost value in p_2 is 1+3 = 4. This is the position of the third element in the sorted row.

The third element a_2 = 40 is less than four elements on the right side of it: 90, 50, 100 and 70. The values in the positions of these elements are increased with 1. So the value in p_5 and p_7 is 3+1 = 4, in p_6 is 1+1 = 2 and in p_9 is 2+1 = 3.

The 4-th element a_3 = 30 is bigger than two elements on the right side of it: 20 and 10. Then 2 times the value in p_3 is increased with 1. Thus the endmost value in p_3 is 1+2 = 3. This is the position of the 4-th element in the sorted row.

The 4-th element a_3 = 30 is less than four elements on the right side of it: 90, 50, 100 and 70. The values in the positions of these elements are increased with 1. So the value in p_5 and p_7 is 4+1 = 5, in p_6 is 2+1 = 3 and in p_9 is 3+1 = 4.

The 5-th element a_4 = 20 is bigger than one element on the right side of it: 10. The value in p_4 is increased with 1. Thus the endmost value in p_4 is 1+1 = 2. This is the position of the 5-th element in sorted the row.

The 5-th element a_4 = 20 is less than four elements on the right side of it: 90, 50, 100 and 70. The values in the po-

sitions of these elements are increased with 1. So the value in p_5 and p_7 is 5+1 = 6, in p_6 is 3+1 = 4 and in p_9 is 4+1 = 5.

The 6-th element a_5 = 90 is bigger than three elements on the right side of it: 50, 10 and 70. Then 3 times the value in p_5 is increased with 1. Thus the endmost value in p_5 is 6+3 = 9. This is the position of the 6-th element in the sorted row.

The 6-th element a_5 = 90 is less than one element on the right side of it: 100. The value in the position of this element is increased with 1. So the value in p_7 is 6+1 = 7.

The 7-th element a6 = 50 is bigger than one element on the right side of it: 10. The value in p_6 is increased with 1. Thus the endmost value in p_6 is 4+1 = 5. This is the position of the 7-th element in the sorted row.

The 7-th element a_6 = 50 is less than two elements on the right side of it: 100 and 70. The values in the positions of these elements are increased with 1. So the value in p_7 is 7+1 = 8, and in p_9 is 5+1 = 6.

The 8-th element a_7 = 100 is bigger than two elements on the right side of it: 10 and 70. Then 2 times the value in p_7 is increased with 1. Thus the endmost value in p_7 is 8+2 = 10. This is the position of the 7-th element in the sorted row.

The 8-th element a_7 = 100 haven't less elements on the right side of it. The values in the positions of the elements on the right are not changed.

The 9-th element a_8 = 10 haven't bigger elements on the right side of it. The value 1 in p_8 isn't changed. The position of a_8 in sorted row is 1.

The 9-th element a_8 = 10 is less than the last elements a_9 on the right of it. The value in position c_9 is increased with 1 – 6+1 = 7.

The position of the 10-th element is 7. The value in c_9 is increased with 1 for each element less than element a_9.

Table 1. Finding the positions of the elements 60, 80, 40, 30, 20, 90, 50, 100, 10, 70 in the sorted row

Positions **j**	0	1	2	3	4	5	6	7	8	9
Values of $\mathbf{a_j}$	60	80	40	30	20	90	50	100	10	70
Initial values	1	1	1	1	1	1	1	1	1	1
Compares of $\mathbf{a_0}$	6 (5)	2	1	1	1	2	1	2	1	2
Compares of $\mathbf{a_1}$		8 (6)	1	1	1	3	1	3	1	2
Compares of $\mathbf{a_2}$			4 (3)	1	1	4	2	4	1	3
Compares of $\mathbf{a_3}$				3 (2)	1	5	3	5	1	4
Compares of $\mathbf{a_4}$					2 (1)	6	4	6	1	5
Compares of $\mathbf{a_5}$						9 (3)	4	7	1	5
Compares of $\mathbf{a_6}$							5 (1)	8	1	6
Compares of $\mathbf{a_7}$								10 (2)	1	6
Compares of $\mathbf{a_8}$									1 (0)	7
Positions $\mathbf{p_j}$ in sorted row	**6**	**8**	**4**	**3**	**2**	**9**	**5**	**10**	**1**	**7**

Table 1 shows finding the positions of the elements of given row in sorted row. Each row of the table shows the current values of the positions p of the elements after termination of the compares of the current element. The cells in black contain the positions of the elements in the sorted row. In parentheses are written the numbers of elements less than corresponding element.

