Modified Merge Sort Algorithm for Large Scale Data Sets

Marcin Woźniak[1], Zbigniew Marszałek[1],
Marcin Gabryel[2], and Robert K. Nowicki[2]

[1] Institute of Mathematics, Silesian University of Technology,
ul. Kaszubska 23, 44-100 Gliwice, Poland
[2] Institute of Computational Intelligence, Czestochowa University of Technology,
Al. Armii Krajowej 36, 42-200 Czestochowa, Poland
{Marcin.Wozniak,Zbigniew.Marszalek}@polsl.pl,
{Marcin.Gabryel,Robert.Nowicki}@iisi.pcz.pl

Abstract. Sorting algorithms find their application in many fields. One of their main uses is to organize databases. Classical applications of sorting algorithms often can not cope satisfactorily with large data sets or with unfavorable poses of sorted strings. Typically, in such situations, we try to use other methods or apply sorting process to reshuffled input data. Unfortunately, this approach complicates sorting process and often results in significant prolongation of the time. In this paper, the authors examined an algorithm dedicated to the problem of sorting large scale data sets. In the literature, there are no studies of such examples. These studies will allow to describe the properties of sorting methods for large scale data sets. Performed tests have shown superior performance of the examined algorithm, especially for large scale data sets. Changes sped up sorting of data with any arrangement of the input elements.

Keywords: computer algorithm, data mining, data sorting, analysis of computer algorithms.

1 Introduction

In the literature [4, 11, 12, 24] are shown classic versions of the merge sort algorithm which are using recursion operation. The classic solution is not optimal and many researchers have proposed modifications and extensions of the basic sorting algorithms. The authors of [3, 6, 9, 10] presented the possibility of multi-threading sort algorithms. However, the authors of [2, 7, 13] presented a comparison of properties of different algorithms. The works [14, 15, 18, 21] described the impact of the algorithm on memory resources. In the paper [23] authors showed the ability to create hybrid algorithms and their implementations. The authors of the papers [8, 25–27] described the possibilities of a special matching merge algorithm. The results of the optimization algorithm for large data sets are described by the authors of [6, 21]. Recursive solutions have some limitations. The use of recursion increases time-consuming operations due to

L. Rutkowski et al. (Eds.): ICAISC 2013, Part II, LNAI 7895, pp. 612–622, 2013.

the need to operate on the stack. Processing of large collections is related to the high number of levels of recursion, which can lead to an internal stack overflow. The authors of the present study examined the merge sort algorithm for large data sets. Presented method performs the sorting process without using recursion, which increases stability and reduces the execution time. In the examined method recursive procedure for the construction of the stack was replaced by smart stacking elements into a sequence.

1.1 Classic Merge Sort Algorithm

First versions of the merge sort algorithm were published by the authors of [1, 12]. Classical algorithm known also from the literature [4, 24] splits the input string into two substrings, sorts them recursively and then merges. It uses an additional array in which are put further substrings of numbers. The procedure merges two sequences of numbers $a_p \leq \ldots \leq a_q$ and $a_{q+1} \leq \ldots \leq a_r$ stored in the array a from index p to q and from $q+1$ to r. Merging of substrings starts at the element with index $\lfloor \frac{p+r}{2} \rfloor$. Unfortunately, recursive methods, while operating on the system stack, use significantly system resources and prevent proper assessment of actual computational complexity. Moreover, in practice, for very large input strings, system stack overflow occurs frequently which leads to the suspension of the program. The solution is to use hybrid methods described in the literature, for example in [23]. However more effective solution is to use the directly acting method, which is the topic of this paper.

2 Examined Merge Sort Algorithm

The authors examined non recursive merge sort algorithm dedicated to large collections of data. Performed tests showed that changes have improved the algorithm stability measured as a reduction in dispersion (standard or average deviation) of variability of the number of clock cycles during sorting of any initial configuration of the input data. Recursive procedure for the construction of the stack, known from the literature [11, 12], was replaced by stacking items in the substrings queue. The examined algorithm has been designed to increase the efficiency of sorting large scale data sets as the algorithm based on two component algorithms. The first one is merging sorted items into the stack. The second component is a merge sorting procedure. In the first step we compare subsequent elements of the strings. As a result of the first step, we receive stacks with a given number of elements. In the second step we merge received stacks. The algorithm performs merging until there is only one present stack. Sorting algorithm will perform the procedure a sufficient number of times until there is a completely sorted input string.

