Malaysian Journal of Computer Science, Vol. 18 No. 1, June 2005, pp. 40-49

AN EXTERNAL SORTING ALGORITHM USING IN-PLACE MERGING AND WITH NO ADDITIONAL DISK SPACE

Md. Rafiqul Islam and S. M. Raquib Uddin
Computer Science and Engineering Discipline
Khulna University, Khulna-9208, Bangladesh
email: cseku@khulna.bangla.net
raquibuddin26@yahoo.com

ABSTRACT

This paper presents an external sorting algorithm using linear-time in-place merging and without any additional disk space. The algorithm uses quick sort to produce runs in the first phase. In the second phase it uses special verification technique and uses in-place merging technique to reduce the average time complexity and disk I/Os especially the output (write) operations. The I/O and time complexities are analysed and compared with another algorithm [5] which also uses no additional disk space.

Keywords: ***External sorting; Algorithms; In-place merging***

1.0 INTRODUCTION

The problem of how to sort data efficiently has been widely discussed. Most of the time, sorting is accomplished by external sorting, in which the data file is too large to fit into the main memory and must reside in the secondary memory. Disk I/Os are more appropriate measures in the performance of external sorting and other external problems, because the I/O speed is much slower than the CPU speed. The most commonly used external sorting is still the merge sort as described by Knuth [1], Sing and Naps [2] and others. A two-way merge sort requires extra disk space. Fang-Cheng Leu, Yin-Te Tsai and Chuan Yi Tang [3] proposed an algorithm to reduce disk I/Os but it did not reduce the time complexity of sorting. Dufrene and Lin [4], M. R. Islam *et al*. [5] proposed algorithms with no additional disk space. They reduced time complexity but disk I/Os occur frequently in their algorithms. By using the linear-time in-place merging proposed by Huang, B. C., and Langston, M. A. [6], we present an efficient external sorting algorithm with special cases and without any additional disk space.

2.0 EXTERNAL SORTING ALGORITHMS WITH NO ADDITIONAL DISK SPACE

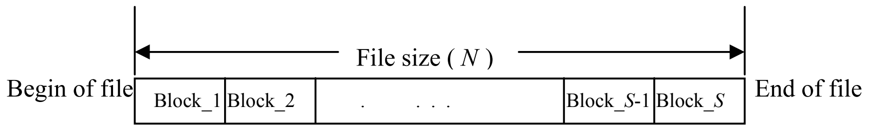

Fig. 1: External file after splitting into blocks

In the algorithm proposed by Dufrene and Lin [4], the external file is divided into equal sized blocks, which are approximately one half of the available memory (RAM) of the computer. Now if M is the size of memory, then the block size, $B = M/2$. Again if N is the size of the external file, then the number of blocks, $S = N/B$ (Fig. 1). At the first iteration Block_1 and Block_S are read into the lower half and upper half of the memory array, respectively and after sorting, the records of the upper half of the memory array are returned to the Block_S area of the external file. Now Block_$S-1$ comes into the upper half and the process continues. The loop terminates when Block_2 has been processed. Each iteration reduces the file size by one block. The next iteration starts again with Block_2. The last two blocks to be processed are Block_$S-1$ and Block_S and upon completion, the entire file is sorted. This algorithm uses quicksort to produce runs.