The number of operations for finding the positions of the elements of the given row in sorted increased row is: 45 compares and 45 increments (records).

Table 2 shows the sorted row.

Table 2. Positions of the elements in sorted row and the sorted row

Positions **j**	0	1	2	3	4	5	6	7	8	9
Values of $\mathbf{a_j}$	60	80	40	30	20	90	50	100	10	70
Positions $\mathbf{p_j}$ in sorted row	**6**	**8**	**4**	**3**	**2**	**9**	**5**	**10**	**1**	**7**
Sorted row	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**

The operations which are used in proposed algorithm are as follows:

1) compares of the elements;

2) records (increments) after comparing the current element with the elements on the right side;

3) moves the elements in the positions of the sorted row.

The number of the compares is: n(n-1)/2.

The number of records (increments) is: n(n-1)/2.

The number of moves for building the sorted row is: n.

If t_1 is time to execute the operation compare, t_2 is time to execute the operation record (increment), t_3 is time to execute the operation move (record), the time for building the sorted row is

$T = (t_1+t_2)n(n-1)/2+t_3n$.

3. Modification of the Proposed Algorithm

When the position p_i, i< n, of the element ai in sorted row is defined it is obviously that the positions of the elements $a_k \geq a_i$, k=i+1, i+2,…, n, can't be less than p_i+1. Therefore the value p_i+1 can be assigned as current to the positions p_κ: $p_k = p_i+1$ when $a_k \geq a_i$.

When we find the position of the current element p_i, i = 1, 2,…, n, and after assignments $p_k = p_j+1$ if $a_k \geq a_i$, k = i+1, i+2,…, n, the set of elements (without the elements of which the positions in sorted row are defined) is divided to non-intersecting subsets. The elements of each of these subsets have the same value of the current position.

This value is minimal value of position from the positions of the elements of given subset in sorted row.

The maximal possible value of position of the elements of given subset in sorted row is p_q–2, where p_q is the most near bigger minimal value of position of the elements of the set.

Each subset is information independent in the following steps of the sorting.

Thus each value p_i of the element a_i, i = 1, 2,…, n, in sorted row is computed as:

– the value of p_i, i = 1, 2,…, n–1 is assigned to d;

– the elements a_k, k = i+1, i+2,…, n, less than or equal to a_i ($a_i \geq a_k$) with positions p_k = d are counted: p_i++;

– the indices of the elements $a_k > a_i$, k = i+1, i+2,…, n, with p_k = d are stored;

– after finding the position p_i (new value) to the positions of $a_k > a_i$, k = i+1, i+2,…, n, with p_k = d (prior value of p_i) are assigned the new value p_i+1 (these are positions of the elements of which the indices are stored).

Thus (p_k – d) increments will be avoided. But we must store the indices of the elements $a_k > a_i$, k = i+1, i+2,…, n, with $p_k = p_i$ (prior value) and we must compare these elements with element a_i.

Notice that in comparison with the basic algorithm in its modification the signs '≥' and '<' are exchange their places when we compare the elements ai and ak. Thus the number of stored indices is decreased – the indices of elements ak equal to ai are not stored.

The algorithm for building the sorted ascending row which realizes described above considerations is as follows.

The array a[n] contains the initial data.

The number of position of every element in a sorted row will be recorded in array p[n].

The array b[n] contains the indices of the elements $a_k > a_j$, k=j+1, j+2,…, n, with $p_k = p_j$ (prior value).

All initial values of the elements of the array p[n] are 1.

1. i = 0.

