Asian Journal of Research in Computer Science

4(4): 1-7, 2019; Article no.AJRCOS.53311
ISSN: 2581-8260

Performance Analysis of a Faster In−place External Sorting Algorithm

Asaduzzaman Nur Shuvo[1], Apurba Adhikary[1,2*], Md. Bipul Hossain[1] and Sultana Jahan Soheli[1]

[1]Department of Information and Communication Engineering, Noakhali Science and Technology University, Noakhali − 3814, Bangladesh.
[2]Electronics and Communication Engineering Discipline, Khulna University, Khulna − 9208, Bangladesh.

Authors' contributions

This work was carried out in collaboration among all authors. Author ANS designed the study, performed the statistical analysis, wrote the protocol and wrote the first draft of the manuscript. Authors AA and MBH managed the analyses of the study. Author SJS managed the literature searches. All authors read and approved the final manuscript.

Article Information

DOI: 10.9734/AJRCOS/2019/v4i430122
Editor(s):
(1) Dr. M. Ilayaraja, Assistant Professor, Department of Computer Science and Information Technology, Kalasalingam University, Krishnankoil, India.
Reviewers:
(1) Mohammad Shabaz, Chandigarh University, India.
(2) Sapan Naik, Uka Tarsadia University, India.
(3) Faki Ageebee Silas, Bingham University, Nigeria.
Complete Peer review History: http://www.sdiarticle4.com/review-history/53311

Original Research Article

Received 05 October 2019
Accepted 11 December 2019
Published 09 January 2020

ABSTRACT

Data sets in large applications are often too gigantic to fit completely inside the computer's internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottle−neck. While applying sorting on this huge data set, it is essential to do external sorting. This paper is concerned with a new in−place external sorting algorithm. Our proposed algorithm uses the concept of Quick−Sort and Divide−and−Conquer approaches resulting in a faster sorting algorithm avoiding any additional disk space. In addition, we showed that the average time complexity can be reduced compared to the existing external sorting approaches.

**Corresponding author: E-mail: adhikary_apurba@yahoo.com;*

Keywords: External sorting; in−place sorting; sorting; external memory.

1. INTRODUCTION

External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive. Thus, external sorting algorithms are external memory algorithms and thus applicable in the external memory model of computation.

External sorting is required when the number of records to be sorted is larger than the computer can hold in its high−speed internal memory. It is quite different from internal sorting, even though the problem in both cases is to sort a given file into increasing or decreasing order. External sorting algorithms generally fall into two types, distribution sorting, which resembles quicksort, and external merge sort, which resembles merge sort. The latter typically uses a hybrid sort−merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub−files are combined into a single larger file. The most common External sorting algorithm used is still the Merge−Sort as described by Knuth [1], Nasim and Islam [2] and others.

Fang−Cheng Leu, Yin−Te Tsai and Chuan Yi Tang [3] proposed an algorithm in which they gave attention to reduce disk I/O complexity but they did not give attention to reduce the time complexity of sorting. By exploiting the sorting technique of Dufrene and Lin [4], here we propose a new external sorting algorithm. The proposed algorithm is faster than the algorithm proposed by Dufrene and Lin [4], and uses Quick sort and special merging process described by Singh and Naps [5] demanding no other external files except the original one. Since our proposed algorithm is based on the algorithm proposed by Dufrene and Lin [4], the algorithm is reviewed in Methodology section.

2. METHODOLOGY

We used optimal External Memory (EM) algorithms for sorting. The following bound is the most fundamental one that arises in the study of EM algorithms:

Theorem 2.1 ([1]). The average−case and worst−case number of I/Os required for sorting N = nB data items using D disks is,

$$\text{Sort}(N) = \varphi\left(\frac{n}{D}\log_M n\right) \qquad (2.1)$$

where,

N = The size of the external file (MB)
n = Number of blocks
B = Block size (MB) = M/2
M = size of memory (MB)

We discuss some recently developed external sorting algorithms that use disks independently and achieve bound (2.1) as used in Horowitz et al. [6] and Vitter and Shriver [7]. The algorithms are based upon the important distribution and merge paradigms, which are two generic approaches to sorting. They use online load balancing strategies so that the data items accessed in an I/O operation are evenly distributed on the D disks. The distribution sort and merge sort methods using randomized cycling, Randomized Cycling Device (RCD) and Randomized Cycling Memory (RCM) and the simple randomized merge sort (SRM) are the methods of choice for external sorting. For reasonable values of size of RAM, M and D, they outperform disk striping in practice and achieve the I/O lower bound (2.1) with the lowest known constant of proportionality. The steps of the algorithm are shown in Fig. 1.