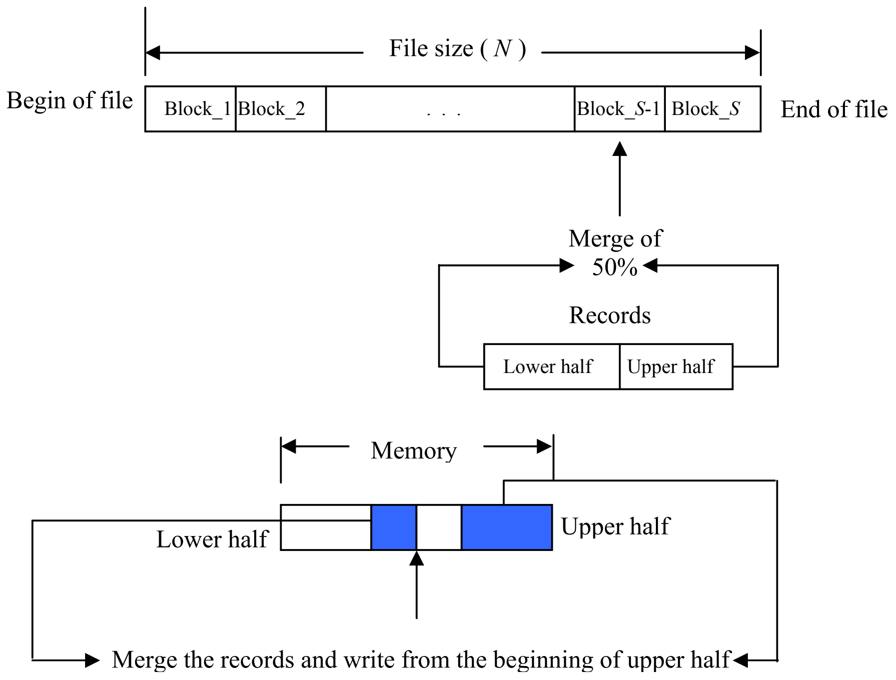

Fig. 2: Sorting by special merging technique

The algorithm proposed by M. R. Islam *et al.* [5] works in two phases (Fig. 2). The first phase is the same as the algorithm proposed by Dufrene and Lin [4]. The algorithm then uses special merging techniques in the second phase. The merging process used here is accomplished in two steps. In the first step, merging is applied to sort the records (as both halves of the 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 the memory array. After this, Block_ $S-2$ is read into the lower half of the memory array and iteratively when Block_ 2 has been processed, the upper half of the 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_ $S-1$ which are read into the lower and upper half of the memory array respectively and the process continues. The last blocks to be processed are Block_ 2 and Block_ 3 and upon the completion, the entire file is sorted. In the next section we present an overview of the main ideas behind this $O(n)$ time merging technique mainly deduced by Huang, B. C., and Langston, M. A. [6], which is used in the proposed algorithm.

2.1 Brief Description of the In-place Merging

Let L denote a list of n records containing two sub lists to be merged, each in non-decreasing order. To generalise the algorithm, let us suppose that n is any number regardless of whether it is even or odd. Fig. 3a illustrates such a list with $n = 18$. Only record keys are listed, denoted by capital letters. Subscripts are added to keep track of duplicate keys as the algorithm progresses. Let us also assume that $\sqrt{n}$ largest-keyed records from the elements of L have permuted at the front of the list (the relative order is immaterial i.e. in any order of the $\sqrt{n}$ largest-keyed records from the elements of L), followed by the remainders of the two sub lists, each of which contains the remaining records in non-decreasing order. To take $\sqrt{n}$ largest-keyed records from the elements of L, we choose the ceiling value of the result of this $\sqrt{n}$ quantity for non perfect square. Fig. 3b shows an example list in this format. Now L is viewed as a series of blocks, each of which is not more than $\sqrt{n}$ size where the leading block is used as the internal buffer and the remaining blocks will have to be sorted so that their tails (rightmost elements) form a non-decreasing key sequence. Records within a block retain their original relative order (Fig. 3c). The next section provides pseudo-code that will be helpful for implementing this merging technique to sort any number of records residing in the main memory. In order to locate two series of elements to be merged, the first series begins with the head (leftmost element) of block 2 and terminates with the tail of block i, $i \geq 2$, where block i is the first block so that $\text{tail}(i) > \text{head}(i+1)$. The second series consists solely of the elements of block $i+1$, where $i = 2$ (Fig. 4a). Using the buffer to merge these two series repeatedly, compare the leftmost unmerged element in the first series to the leftmost unmerged element in the second, swapping the smaller with the leftmost buffer element. In general, the buffer may be broken into two pieces as we merge (Fig. 4b). We halt this process when the tail of block i has been moved to its final position. At this point, the buffer must be in one piece, although not on a block boundary. Block $i+1$ must contain one or more unmerged elements (Fig. 4c). To locate the next two series of elements to be merged, the first begins with the leftmost unmerged element of block $i+1$ and terminates as before for some $j \geq i$. The second consists solely of the elements of block $j+1$, where $j = 4$ (Fig. 5a). We resume the merge until the tail of block j has been moved (Fig. 5b). We continue this process of locating series of elements and merging them until we reach a point where only one such series exists, leaving the buffer in the last block (Fig. 5c). A sort of the buffer completes the merge of L (Fig. 5d).