2. k = i+1, j = 0, d = p[i].

3. If d = p(k) go to step 4. Otherwise go to step 5.

4. If a[i] ≥ a[k], p[i]++ and go to step 5. Otherwise b[i] = k, j++ and go to step 5.

5. If k > n–1, r = 0 and go to step 6. Otherwise k++ and go to step 3.

6. If r < j, p[b[r]] = p[i]+1 and go to step 7. Otherwise go to step 8.

7. r++ and go to step 6.

8. If i > n–1, end. Otherwise i++ and go to step 2.

The values of the elements p_j, j=1,2,…,n, show the positions of the elements a_j in sorted ascending row. The flowchart of the modified algorithm is shown on fig.1 and the proposed modified algorithm is illustrated with the example 1 in *table 3*.

We will explain the computation for the values in the row of the element a_2.

$p_2 = 1$ in the previous row (the row of the element a_1 = 80) is compared with p_i, $i = \overline{3,9}$ of the elements on the right side of it.

4 from them are $p_3=p_4=p_6=p_8=1=p_2$.

a_2=40 is compared with a_3=30, a_4=20, a_6=50, a_8=10.

The elements 30, 20, 10 are less than 40. p_2 is incremented 3 times. The value of p_2 in the row of a_2 is 4. This is the position of the element 40 in sorted ascending row.

The element 50 is bigger than 40. Its index 6 is stored.

The value p_3+1 = 4+1 = 5 is assigned to p_6 in the row of a_2: p_6 = 5.

The other values of p_5, p_7, p_9 in the row of a_2 are the same as in the row of a_1.

The number of operations for finding the position of the element a_2 = 40 is: 4 compares, 3 records (increments), 1 record (assignment), 1 record of index.

Table 3. Finding the positions of the elements 60, 80, 40, 30, 20, 90, 50, 100, 10, 70 in sorted ascending row

Positions **j**	0	1	2	3	4	5	6	7	8	9
Values of $\mathbf{a_j}$	60	80	40	30	20	90	50	100	10	70
Initial values	1	1	1	1	1	1	1	1	1	1
Compares of $\mathbf{a_0}$	6 (5)	7	1	1	1	7	1	7	1	7
Compares of $\mathbf{a_1}$		8 (1)	1	1	1	9	1	9	1	7
Compares of $\mathbf{a_2}$			4 (3)	1	1	9	5	9	1	7
Compares of $\mathbf{a_3}$				3 (2)	1	9	5	9	1	7
Compares of $\mathbf{a_4}$					2 (1)	9	5	9	1	7
Compares of $\mathbf{a_5}$						9 (0)	5	10	1	7
Compares of $\mathbf{a_6}$							5 (0)	10	1	7
Compares of $\mathbf{a_7}$								10 (0)	1	7
Compares of $\mathbf{a_8}$									1 (0)	7
Positions $\mathbf{p_j}$ in sorted row	**6**	**8**	**4**	**3**	**2**	**9**	**5**	**10**	**1**	**7**

Let compare the number of operations of the example 1 in both algorithms.

The number of operations of the basic algorithm is: 45 compares and 45 increments.

Total number of operations is 90.

The number of operations of the modified algorithm is:

– 45 compares: 9 compares of the element a0 with the other elements and 36 compares of the current position of each of the other elements with the current positions of the elements on the right side of it;