The proposed algorithm is the generalization of internal Bubble sort. The algorithm works in two phases. In the first phase, this algorithm works as the algorithm described in [1]. That is, Block_1 and Block_S are read into lower half and upper half of memory array respectively and they are sorted using Quick sort. This phase terminates when Block_2 is read into the upper half of memory array and sorted with the remaining records in the lower half of memory array.

After this, the algorithm switches to its second phase. In this phase, Block_S−1 and Block_S are read into the lower and upper half of memory array respectively. Then the algorithm uses the special merging process. The diagrams of sorting by special merging technique have been shown in Fig. 2. We say the merging process used here, is a special one because, the merging is accomplished in two steps. In the first step, merging is applied to sort the records (as both halves of memory array contain sorted records) of the lower and upper half of memory array and the sorted records are written simultaneously in the position of Block_S−1 in the external file until the block is full. In the second step, the remaining records in the lower and upper half of memory array are again merged and the sorted records are written from the beginning of the upper half of memory array simultaneously. Now the upper half of memory array contains the highest ordered records of Block_S and Block_S−1.

After this, Block_S-2 is read into lower half of memory array. In this way, when the last block, Block_2 has been processed, the upper half of memory array contains the highest sorted records of the entire file and they are written in the position of Block_S in the external file.

The next iteration starts with Block_S-2 and Block_S-1 to be read into the lower and upper half of memory array respectively. At the end of this iteration upper half of memory array contains the highest sorted records among the blocks Block_2, Block_3, …., Block_S-1 and they are written in the position of Block_S-1 in the external file. After each pass, as in the case of the Bubble Sort, the size of the external file is decreased by one block. The last blocks to be processed are Block_2 and Block_3, upon the completion of which the whole file is sorted.

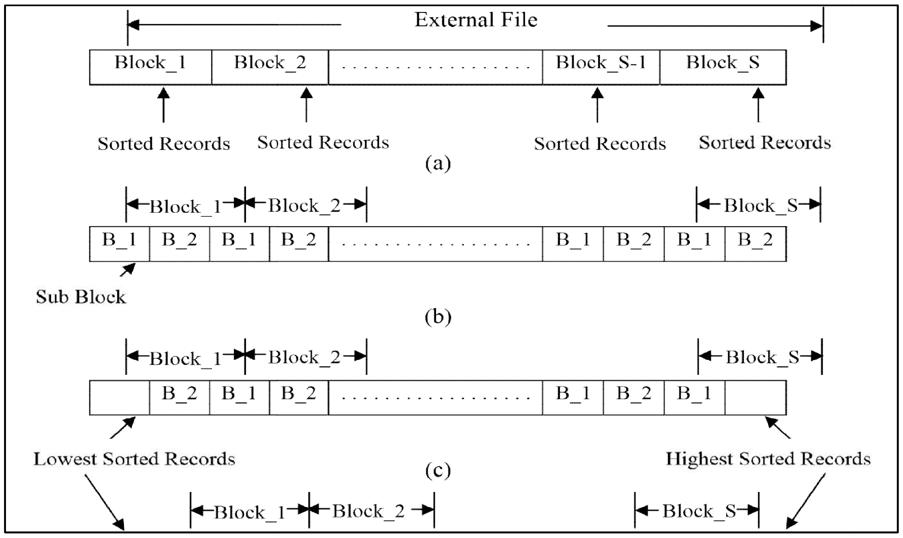

Fig. 1. Steps of algorithm (a) Apply quick sort; (b) Divide sub−blocks; (c) First Iteration

