Journal of Computer Science 6 (2): 163-167, 2010
ISSN 1549-3636

Sorting N-Elements Using Natural Order: A New Adaptive Sorting Approach

[1]Shamim Akhter and [2]M. Tanveer Hasan
[1]Aida Lab, National Institute of Informatics, Tokyo, Japan
[2] CEO, DSI Software Company Ltd., Dhaka, Bangladesh

Abstract: Problem statement: Researchers focused their attention on optimally adaptive sorting algorithm and illustrated a need to develop tools for constructing adaptive algorithms for large classes of measures. In adaptive sorting algorithm the run time for n input data smoothly varies from O(n) to O(nlogn), with respect to several measures of disorder. Questions were raised whether any approach or technique would reduce the run time of adaptive sorting algorithm and provide an easier way of implementation for practical applications. **Approach:** The objective of this study is to present a new method on natural sorting algorithm with a run time for n input data O(n) to O(nlogm), where m defines a positive value and surrounded by 50% of n. In our method, a single pass over the inputted data creates some blocks of data or buffers according to their natural sequential order and the order can be in ascending or descending. Afterward, a bottom up approach is applied to merge the naturally sorted subsequences or buffers. Additionally, a parallel merging technique is successfully aggregated in our proposed algorithm. **Results:** Experiments are provided to establish the best, worst and average case runtime behavior of the proposed method. The simulation statistics provide same harmony with the theoretical calculation and proof the method efficiency. **Conclusion:** The results indicated that our method uses less time as well as acceptable memory to sort a data sequence considering the natural order behavior and applicable to the realistic researches. The parallel implementation can make the algorithm for efficient in time domain and will be the future research issue.

Key words: Adaptive sorting, mergesort, natural order, partition, algorithm and complexity

INTRODUCTION

Sorting a huge data set in a very nominal time is always a demand for almost all fields of computer science. As mentioned above, in the sorting technique arena, natural order is taken into deep consideration. And in this study, we are proposing a new sorting approach to reduce the running time to O(nlogm), where m <= n/2.

A general and extremely successful strategy for the design and analysis of algorithm is “divide and conquer” and it is the basis of infinitude of sorting algorithms for the usual comparison-based model of computation. Over all view, divide and conquer is a bottom up approach followed by a top down traverse.

In the recent history, measurement of disorder has been studied as a universal method for the development of adaptive sorting algorithms (Chen and Carlsson, 1991). In the adaptive technique, a bottom up traverse is enough after calculating the disorderness. The design of generic sorting algorithms results in several advantages (Estivill-Castro and Wood, 1992a), for example:

- The algorithm designer can focus the efforts on the combinatorial properties of the measures of disorder of interest rather than in the combinatorial properties of the algorithm
- The designer can regulate the trade-off between the number of measures for adaptivity and the amount of machinery required
- The resulting implementations are practical and do not require complex data structures
- Parallelism is present as the approach is inherited from Mergesort (JaJa, 1992)

In the proposed technique, at first, the disorderness of the data is checked and partitioned in a single pass over the data set. Thereafter, the partitions are merged according to their order. It has been ensured that the approach provides the optimum time while the bottom up merging tree is balanced.

In the study, we used “log” to denote the base 2 logarithms, “n” is the total number of elements in the data set, “m” is the number of partition buffers required. BELLOW has been used as a notation of BELLOW(n); for example the running time for any

Corresponding Author: Shamim Akhter, Aida Lab, National Institute of Informatics, Tokyo, Japan

algorithm BELLOW(n) means that time needed for the particular algorithm is at most n.

Natural order: Any raw data set contains some natural order or sequence among them. Even in the most disordered situation at least two elements have an ordered sequence, may be increasing or decreasing. For an example, let's consider the data set {9, 5, 3, 4, 10, 12, 8 and 2}. Using these data we will get the following Zig-zag diagram, Fig. 1. Our goal is to make the data sorted, means the result set of the above data will be {2, 3, 4, 5, 8, 9, 10, 12}. Figure 2 presents the Zig-zag diagram represents sorted data in natural order.