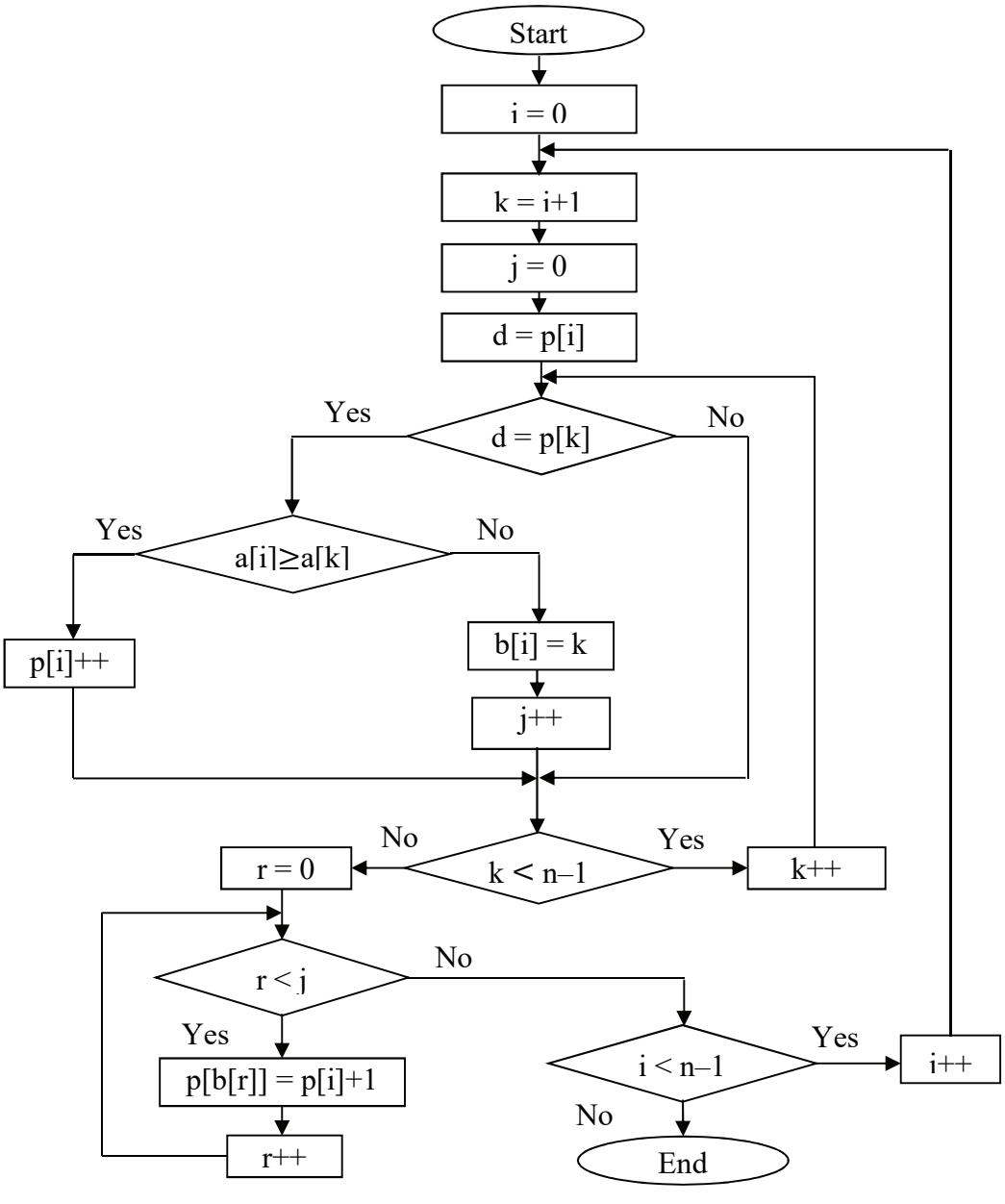


Figure 1. Flowchart of the modified algorithm

– 12 increments of the positions of the current elements;

– 11 compares of the current element with the elements of its subset;

– 8 records of the indices of the elements of the subset bigger than current element;

– 8 assignments of new values to the current positions of the elements bigger than current element.

Total number of operations is 84.

4. Parallel Modified Algorithm

As it is mentioned after finding the position of the current element p_j, j=1, 2,…, n, and assignments $p_k = p_j+1$ when $a_k \geq a_j$, k=j+1, j+2,…, n, the set of the elements (without the elements which positions in sorted row are defined) is divided to t, t≤j+1, non-intersecting subsets. The elements of each of these sets have the same value of the current position.