$$\underbrace{C_1C_2D_1E_1G_1J_1M_1Q_1Q_2}_{\text{Sublist 1}}\underbrace{E_2H_1I_1K_1K_2L_1Q_3R_1V_1}_{\text{Sublist 2}}$$

a) Example list L, with $n = 18$

$$\underbrace{V_1R_1Q_3Q_2Q_1}_{\text{Buffer}}\underbrace{\underbrace{C_1C_2D_1E_1G_1}\underbrace{J_1M_1}}_{\text{remainder of sublist 1}}\underbrace{\underbrace{E_2H_1I_1K_1K_2}\underbrace{L_1}}_{\text{remainder of sublist 2}}$$

b) Internal buffer is extracted

$$\underbrace{V_1R_1Q_3Q_2Q_1}_{\text{buffer}}\underbrace{C_1C_2D_1E_1G_1}_{\text{block 2}}\underbrace{E_2H_1I_1K_1K_2}_{\text{block 3}}\ \underbrace{L_1}_{\text{block4}}\ \underbrace{J_1M_1}_{\text{block5}}$$

c) Blocks are sorted

Fig. 3: Initial rearrangement of blocks

$$\underbrace{V_1R_1Q_3Q_2Q_1}_{\text{buffer}}\underbrace{C_1C_2D_1E_1G_1}_{\text{Series 1}}\underbrace{E_2H_1I_1K_1K_2}_{\text{Series 2}}\ \underbrace{L_1}_{\text{block4}}\ \underbrace{J_1M_1}_{\text{block5}}$$

a) Locating the first two series of elements, $i = 2$

$$\underbrace{C_1C_2D_1E_1E_2}_{\text{merged}}\underbrace{V_1R_1Q_3Q_2G_1}\underbrace{Q_1H_1I_1K_1K_2}\ \underbrace{L_1}_{\text{block4}}\ \underbrace{J_1M_1}_{\text{block5}}$$

b) Merging is partially completed

$$C_1C_2D_1E_1E_2G_1\underbrace{R_1Q_3Q_2V_1Q_1}_{\text{buffer}}H_1I_1K_1K_2\ \underbrace{L_1}_{\text{block4}}\ \underbrace{J_1M_1}_{\text{block5}}$$

c) First two series are merged

Fig. 4: Merging the first two series of elements

$$C_1C_2D_1E_1E_2G_1\underbrace{R_1Q_3Q_2V_1Q_1}_{\text{buffer}}\underbrace{H_1I_1K_1K_2L_1}_{\text{series1}}\underbrace{J_1M_1}_{\text{series2}}$$

a) Locating the next two series of elements, $j = 4$

$$C_1C_2D_1E_1E_2G_1H_1I_1J_1K_1K_2L_1\underbrace{Q_3V_1Q_1R_1Q_2}_{\text{buffer}}M_1$$

b) Series is merged

$$C_1C_2D_1E_1E_2G_1H_1I_1J_1K_1K_2L_1M_1\underbrace{Q_3V_1Q_1R_1Q_2}_{\text{buffer}}$$