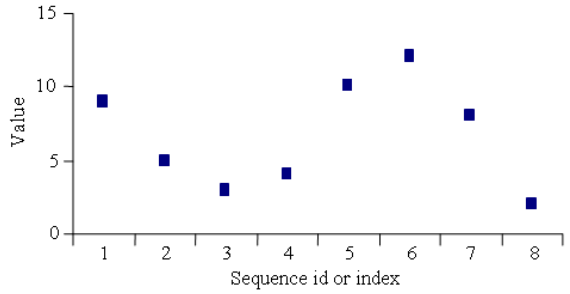

Fig. 1: Zig-zag diagram for natural disorder data set

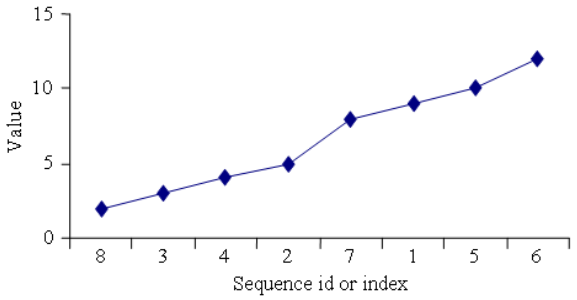

Fig. 2: Zig-zag diagram for sorted data in natural order

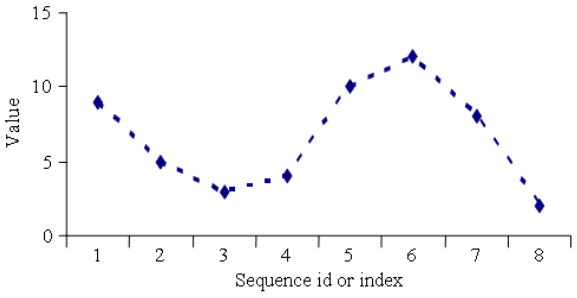

Fig. 3: Revised Zig-zag diagram for natural disorder data set

Figure 3 and 4 will help to understand the difference between classical sorting algorithm and adaptive sorting algorithm. In all classical ways, sequence Ids are shifted to gain the sorted order. However, in adaptive sorting scheme, lines, connecting the points are taken into consideration. And by doing so, all the points on the two lines (currently under process), are in action. In natural order, in the proposed technique, at least two points are in one line and there comes the time complexity of proposed technique:

BELLOW(m)

where, m <= n/2.

According to the Fig. 4, we make lines L1 = <9, 5, 3>, L2 = <4, 10, 12>, L3 = <8, 2> and finally merging these lines we will get approximately a straight line (represents the data are successfully sorted) present in Fig. 2.

Ordering complexity: In order to express the performance of a sorting algorithm in terms of the length and the disorder in the input, we must evaluate the disorder in the input. Intuitively, a measure of disorder is a function that is minimized when the sequence has no disorder and depends only on the relative order of the elements in the sequence (Estivill-Castro and Wood, 1992a).

There are several measures of disorder. We define the most three common measures of disorder (Estivill-Castro and Wood, 1992b). Runs(n) as the minimum number of contiguous up-sequences required to cover n data. A natural generalization of Runs is the minimum number of ascending subsequences required to cover the given sequence and denoted by Shuffled Up-Sequences (SUS). We generalize again and define SMS(n) (for Shuffled Monotone Subsequence) as the minimum number of monotone (ascending or descending) subsequences required to cover the given sequence.

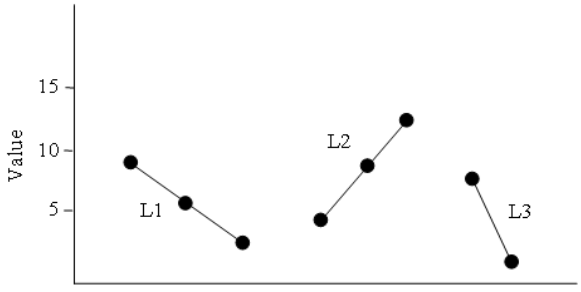

Fig. 4: Zig-zag diagram after buffering

For example W0 = <6, 5, 8, 7, 10, 9, 4, 3, 2, 1> has Runs (W0) = 8, while SUS(W0) = ||{<6, 8, 10>, <5, 7, 9>, <4>, <3>, <2>, <1>}|| = 7 and SMS(W0) = ||{<6, 8, 10>, <5, 7, 9>, <4, 3, 2, 1>}|| = 3 (Estivill-Castro and Wood, 1992a). This technique also provide ||{<6, 5>, <8, 7>, <10, 9, 4, 3, 2, 1>}|| = 3. The number of ascending runs is directly related to the measure Runs, Natural Mergesort takes O(|n| (1 + log[Runs(n) + 1])) time. Quick sort takes O(|n| log[n+1]) running time in average case.