Each subset is information independent in the following steps of the sorting. In other words the task is decomposed to separated sub tasks. Their number is equal to the number of the sub sets. The sorting in all sub sets can be done simultaneously (in parallel). After finding q (q≤k) different values of the current positions the computation can continue with simultaneous sorting of the q subsets.

Therefore after finding q subsets the sorting of given row can be executed with parallel algorithm. The sorting continues in each sub row of the row. The main processor of parallel computer system (PCS) defines q sub sets (each with the same values of current positions) and transfer the elements of each subset to one from other processors of the

PCS. Each of these processors sorts elements of its subset. The main processor sort one of the subsets too.

Each processor which is sorts one of the subsets also can transfers the elements of some subsets of its subset to other processors of the PCS.

Thus computation will be accelerated because of simultaneously sorting of the sub rows.

Example 2

Sort in ascending order the row with 10 elements: 50, 20, 80, 40, 70, 10, 100, 90, 30, 60.

After finding the position 5 of the number 50 the elements of the row are divided in 2 sub sets: 20, 40, 10, 30 with current position 1 and 80, 70, 100, 90, 60 with current position 6. The elements of the first sub set are less than 50 and their positions can't be bigger than 5. Both sub rows are independent and they can be computed in parallel.

The positions of the elements 20 and 80 can be computed simultaneously. They are respectively 2 and 8. The independent sub rows are already 4: 40, 30 with current position 3; 70 and 60 with current position 6, 10 with current position 1; 100 and 90 with current position 9. The sub row with current position 1 has one element – 10. So this position is its position in sorted row.

The positions of the elements 40, 70 and 100 can be found simultaneously. Their positions are respectively 4, 7 and 9. The other elements are 30, 60 and 90. They are in different sub rows. So their positions in sorted row are respectively 3, 6 and 9. The sorting is ended.

Table 4 illustrates the parallel algorithm for the row in example 2.

It is seen that the consecutive algorithm is executed for (n–1) steps. In each step is found the position of one element. The row in example 1 is executed for 9 steps.

Table 4. Parallel computation the positions of the elements of the row 50, 20, 80, 40, 70, 10, 100, 90, 30, 60 in sorted increased row

Positions **j**	0	1	2	3	4	5	6	7	8	9
Values of $\mathbf{a_j}$	50	20	80	40	70	10	100	90	30	60
Initial values	1	1	1	1	1	1	1	1	1	1
Compares of $\mathbf{a_0}$	5 (4)	1	6	1	6	1	6	6	1	6
Compares of $\mathbf{a_1}$, $\mathbf{a_2}$		2 (1)	8 (2)	3	6	1	9	9	3	6
Compares of $\mathbf{a_3}$, $\mathbf{a_4}$, $\mathbf{a_6}$				4 (1)	7 (1)		10 (1)	9	3	6
Positions $\mathbf{p_j}$ in sorted row	**5**	**2**	**8**	**4**	**7**	**1**	**10**	**9**	**3**	**6**

The row in example 2 is executed for 3 steps with the parallel algorithm:

– in first step is found position of element 50;

– in second step are found positions of elements: 20, 80, 10;

– in third step are found positions of elements: 40, 70, 100, 30, 60, 90.

5. Experimental Results

The proposed modified algorithm for sorting without moves is implemented on C++, MS Visual Studio 2005 Version 8. For experiments is used computer system with processor Intel Celeron E3300 2,5GHz, RAM 3 GB. The elements of the rows are generated by the function rand() and srand(). The number of the elements of the rows for these experiments is from 1000 to 10000.

We named the proposed algorithm SWM. SWM and four known algorithms sort 10 times 10 different rows. The execution time is the averaged value.

Table 5 and *figure 2* show the results from experiment.