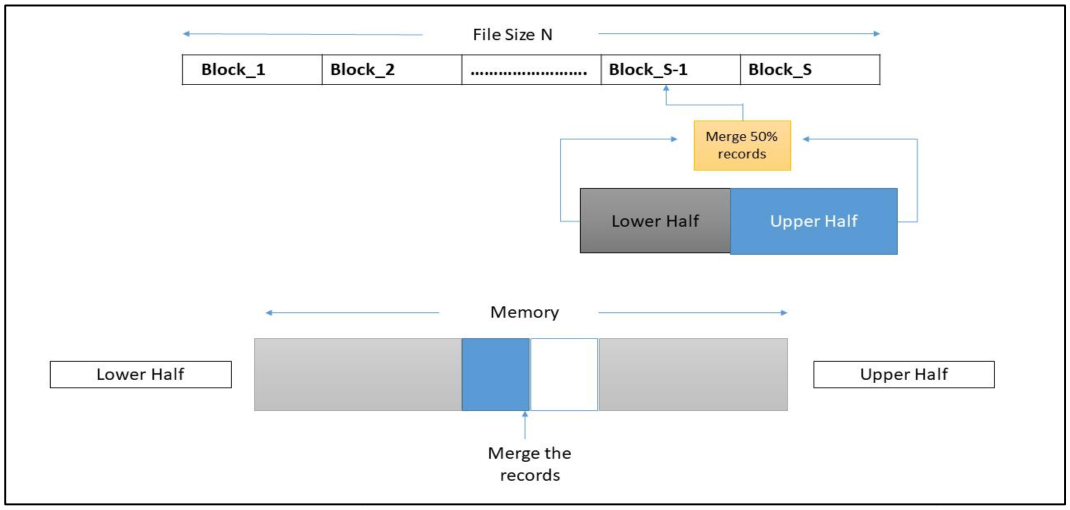

Fig. 2. Sorting by special merging technique

2.1 Algorithm Using Special Merging Technique

1. Declare the blocks in the external file to be half of memory array. Let the blocks are Block_1, Block_2, …, Block_*S*−1, Block_*S*.
2. Read Block_1 into the lower half of memory array. Set *T* = *S*.
3. Read Block_T into the upper half of memory array.
4. Sort the entire memory array using Quick sort.
5. Write upper half of memory array to Block_*T* area of external file.
6. T = T−1.
7. Go to step 3 if T $\neq$ 1.
8. Write lower half of memory array to Block_1 area of external file. Set P = S.
9. Read Block_*P* into the upper half of memory array and set Q = P−1.
10. Read Block_*Q* into the lower half of memory array.
11. Sort (merge) the memory array by using Merge(). Here Merge () writes lowest sorted half of the records of memory array to the Block_*Q* area of the external File. And then the remaining records in the lower and upper half of the memory array are written from the beginning of the upper half of memory array by Merge(), so that the upper half of memory array contains sorted records.
12. Q = Q – 1.
13. Go to step 10 if Q $\neq$ 1.
14. Write the upper half of memory array to the area of Block_*P* in the external file.
    P = P−1
15. Go to step 9 if p $\neq$ 2.

```
Merge()
{
// This procedure is used to merge (sort) the records in memory array.
// RAM [ ] is representing the memory array.
// n is the number of records that fit into memory array. That means block size is n/2.
h: = 1; // first position of the lower half of memory array.
Pt: = start of Block_Q.
j: = n/2 + 1; // first position of the upper half of memory array.
S: = j; copy: = j; loop: = 1;
/ / The Special merging process of the algorithm.

While (loop < S)
{
        if (RAM [h] <= RAM [j]) then
        {
        Block_Q [pt]: = RAM [h]; h: = h + 1;
        }
        else
        {
        Block_Q[pt]: = RAM [j]; j: = j + 1;
        }
                pt: = pt + 1; loop: = loop + 1;
}
while (h <= n/2)
{
      if (RAM [h] <= RAM [j]) then
      {
        RAM [copy]: = RAM [h]; h: = h + 1;
      }
    }
  }
```

3. RESULTS AND DISCUSSION

The time complexity of the internal Quick sort is O (n $log_e n$) in average case, as given by Knuth [1]. Here, n is the number of records to be sorted. So, the time complexity for the first phase of our algorithm is n $log_e n$ (N / B−1). In the second phase, we use the special merging technique by Merge ().