Many researches were conducted to focus on the time complexity minimization of the sorting algorithm and their proposed algorithms successfully partitioned the input data, but they didn't focus on the partitioning with both ascending and descending order. Moreover, the cost needed to partition is also an important point and need to take under consideration. In our view, if we consider both ascending and descending order, which will be needed only at the bottom level in the bottom-up traversing of the merge-tree and this approach will reduce the number of partitions. Thus, the time needed in worst case will be:

$$\text{BELLOW } (n + n\log m)$$

where, m <= n/2.

It can also be mentioned, in an average case of disorder in data set, m<n/2 and for the best case m = 1. Thus, the time complexity of the new approach would be:

$$O(|n|(\log[m]))$$

Inheriting the thoughts of co-thinkers walking in this arena of adaptive sort, we have used merge sort to implement the new adaptive method and that is why, this approach will also provide the chance of improvement using parallel algorithm. Parallelism in merge sort improves the run time complexity. Using merge sort algorithm, sorting a sequence of n elements can be done optimally in O(logn log (log n)) (JaJa, 1992). According to Simple Merge Sort (JaJa, 1992), the running time of this algorithm is O(logn log (log n)) and the total number of operations used is O(n log n) (where PRAM model will be CREW PRAM). And just to mention again, (Fig. 6) our technique reduces the number of nodes in the merge tree, so reduces the time needed by parallel processing.

MATERIALS AND METHODS

In the classical merge sort algorithm, first comes a top down traverse and then follows a bottom up merge. This has been showed in Fig. 5 with a data set {9, 5, 3, 4, 10, 12, 8, 2}.

In the adaptive sorting algorithm, using the proposed ordering scheme, for the same data set, {9, 5, 3, 4, 10, 12, 8, 2}, the partitions will be {9, 5, 3}, {4, 10, 12}, {8, 2}. And the following merging is represented in Fig. 5. In this averagely ordered data set, the proposed algorithm traverse only a tree of height 3, followed by a single pass over the data set.

For n element data set, we first make some buffers (m) according to their sequential order, the order may be in ascending or descending, the running time will need O(n). Each buffer will get information about the starting index and the ending index of the sequential sorted data and also a flag which will provide us the order of the sequential data (flag 0 means ascending, flag 1 means descending order), this flag will be needed to check only in the first level comparisons but the rest levels no need to check.

In Table 1, we consider the Fig. 4, the buffer L1 = <9, 5, 3>, where 9 is the first and 3 is the last element of this particular data set. So, the Starting index is 1 and Ending index is 3. Data set 9 to 3 is in descending order, so the flag is set to 1.

If number of buffers is m, for a data set of n elements and divide-and-conquer is the approach to merge the m buffers then merging m sorted buffers(total n data) needs O(nlogm) time (Horowitz *et al*., 1997). Deriving from this information, the proposed algorithm has a time complexity of O(nlogm) where m<n. In the best data set distribution, all the elements are sorted naturally, in an ascending order or descending.