2.1 Double Merge Sort Algorithm

The examined algorithm in the first step begins with a comparison of the elements in pairs of the input string $a_0, \ldots, a_{n-1}$. In this way we obtain a sequence

of pairs of elements. In the second step we merge sequences received in the first step. As a result of this operation we always receive strings containing double number of the elements. We merge these until there is only one string. If the initial string contains an odd number of elements it is sufficient to rewrite the last element till the last step in the algorithm, as shown in Fig.1. In the last step we merge this item with the string and receive completely sorted input data. Merging method receives as the input two numerical sequences $x_0 \leq \ldots \leq x_{m-1}$

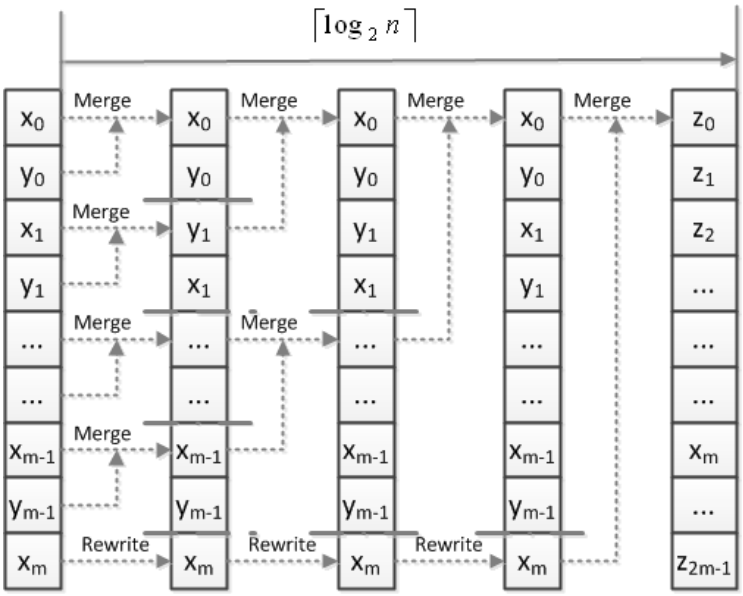

Fig. 1. An example of merging with the examined algorithm

and $y_0 \leq \ldots \leq y_{m-1}$ sorted in the previous step. It returns sorted sequence of numbers $z_0 \leq \ldots \leq z_{2m-1}$. In the following steps of the algorithm, the next element of the output string is given by selecting the smallest element of merged strings X and Y. If it happen to have the equal elements in both sequences X and Y we assume that the next selected element is the one of the string X. Due to the fact that X and Y are already sorted, we can perform merging with not more than $2m-1$ comparisons, where $2m$ is the sum of lengths of strings X and Y. The examined algorithm is based on two components presented in Algorithm 1 and Algorithm 2.

2.2 Double Merge Sort Algorithm Time Complexity

Presented algorithm of double merge sorting has time complexity described on the base of Theorem 1.

```
Begin
Load numerical sequences
Count the number of elements and write it in variables c1 i c2
1 if Strings contain elements then
    if Element of the first string is smaller than the first element of second
    string then
        Write element of the first string into the sorted string
        Cross out this item from the first string
        Decrease the number of elements in the first string
        Go to 1
    else
        Write element of the second string into the sorted string
        Cross out this item from the second string
        Decrease the number of elements in the second string
        Go to 1
    end
else
    if The first string contains elements then
        Write elements of the first string into the sorted string
        Stop
    else
        Write elements of the second string into the sorted string
        Stop
    end
end
```