Now, as per special merging technique, if there are n records in memory array, n / 2 records are merged at a time. In addition, the time complexity of merging is: 1/ 2 × (n/ 2 + n/ 2) = n/ 2 (as given by Knuth [1]). Now, merging of records occurs twice with n/2 records when special merging technique is encountered in the algorithm. So, the time complexity in the second phase is:

$$[(N / B - 2) + (N / B - 3) \dots + 1] \times (n/ 2 + n/ 2)$$

$$= [(N / B - 2) + (N / B - 3) + \dots + 1] \times n$$

$$= n\sum_{i=1}^{\frac{N}{B}-2} i$$

So, the total time complexity of the algorithm is:

$$T1 = n log_e n(N/B - 1) + \sum_{i=1}^{\frac{N}{B}-2} i$$

3.1 Comparison of Time Complexity

The algorithm proposed by Dufrene and Lin [4] uses only Quick sort to sort the external file. So, the time complexity of the algorithm (in the average case) is:

$$T2 = [(N/B - 1) + (N/B - 2) + \dots + 1] \times n log_e n$$

$$= n log_e n \sum_{i=1}^{\frac{N}{B}-1} i$$

Now we assume that, T1 = T2.

$$\rightarrow \quad n log_e n(N/B - 1) + n\sum_{i=1}^{\frac{N}{B}-2} i = n log_e n \sum_{i=1}^{\frac{N}{B}-1} i$$

$$\rightarrow \quad n\sum_{i=1}^{\frac{N}{B}-2} i = n log_e n \sum_{i=1}^{\frac{N}{B}-2} i$$

$$\rightarrow \quad \sum_{i=1}^{\frac{N}{B}-2} i = log_e n \sum_{i=1}^{\frac{N}{B}-2} i$$

$$\rightarrow \quad 1 = log_e n$$

But, $1 \neq log_e n$ (For *n* is a positive integer).
For n = 3,4,5… ∞

$$1 < log_e n$$

Therefore, T1<T2 for n>2. So, our algorithm is better than the algorithm proposed by Dufrene and Lin [1].

Here, the reduction of time complexity of the proposed algorithm from the algorithm proposed by Dufrene and Lin [4] is calculated and shown in the Table 1.