c) Leaving buffer in the last block

$$C_1C_2D_1E_1E_2G_1H_1I_1J_1K_1K_2L_1M_1Q_3Q_1Q_2R_1V_1$$

d) Buffer is sorted, completing merge

Fig. 5: Completing the merge of L

Algorithm: In-place merging

```
/*mem_arr[] is a global array containing two sorted subsets in its lower and upper halves that is the memory array.
Declare $q$ as the last index of lower half, $r$ as the last index of upper half and $a$ as the ceiling value of square root
of the total number of records in the memory array*/

1. Set $i$=0  //Extracting the buffer elements
2. While ($i$<$a$)
   {
        If mem_arr[$q$] >= mem_arr[$r$] then do
        {
              move each element from array index $i$ by one index position until element of index $q$ is rebuilt
              and store mem_arr[$i$] the previous value of mem_arr[$q$],
        }
        else
        {
              move each element from array index $i$ by one index position until element of index $r$ is rebuilt,
              store mem_arr[$i$] the previous value of mem_arr[$q$] and increment $q$ by one;
        }
        Increment $i$ by one
   } //Closing brace of loop
3. Set $p$=1, $start$=2*$a$-1 and $start1$=$q$+$a$  //Rearranging the blocks
4. If ($start$>$q$) then set $start$=$q$
5. If ($start1$>$r$) then set $start1$=$r$ and $p$=2
6. While($p$)
   {
        If (mem_arr[$start$] > mem_arr[$start1$]) then interchange these two blocks and upgrade $q$, $start$
        and $start1$as necessary;
        Increment $start1$ by $a$ and if ($start1$>$r$ && $p$==2) then set $p$=0, else if ($start1$>$r$) then set $start1$=$r$ and $p$=2;
   } //Closing brace of while loop
7. Set $p$=1, $start$+=$a$ and $start1$=$q$+$a$
8. If ($start1$ > $r$) then set $start1$=$r$
9. If ($start$>$r$ || $start$ >= $start1$ || $q$==$r$) then jump to step 10, else if ($start$>$q$) then set $start$=$q$ and begin from step 6
10. Set $left$=0, $i$=$a$ and $j$=$a$  //Merging
11. If (mem_arr[$i$+1] < mem_arr[$i$]) then set set $k$=$i$+1 and do step 12 and 13
12. Loop
    {
        If(mem_arr[$j$] <= mem_arr[$k$])
        {
              Swap mem_arr[$left$] with mem_arr[$j$] and increment $left$ by one;
              If( $j$< $left$) then set $left$=$j$;
              Increment $j$ by one;
        }
        else
        {
              Swap mem_arr[$left$] with mem_arr[$k$] and increment $left$ by one;
              If($k$ < $left$) then set $left$=$k$;
              Increment $k$ by one and if ($k$ > $r$) then set $k$=$r$;
        }
        If $j$ is greater than $i$ then exit from the loop;
    } //Closing brace of loop
13. Set $i$=$k$-1and $j$=$k$
14. Increment $i$ by one and repeat from step 11 until $i$ is not equal to $r$
15. Set $k$=$left$+$a$  //Moving buffer elements as the rightmost block's elements
16. If $k$ is less than or equal to $r$ then move each element from array index $left$ by one index position until element of
    index $k$ is rebuilt and store mem_arr[$left$] the previous value of mem_arr[$k$]
17. Increment $k$ by one
```

18. Go to step 16 if k is less than or equal to r
19. Quicksort the buffer elements //Sorting the buffer elements
// Buffer elements are sorted through which in-place merging technique is completed

2.2 Algorithm Using In-place Merging

The algorithm works in two phases. In the first phase, the algorithm works as the algorithm proposed by Dufrene and Lin [4] that is, $\text{Block\_}1$ and $\text{Block\_}S$ are read into lower half and upper half of the memory array, respectively, and they are sorted using Quick sort. This phase terminates when $\text{Block\_}2$ is read into the upper half of the memory array and sorted with the remaining records in the lower half of the memory array. Thus we get sorted runs. After this phase, the lower half of the memory array contains the lowest sorted records of the entire file. Then, the algorithm switches to its second phase, whereby the sorting process continues considering the following two cases:

Case 1: Here the required blocks are read and if the last record of the lower half of the memory array is smaller than the first record of the upper half of the memory array, then it is not required to sort the records of the memory array and then the next block will be read for further approach.

Case 2: This is the general case. The in-place merging technique is used here.

In the second phase, $\text{Block\_}S-1$ and $\text{Block\_}S$ are read into the lower and upper halves of the memory array respectively. For Case 1 the blocks are not required to write back in the external file. In Case 2, after applying the in-place merging, the upper half of the memory array contains the highest ordered records of $\text{Block\_}S$ and $\text{Block\_}S-1$ and the lower half is sent back to its corresponding position in the external file (Fig. 6). After this, $\text{Block\_}S-2$ is read into the lower half of the memory array and checked for the conditions specified in Case 1 or Case 2. In this way, when $\text{Block\_}2$ has been processed, the upper half of the memory array contains the highest sorted records of the entire file and they are written in the position of $\text{Block\_}S$ in the external file for Case 2. The next iteration starts with $\text{Block\_}S-2$ and $\text{Block\_}S-1$ to be read into the lower and upper halves of the memory array respectively. At the end of this iteration, the upper half of the memory array contains the highest sorted records among the blocks i.e. $\text{Block\_}2$, $\text{Block\_}3$, . . . , $\text{Block\_}S-1$ and they are written in the position of $\text{Block\_}S-1$ in the external file for Case 2. After each pass, the size of the external file is decreased by one block. The last two blocks to be processed are $\text{Block\_}2$ and $\text{Block\_}3$, which upon completion, the entire file is sorted.