Algorithm 1: Algorithm to merge sequences

Theorem 1. *Double merge sorting algorithm has time complexity*

$$T_{avg}(n) = O(n \cdot \log_2 n). \tag{1}$$

Proof. Suppose we want to sort string composed of n elements. Performing examined double merge sort algorithm we compare in each step sequences arranged in the previous step. In the first step we merge 1-element sequences which we compare in successive pairs. Thus, in this iteration, we make no more than n comparisons. In the next iteration we merge strings from the first step. As a result of the merge operation in step two we have in order strings of doubled elements. Such an arrangement is obtained as a result of no more than n comparisons. Merging continues in subsequent iterations respectively comparing each time strings of a double length, resulted from consolidation in the previous step. If we compare two sequences of length m we will do no more than $2m - 1$ comparisons. Every time by merging such two strings we obtain a double sequence. To sort the input string we will do k merge steps in the algorithm, and in each of them we do no more than n comparisons. Without loss of generality, we can

```
   Start
   Load sorted string into a table a
   Count number of elements and write it in the variable n
   Set logical variable t of direction on true
   Create space for array b with size of the number of sorted elements n
   Set size of merged series as 1 and write it in a variable m
1  if Size of merged series is smaller than n then
       Set index of first merged string as 0 and write it in variable i
2      if i is smaller than n then
           Set index pb of merged string on i
           Set index p1 of first merged string on i
           Set index p2 of second merged string on i + m
           if index of second string p2 is smaller or equal to n then
               Set number of elements in first string c1 as m
               Set number of elements in second string c2 as n − p2
               if Number of elements in second string is bigger than m then
                   Set number of elements of second string c2 as m
                   Go to 3
               else
                   Go to 3
               end
3              if Check if logical variable t is set on true then
                   Proceed Algorithm to merge sequences from array a and
                   write them into array b
                   Increase index i of first string by 2 ∗ m
                   Go to 2
               else
                   Proceed Algorithm to merge sequences from array b and
                   write them into array a
                   Increase index i of the first string by 2 ∗ m
                   Go to 2
               end
           else
               Increase index i of the first string by 2 ∗ m
               Go to 2
           end
       else
           Change logical variable t of direction to opposite
           Double size of merged series
           Go to 1
       end
   else
       if Negation of direction variable is set on true then
           Copy elements from array b to array a
           Stop
       else
           Stop
       end
   end
```

Algorithm 2: Algorithm of merge sorting

assume that the sorted sequence of n elements has approximately 2^k compared elements. Thus, it is reasonable to estimate the following

$$\min_{k \in N} 2^k \geq n. \tag{2}$$

Logarithms both sides (2) we obtain

$$\min_{k \in N} \log_2 2^k \geq \log_2 n. \tag{3}$$

Thus, on the basis of a logarithmic function we obtain the following inequality

$$\min_{k \in N} k \cdot \log_2 2 \geq \log_2 n. \tag{4}$$

Finally, we can assume that the number of operations performed by sorting the string will be

$$k = \lceil \log_2 n \rceil . \tag{5}$$

So time complexity is

$$T_{avg}(n) = n \cdot k = n \cdot \log_2 n, \tag{6}$$

which completes our discussion. □

2.3 Results of Egzaminations

The examined algorithm for sorting by double merging has been tested to evaluate it's performance for large scale data sets. The method was programmed using standard CLR in MS Visual Studio 2010 Ultimate. To tests were taken random samples of 100 series in each class of frequencies, including unfavorable positioning. Tests were carried out on processor i7 series. During performed tests were analyzed CPU clock cycles. Statistical analysis of the results is presented in Table 1. The speed of performed operations depends on the initial arrangement of the input data. It is also affected by the number of changes made at a constant number of comparisons. In this study, the authors have examined the values shown in Table 1. Experiments were performed for a sorted string, inversely sorted string and other numerical sequences taken at random. Table 1 shows statistical study of the CPU clock cycles during the tests of the algorithm described in Section 2.1. Results presented in Table 1 were plotted. On the charts are also plotted results of similar experiments carried out for the base algorithm. The aim of the analysis and comparison is to verify the thesis that the changes improved large scale data sets sorting. In the examinations and tests were compared the characteristics of the examined algorithm with the version known from the literature [4, 11, 12, 24]. The analysis of Figure 2 shows that the examined method behaves similarly to the classic version of the algorithm. Size of the average number of CPU clock cycles shown in Figure 2 is related to the standard deviation shown in Figure 3. Charts shown in Figure 3 show the comparison of the characteristics of the standard deviation of CPU clock cycles.