Table 5. Averaged execution time

Number of the elements / Algorithms	1000	2000	3000	4000	5000	6000	7000	8000	9000	10 000
Bubble sort	0,0143	0,0498	0,1062	0,1887	0,2957	0,4275	0,5717	0,7437	0,9671	1,183
Insertion sort	0,0014	0,0053	0,0121	0,0204	0,0327	0,0474	0,0632	0,0821	0,1093	0,1277
Selection sort	0,006	0,0243	0,0534	0,0948	0,1486	0,2052	0,2833	0,367	0,4694	0,5909
Quick sort	0,0002	0,0004	0,0008	0,0008	0,0008	0,001	0,0011	0,0016	0,0018	0,002
SWM	0,0011	0,003	0,0064	0,0115	0,0172	0,0251	0,0348	0,0463	0,0532	0,072

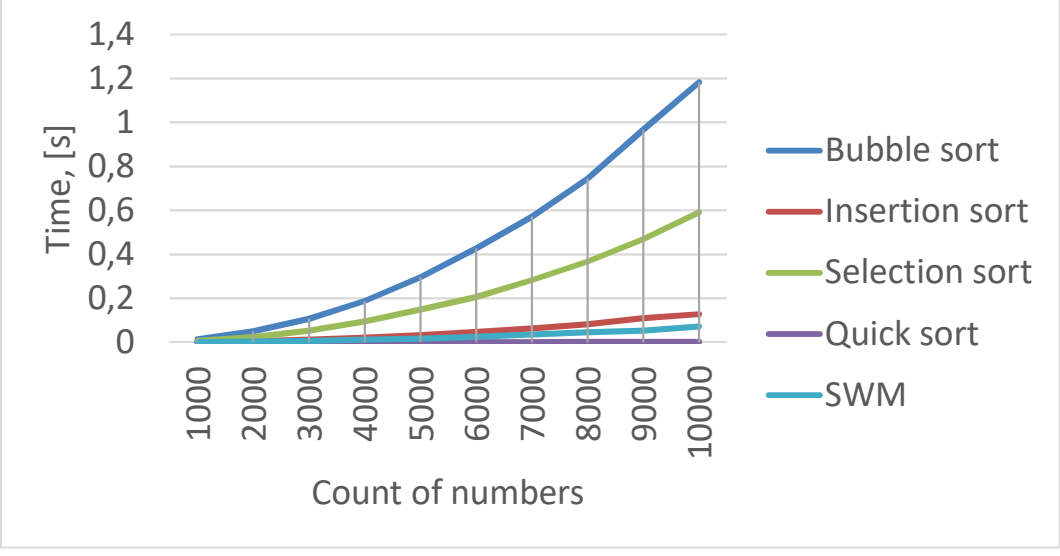


Figure 2. Experimental results

6. Conclusion

Experimental results show that algorithm SWM is not competitive to the best algorithm for sorting (quick sort). But it is faster than Bubble sort, Insertion sort and Selection sort. SWM algorithm has one valuable feature: after finding the position of some element in sorted row the rest elements are divided into information independent sub sets. Therefore after finding the position of q element in sorted row, q<<n, the sorting in all sub sets can be done simultaneously (in parallel) with one of the most algorithms.

The future work will continue with:

1) analysis of the complexity of the proposed algorithm SWM;

2) implementation of proposed parallel algorithms SWM;

3) implementation of parallel algorithms on the base of SWM and quick sort.

Acknowledgment

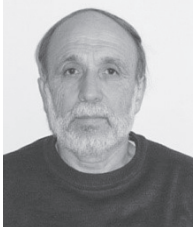
This research was supported by University of Food Technologies according 16/18-H UFT Plovdiv research item.