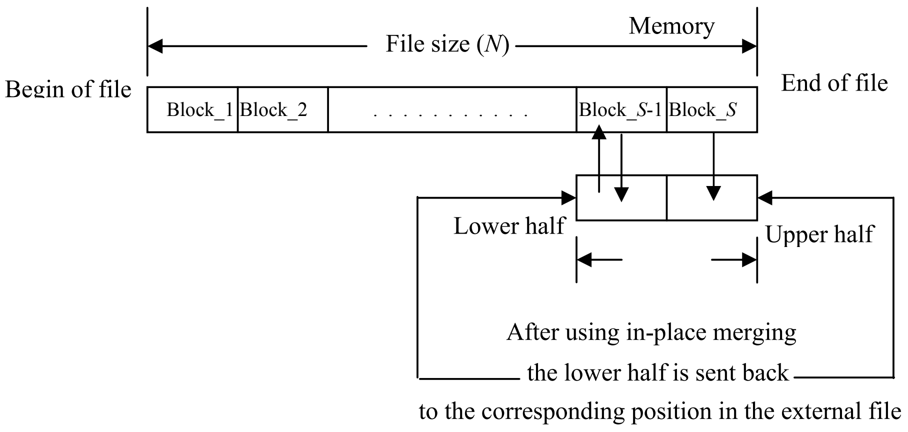

Fig. 6: Sorting by in-place merging

Algorithm: An external sorting algorithm using in-place merging and with no additional disk space

```
1.  Declare the blocks in external file to be half of memory array. Let the blocks be Block_1, Block_2, …, Block_$S-1$, Block_$S$
2.  If there is only one block in the external file then quicksort the entire memory array
3.  Read Block_1 into the lower half of memory array. Set $T = S$ //Begins first phase
4.  Read Block_$T$ into upper half of memory array
5.  Sort the entire memory array using quicksort
6.  Write upper half of memory array to Block_$T$ area of external file
7.  Decrement Block_$T$ by one block
8.  Repeat from step 4 if Block_$T$ is not equal to Block_1
9.  Write lower half of memory array to Block_1 area of external file
10. Set $P = S$ //Begins second phase
11. Read Block_$P$ into the upper half of memory array and set $Q = P - 1$
12. Read Block_$Q$ into the lower half of memory array
13. If last element of the lower half is greater than first element of the upper half then sort (merge) the memory array using in-place merging and write lower half of memory array to Block_$Q$ area of external file
14. Decrement Block_$Q$ by one block
15. Repeat from step 12 if Block_$Q \neq$ Block_1
16. Write upper half of memory array to Block_$P$ area of the external file. Decrement Block_$P$ by one block
17. Repeat from step 11 if Block_$P \neq$ Block_2
//End of sorting procedure
```

3.0 DISK I/OS AND TIME COMPLEXITIES

3.1 Disk I/Os

If M is the size of the available memory i.e., the number of records that can fit into the main memory, then the block size, $B = M/2$. Again if N is the size of the external file then the number of blocks is $S = N/B$. In the first phase, N/B blocks will have to be processed and it takes N/B read (input) operations. After this phase, Block_1 of the external file contains the lowest sorted records of the entire file. $N/B-1$ blocks will have to be processed in the second phase. After each pass of the second phase the file size is reduced by one block. The last blocks to be processed are Block_2 and Block_3 and upon completion, the entire file is sorted. Thus in the second phase, the number of input operations is $(N/B-1)+(N/B-2)+(N/B-3)+\ldots+2$.

Total number of disk input is:

$$N/B+(N/B-1)+(N/B-2)+(N/B-3)+\ldots+2 \tag{1}$$
$$=[N/B+(N/B-1)+(N/B-2)+(N/B-3)+\ldots+2+1]-1$$
$$=\tfrac{1}{2}(N/B)(N/B+1)-1$$
$$=\frac{N^2}{2B^2}+\frac{N}{2B}-1$$

The number of output (write) operations is as follows. In the first phase, it takes N/B output (write) operations. After this phase, each block of the external file is individually sorted. In the second phase, $N/B-1$ blocks will have to be processed. Each pass of this phase reduces the file size by one block. Now we will consider the two cases specified in the algorithm. For Case 1, if the last record of the lower half of the memory array is smaller than the first record of the upper half of the memory array, then the blocks are not required to write back to the external file. Otherwise the control switches to Case 2. Now let the probability to fall in Case 1 be P and the probability to fall in Case 2 be Q. Here $P+Q=1$ and $0 \le P \le 1$. Therefore $P=1-Q$ where $0 \le Q \le 1$. So the number of output operations in the second phase is:
$P*0+Q[(N/B-1)+(N/B-2)+(N/B-3)+\ldots+2] = Q[(N/B-1)+(N/B-2)+(N/B-3)\ldots+2]$. Here, in the worst case (if always case 2 is encountered) $Q=1$. Then the number of output operations in the second phase is