Table 1. Table of CPU clock cycles of the examined double merge sorting algorithm

cpu tics [ti]	Number of sorted elements			
	100	1000	10000	100000
avg	2298, 4	2987, 6	8631, 2	32754, 4
std deviation	801, 397	265, 39	2619, 49	10721, 64
avg deviation	472, 16	215, 52	2139, 44	9266, 08
coef. of variation	0, 35	0, 09	0, 30	0, 33
var. area upper end	1497	2722, 21	6011, 71	22032, 76
var. area lower end	3099, 8	3252, 99	11250, 69	43476, 04
	1000000	10000000	100000000	
avg	363782, 8	4353284	48195771, 8	
std deviation	125670, 76	1384543, 28	15067068, 72	
avg deviation	110078, 16	1211305, 2	13203157, 76	
coef. of variation	0, 34	0, 32	0, 31	
var. area upper end	238112, 04	2968740, 72	33128703, 08	
var. area lower end	489453, 56	5737827, 28	63262840, 52	

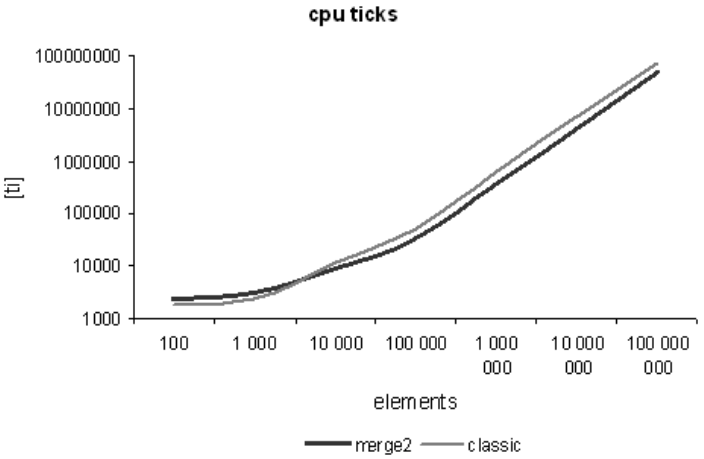

Fig. 2. Comparing the values of the characteristics of CPU clock cycles

Graphs shown in Figure 3 show that the examined method has a positive influence on sorting large scale data sets. At the same time the algorithm described in Section 2.1 is characterized by increased stability. Coefficients of variation were approximied by polynomial approximation, which is shown in Figure 4. The resulting curves characterize variability of characteristics of merge sort algorithm. An analysis of the chart shown in Figure 5 shows that classical form is efficient only for series smaller than 1000 elements. Above this number of items, described in Section 2.1 algorithm runs about 40% faster than the classic version. At the same time a comparison of the values shown in Figure 3 shows that it is working with more stability than the classic version. Figure 5 shows that the examined

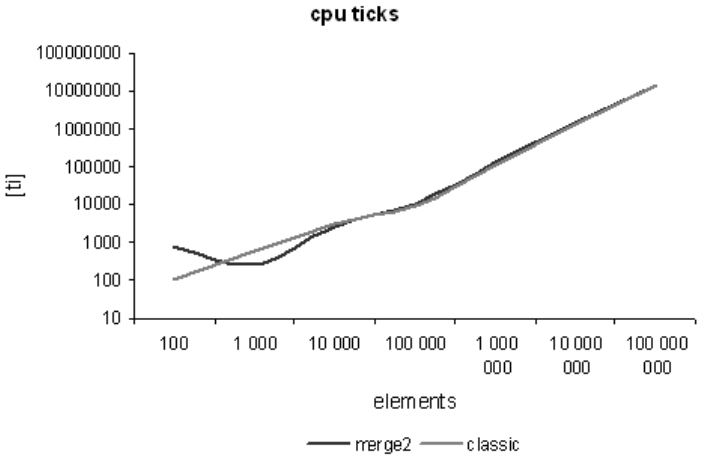

Fig. 3. Comparing the values of standard deviation of CPU clock cycles

Table 2. Comparative table of CPU clock cycles variations of the classic and the examined algorithm