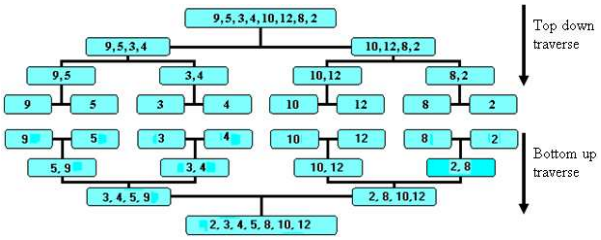

Fig. 5: Classical merge sort algorithm

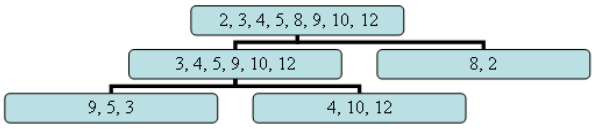

Fig. 6: Merge sort with proposed algorithm

Table 1: The buffer information

Starting index	1
Ending index	3
Flag	1

Table 2: Proposed algorithm run time statistic

Size of inputted data n	Partitioning comparisons	Buffer size (m)	Tree height log (m)	Comparisons in each level	Theoretical total value [n*log m + n]	Algorithm generate value
1000	1000	414	9	928	9693	9064
2000	2000	821	10	1875	21362	20154
3000	3000	1224	10	2818	33772	31905
4000	4000	1649	11	3773	46748	44320
5000	5000	2058	11	4710	60036	56845

For this Naturally Sorted data, there will be only one buffer, i.e., m = 1 and no need to apply merge. This reduces the best case time complexity to O(n).

The steps of the proposed algorithm are presented bellow, assuming n elements of data set is stored in an array A[1..n].

Partition(n):

Beginning from index 1 in A[], continue traversing up to index k where A[1], A[2] … A[k] is sorted in any order (ascending or descending). If A [1…k] is in ascending, i.e. flag = 0, then A [k+1]<A[k] else for descending, i.e., flag = 1, A[k]>A[k+1]. This represents a line in the data sequence (Table 1).

Thereafter, store the information (start_index, ending_index, flag) in an array of buffer. Continue this procedure up to n.

After the Partition(n) function we will get m buffers, where m will be at most n/2 .

Sort(m):

If all data is in a Natural Sorted, means that the number of buffers m is 1 (Reverse, if needed), then Terminate

Otherwise merge m buffers, taking two at a time. By following this level-by-level bottom up procedure will assure the merging tree to be appropriately balanced

foreach level of the bottom up traversal
for(i is 1 to |m|)
merge $(2i-1)^{th}$ and $2i^{th}$ buffers

RESULTS AND DISCUSSION

Based on the proposed algorithm, Table 2 statistics have been presented. In this statistics, we have used a randomly picked data set and values are the average of 1000 times operation.

Table 2 shows statistics sounds the same harmony that is present in the theoretical calculation. Finding out the natural sub-sorted sequence will need only exactly n unit of time. Here, m denotes the number of buffers made after partition. So, if the merging goes in a balanced tree, height of the tree will be [log m]. In most of the cases, calculated value is less than estimated value. It is because, in the theoretical calculation, merging buffers in any level of the tree needs O(n) time. However, sometimes, it is less than that, when sizes of the buffers are not unique. In the worst case, time in each level we have to compare n data. So, total needing comparison will be n*log m + n and m is n/2. So, total comparison will be O(nlogm).

CONCLUSION

The research evaluates the power of a new scheme in the era of sorting algorithm. A good sorting algorithm is always preferable for any kind of application developed in fields Computer Aided Technology and the proposed technique uses less time as well as acceptable memory to sort a sequence considering the natural order, which is already there.

We have already mentioned that our providing criteria will give best effort when the tree (after buffering) will be balanced. So, there will be a great chance for the future developer to implement this technique using parallel approach. Additionally, for any formal high-level language, a library function can be provided using this algorithm, like we have for Quicksort (Horowitz *et al*., 1997).

REFERENCES

Chen, V. and S. Carlsson, 1991. On partitions and presortedness of sequences. Proceeding of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, Jan. 28-30, ACM Press, San Francisco, California, United States, pp: 63-71. http://portal.acm.org/citation.cfm?id=127807&dl=GUIDE&coll=GUIDE&CFID=79579737&CFTOKEN=16004594

Estivill-Castro, V. and D. Wood, 1992a. A generic adaptive sorting algorithm. Comput. J., 35: 505-512.

Estivill-Castro, V. and D. Wood, 1992b. A survey of adaptive sorting algorithms. ACM Comput. Surveys, 24: 441-476. http://portal.acm.org/citation.cfm?id=146381

Horowitz, E., S. Sahni and S. Rajasekaran, 1997. Computer Algorithms. 2nd Edn., Computer Science Press, ISBN: 0716783169, pp: 769.

JaJa, J., 1992. An Introduction to Parallel Algorithms. 1st Edn., Addison-Wesley, ISBN: 13: 9780201548563, pp: 576.