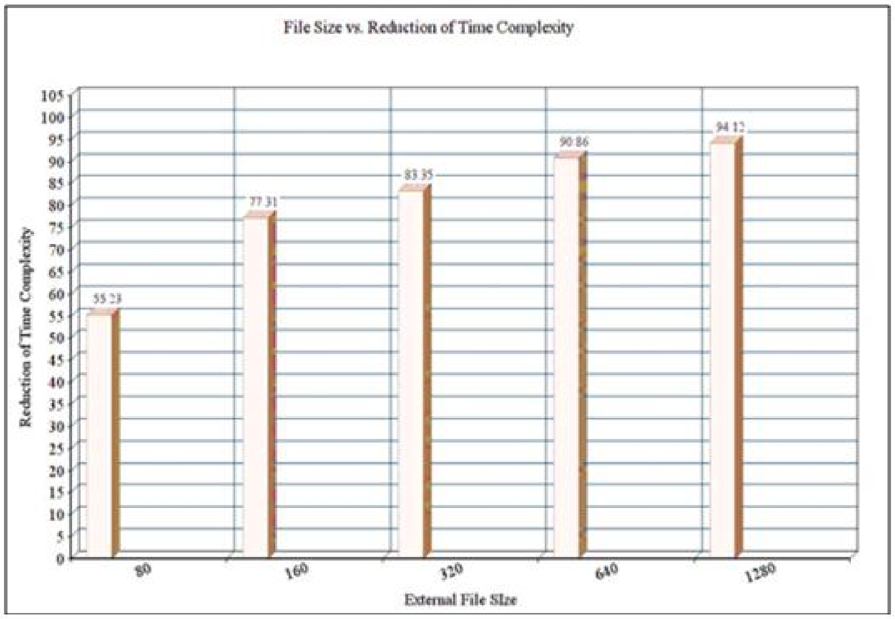

Fig. 3. Reduction of time complexity vs external file size

Table 1. Reduction of time complexity

External file size (MB)	RAM size (MB)	Record size(Byte)	Number of records In RAM (n)	T1 for our proposed algorithm(minutes)	T2 for algorithm by Dufrene and Lin[4] (minutes)	Ratio of time complexity (T1/T2)	Reduction of time complexity (%)
80	32	1024	32768	183	400	0.4575	54.25 %
160	32	1024	32768	2769	10000	0.2769	72.31 %
320	32	1024	32768	373	2000	0.1865	81.35 %
640	32	1024	32768	1411	10000	0.1411	85.89 %

From Table 1, it is definite that with the increase of the size of the eternal file size, the reduction of time complexity increases compared to the algorithm proposed by Dufrene and Lin [4].

$$\text{Here, T1/T2} = \frac{nlog_e n + \frac{N-M}{M} \times n}{nlog_e n + \frac{N-M}{M} \times nlog_e n}$$

Now it is vibrant that, if (N−M)/M is higher, then the proposed algorithm will have a clear benefit over the algorithm proposed by Dufrene and Lin [4]. Because, the proposed algorithm will substitute Quick sort (N−M)/M times by the special merging technique. So, the algorithm will sort faster if the external file is many times larger than the available memory (RAM) of the computer. We have calculated the reduction of time complexity (in percentage) of the proposed algorithm from the algorithm proposed by Dufrene and Lin [4]. Moreover, our proposed algorithm is faster than the algorithm proposed by L. Arge [8], R. Bayer and McCreight [9] and Rafiqul and Raquib [10]. The reduction of time complexity with respect to external file size has been shown in Fig. 3.

4. CONCLUSION

In this paper, we proposed an external sorting algorithm that proves to be very efficient in this respect as each pass completes sorting of a part of the external file. There is no need of extra disk space in our proposed algorithm. The proposed algorithm takes minimum comparisons to sort the records and creates no extra file or huge priority queue. In addition, our proposed algorithm revealed that the average time complexity can be reduced compared to the existing external sorting approaches. Furthermore, we believe that our proposed algorithm can be improved further by reducing the disk I/O complexity.

COMPETING INTERESTS

Authors have declared that no competing interests exist.

REFERENCES

1. Knuth DE. The art of computer programming, sorting and searching, addition Wesley reading. MA. 1973;3.
2. Md. Nasim Adnan, Md. Islam, Md. Nur Islam, Md. Shohorab Hossen. A faster hybrid external sorting algorithm with no additional disk space. International Conference on Computer and Information Technology (ICCIT); 2002.
3. Fang−Cheng Leu, Yin−Te Tsai, Chuan Yi Tang. An efficient external sorting algorithm; 2000.
4. Dufrene WR, Lin FC. An efficient sorting algorithm with no additional space. Comput. J. 1992;35.
5. Singh B, Naps TL. Introduction to data structures. West publishing Co, St Paul, MN; 1985.
6. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran. Fundamentals of computer algorithms. W. H. Freeman and Company; 1998.
7. Vitter JS, Shriver EAM. Algorithm for parallel memory, I: Two−level memories. Algorithmica. 1994;110−147.
8. Arge L, Vitter JS. Optimal dynamic interval management in external memory. In Proc. IEEE Symp. on Foundations of Comp. Sci; 1996.
9. Bayer R, McCreight E. Organization and maintenance of large ordered indizes. Acta Informatica. 2012;1:173.
10. Md. Rafiqul Islam, Raquib Uddin SM. An external sorting algorithm using merging. Malaysian Journal of Computer Science. 2015;18(1):40−49.