cpu tics [ti]	Number of sorted elements						
	100	1000	10000	100000	1000000	10000000	100000000
classic	0, 05	0, 25	0, 24	0, 18	0, 36	0, 2	0, 19
merge2	0, 35	0, 09	0, 3	0, 33	0, 34	0, 32	0, 31

double merge sort can effectively sort any set of elements. Simultaneously, as shown by the earlier analysis, it has greater stability. The analysis and comparison of charts in Figure 5 shows that the examined method can sort efficiently any set of elements, which confirms previous conclusions. Described in Section 2.1 algorithm behaves more stable and sort quickly any collection of large data regardless of the orientation of sorted items.

3 Final Remarks

In conclusion, presented double merge sort algorithm has a good stability for large scale data sets. It is also a fast version for sorting large input data of any arrangement of the elements. For described in Section 2.1 algorithm, during the experiments there were no difficulties in sorting adverse poses of elements in the input series. The presented method allows sorting any data sets and increases the stability in comparison to the classical form. This effect is mainly due to the lack of handling the return stack. Thus, the examined algorithm appears to be appropriate for use for large scale data sets. However authors consider whether it is possible to increase the efficiency of sorting large data sets. Further research and work will focus on improving the speed and stability. In further

work on increasing the efficiency the authors plan to focus on building and improving multiple merging algorithm. An important change may be to abandon the classical method 'divide and conquer' for increased distribution of the stack.

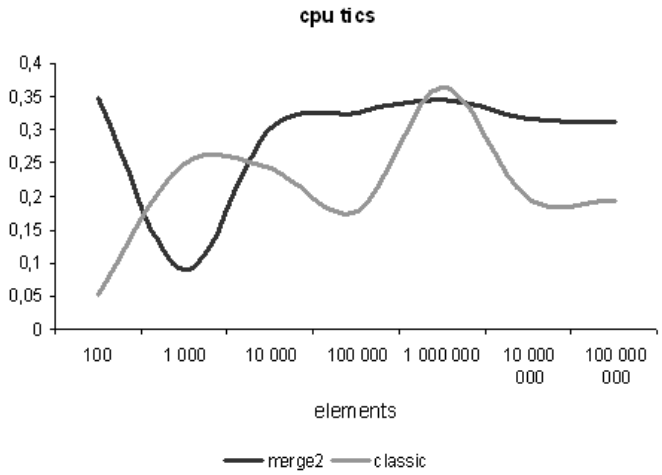

Fig. 4. Comparing the values of expected variability of CPU clock cycles

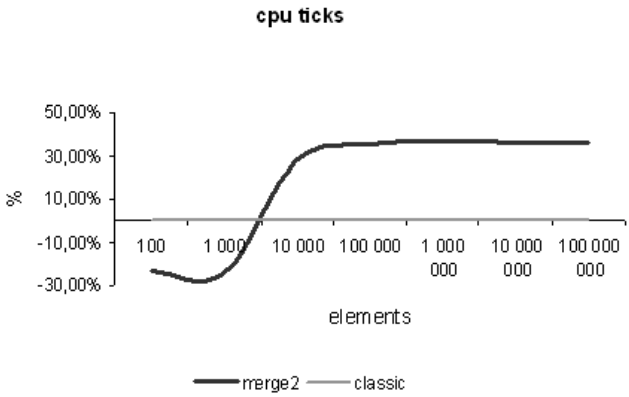

Fig. 5. Illustration of the percentage difference of CPU clock cycles between classic version and examined algorithm

Described in Section 2.1 double merge sort algorithm is a stable method, especially for collections of over 10000 elements, as shown in Figure 4. The question is, what is the difference between the classic version and described in Section 2.1 algorithm. Comparison of the characteristics is illustrated in Figure 5. As the reference method is chosen classic algorithm.