References

1. Knuth, D. The Art of Computer Programming, V3. Sorting and Searching. Addison Wesley Publishing Company, 1973.
2. Stoichev, S. Synthesis and Analysis of Algorithms. BPS, Sofia, 2003 (in Bulgarian).
3. Nakov, P., P. Dobrikov. Programming=++Algorithms. TopTeam Co, 2003 (in Bulgarian).
4. Sedgewick, R. Algorithms. Addison Wesley Publishing Company, 1984.
5. Wirth, N. Algorithms+Data Structures=Programs. Prentice-Hall, 1976.
6. Cormen, T. H., Ch. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms. Second Edition, MIT Press, 2001.
7. Elkahlout, A. & A. Maghari. A Comparative Study of Sorting Algorithms Comb, Cocktail and Counting Sorting. – *International Research Journal of Engineering and Technology* (*IRJET*), 4, 2017, 1387-1390.
8. Hammad, J. A Comparative Study between Various Sorting Algorithms. – *IJCSNS International Journal of Computer Science and Network Security*, 15, 2015, No. 3, 11-16.
9. Sarvjeet, S., S. Kaur. Freeze Sorting Algorithm Based on Even-Odd Elements. – *IOSR Journal of Engineering*, 4, 2014, No. 3, 18-23.
10. Alnihoud, J. & R. Mansi. An Enhancement of Major Sorting Algorithms. – *International Arab Journal of Information Technology*, 7, 2010, No. 1, 55-62.
11. Khairullah, Md. Enhancing Worst Sorting Algorithms. – *International Journal of Advanced Science and Technology*, 56, 2013, 13-26.
12. Paira, S., A. Agarwal, Sk. Alam Sk, S. Chandra. Doubly Inserted Sort: A Partially Insertion Based Dual Scanned Sorting Algorithm. ERCICA'15, 2015, DOI: 10.1007/978-81-322-2550-8_2.
13. Mundra, J., B. Pal. Minimizing Execution Time of Bubble Sort Algorithm. – I*nternational Journal of Computer Science and Mobile Computing*, 4, 2015, No. 9, 173-181.
14. Aremu, D., O. Adesina, O. Makinde, O. Ajibola, O. Agbo-Ajala. A Comparative Study of Sorting Algorithms. – *African Journal of Computing & ICT*, 6, 2013, No. 5, 199-206.
15. Hoare, C. A. R. Quicksort. – *The Computer Journal*, 5, 1962, 10-16.
16. Xiao, Li, X. Zhang, S. Kubricht. Improving Memory Performance of Sorting Algorithms. – *Journal of Experimental Algorithmics*, 5, 2000, Paper 3, DOI: https://doi.org/10.1145/351827.384245.
17. Bunse, Ch., H. Höpfner, S. Roychoudhury, E. Mansour. Choosing the "Best" Sorting Algorithm for Optimal Energy Consumption. ICSOFT 2009 – 4th International Conference on Software and Data Technologies, 2009, Proceedings 2, 199-206.
18. Vasilev, N., A. Bosakova-Ardenska. Algorithms for Sorting by Left Inversions Table. – *International Review on Computers and Software (IReCoS)*, 7, 2012, No. 2 – Part A, 642-650, ISSN 1828-6003.
19. Vasilev, N., A. Bosakova-Ardenska. A New Sorting Algorithm

with Filling to the Left and Right. – *Balkan Journal of Electrical & Computer Engineering*, 3, 2015, No. 3, 135-141, ISSN: 2147-284X.
20. Vasilev, N., A. Bosakova-Ardenska, N. Shopov. Sorting Data with Tables of Inversions without Moves. Indusrtry 4.0. Business Environment. Quality of Life, 26-28 October 2017, Ruse, 86-91, ISBN 978-954-712-733-3.

Manuscript received on 07.04.2018



Nayden Vasilev *is Associate Professor in Department of Computer Systems and Technologies at Technical University of Plovdiv. He receives PhD in 1976 in Moscow, Russia. Assoc. Prof. Nayden Vasilev was a head of courses in the field of computer technologies such as Operating systems, Discrete Structures, Programming and Computer Applications, Parallel algorithms and etc. His research interests include parallel algorithms, discrete mathematics and music.*

Contacts:
e-mail: mnvasilev@yahoo.com

Atanaska Bosakova-Ardenska *was born in 1980. She received the M.Sc. degree of Computer Systems and Technologies at Technical University of Sofia, Plovdiv Branch 2004. She receives PhD in 2009. From 2014 she is Associated Professor in Department of Computer Systems and Technologies at University of Food Technologies. Her research interests include parallel algorithms, image processing, MPI (Message Passing Interface), C++ and Java programming.*

Contacts:
e-mail: a_bosakova@uft-plovdiv.bg