References

1. Aho, I.A., Hopcroft, J., Ullman, J.: The design and analysis of computer algorithms. Addison-Wesley Publishing Company, USA (1975)
2. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine CM-2. In: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1991), Hilton Head, South Carolina, pp. 3–16 (July 1991)
3. Cole, R.: Parallel merge sort. SIAM J. Comput. 17, 770–785 (1988)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. The MIT Press and McGraw-Hill Book Company, Cambridge (2001)
5. Crescenzi, P., Grossi, R., Italiano, G.F.: Search data structures for skewed strings. Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland. In: Jansen, K., Margraf, M., Mastrolli, M., Rolim, J.D.P. (eds.) WEA 2003. LNCS, vol. 2647, pp. 81–96. Springer, Heidelberg (2003)
6. Dlekmann, R., Gehring, J., Luling, R., Monien, B., Nubel, M., Wanka, R.: Sorting large data sets on a massively parallel system. In: Proceedings of the 6th Symposium on Parallel and Distributed Processing, pp. 2–9. IEEE, Los Alamitos (1994)
7. Estivill-Castro, V., Wood, D.: A Survey of Adaptive Sorting Algorithms. Computing Surveys 4(24), 441–475 (1992)
8. Gedigaa, G., Duntschb, I.: Approximation quality for sorting rules. Computational Statistics & Data Analysis 40, 499–526 (2002)
9. Helman, D.R., Bader, D.A., JaJa, J.: A Randomized Parallel Sorting Algorithm with an Experimental Study. Parllel and Dirtributed Computing 1(52), l-23 (1998)
10. Jeon, M.S., Kim, D.S.: Parallel Merge Sort with Load Balancing. International Journal of Parallel Programming 1(31), 21–33 (2003)
11. Kruse, R.L., Ryba, A.J.: Data Structures and Program Design in C++, 2nd edn. Pearson Education (1999)
12. Knuth, D.E.: Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1998)
13. Larriba-Pey, J.: An Analysis of Superscalar Sorting Algorithms on an R8000 Processor. In: Intl. Conf. of the Chilean Computing Society, Chile, pp. 125–134 (1997)
14. LaMarca, A., Ladner, R.E.: The influence of caches on the performance of sorting. In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, January 5-7, pp. 370–379 (1997)
15. LaMarca, A., Ladner, R.E.: The Influence of Caches on the Performance of Sorting. In: Proc. Eighth Ann. ACM-SIAM Symp. Discrete Algorithms (1997)
16. Larson, P.: External Sorting: Run Formation Revisited. IEEE Transactionson Knowledge and Data Engineering 15(4), 961–972 (2003)
17. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing 4(14), 361–372
18. Pai, V.S., Varman, P.J.: Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis. In: Proc. Int. Conf. Data Eng., pp. 273–282 (1992)
19. Raghaven, P.: Lecture Notes on Randomized Algorithms, tech. report, IBM Research Division, Yorktown Heights, New York (1990)
20. Rashid, L., Hassanein, W.M., Hammad, M.A.: Analyzing and Enhancing the Parallel Sort Operation on Multithreaded Architectures. J. Supercomputer (2009)
21. Salzberg, B.: Merging Sorted Runs Using Large Main Memory. Acta Informatica 27(3), 195–215 (1989)

22. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. J. Exp. Algorithmics 9, 1–5 (2004)
23. Trimananda, R., Haryanto, C.Y.: A Parallel Implementation of Hybridized Merge-Quicksort Algorithm on MPICH. In: 2010 International Conference on Distributed Framework for Multimedia Applications (DFmA)
24. Weiss, M.A.: Data Structure & Algorithm Analysis in C++, 2nd edn. Addison-Wesley Longman (1999)
25. Zhang, W., Larson, P.A.: Dynamic Memory Adjustment for External Mergesort. In: Proc. Very Large Data Bases Conf., pp. 376–385 (1997)
26. Zhang, W., Larson, P.A.: Buffering and Read-Ahead Strategies for External Mergesort. In: Proc. Very Large Data Bases Conf., pp. 523–533 (1998)
27. Zheng, L., Larson, P.A.: Speeding Up External Mergesort. IEEE Trans. Knowledge and Data Eng. 8(2), 322–332 (1996)