$.(N/B-1)+(N/B-2)+(N/B-3)...+2$. Now, for the worst case the total number of output operations of the algorithm is:

$$N/B+\left[(N/B-1)+(N/B-2)+(N/B-3)...+2\right]$$
$$=\left[N/B+(N/B-1)+(N/B-2)+(N/B-3)+\cdots+2+1\right]-1$$
$$=\tfrac{1}{2}N/B(N/B+1)-1$$
$$=\tfrac{1}{2}\left(\frac{N^2}{B^2}+\frac{N}{B}-2\right).$$

Here the average case $Q=1/2$. Then the number of output operations in the second phase is: $\tfrac{1}{2}\left[(N/B-1)+(N/B-2)+(N/B-3)...+2\right]$. Now, the average case for the total number of output operations of the algorithm is:

$$N/B+\tfrac{1}{2}\left[(N/B-1)+(N/B-2)+(N/B-3)...+2\right]$$
$$=\tfrac{1}{2}\left[\{N/B+(N/B-1)+(N/B-2)+\cdots+2+1\}+N/B-1\right]$$
$$=\tfrac{N}{4B}\left(\tfrac{N}{B}+1\right)+\tfrac{N}{2B}-\tfrac{1}{2}$$
$$=\frac{1}{4}\frac{N^2}{B^2}+\frac{3}{4}\frac{N}{B}-\frac{1}{2}$$

3.2 Time Complexity

The time complexity of the internal quick sort is $\mathrm{O}(n\log_e n)$ 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 $(N/B-1)(n\log_e n)$. The second phase consists of two mutually exclusive cases and let the probability to fall in Case 1 be P and the probability to fall in Case 2 be Q. Here, $P+Q=1$ and $0\le P\le 1$. Therefore, $P=1-Q$ where $0\le Q\le 1$. Now for Case 1, it is not required to sort the records of the memory array. For Case 2, if there are n records in the memory array then the time complexity of merging of the records in both halves is $\mathrm{O}(n)$ as given by Huang and Langston [6]. So, the time complexity in the second phase is:
$Q[(N/B-2)+(N/B-3)+...+1]n+P[(N/B-2)+(N/B-3)+...+1]*0$. Here, in the worst case (if always Case 2 is encountered) $Q=1$. Then, the time complexity in the second phase is: $[(N/B-2)+(N/B-3)+...+1]n$. Now the total time complexity of the algorithm in the worst case is:

$(N/B-1)(n\log_e n)+[(N/B-2)+(N/B-3)+....+1]n=n\log_e n(N/B-1)+n\sum_{i=1}^{N/B-2}i.$ Here in average case $Q=1/2$. Then, the time complexity in the second phase is: $\tfrac{1}{2}[(N/B-2)+(N/B-3)+...+1]n$. Now the total time complexity of the algorithm in an average case is:

$$(N/B-1)(n\log_e n)+1/2[(N/B-2)+(N/B-3)+....+1]n$$
$$=n\log_e n(N/B-1)+n/2\sum_{i=1}^{N/B-2}i\ .$$

4.0 COMPARISON OF DISK I/OS AND TIME COMPLEXITIES

4.1 Comparison of Disk I/Os

The algorithm presented by M. R. Islam *et al.* [5] also works in two phases. In the first phase, this algorithm takes N/B I/Os [N/B reads and N/B writes]. In the second phase, $N/B-1$ blocks will have to be processed. In this phase, at each pass whenever Block_2 is encountered in the lower half of the memory array; then for B records of the upper half of the memory array (records after merged and copied from the beginning of the upper half of the memory array) there will be 1 write for *B* records of this block of the upper half. After each pass, the file size is reduced by 1 block. In the first pass of the second phase $(N-B)$ records will have to be processed and in the second pass of the second phase $(N-2B)$ records will have to be processed and so on. Thus in the first pass of the second phase, there will be $N/B-1$ reads and $(N-B)-B+1$ writes and in the second pass, there will be $N/B-2$ reads and $(N-2B)-B+1$ writes and the process continues [Because at each pass of the second phase whenever Block_2 is encountered in the lower half then instead of *B* writes for *B* records of the upper half there will be 1 write]. The total number of disk input of M. R. Islam *et al*.'s algorithm is: $N/B+(N/B-1)+(N/B-2)+(N/B-3)+...+2$; which is similar to equation (1).

So the input (read) operations of M. R. Islam *et al*.'s algorithm is the same when compared to the proposed algorithm.

Table 1: Comparison of output (write) operations for both cases (worst and average) of the proposed algorithm with M. R. Islam *et al*.'s algorithm

External File Size, N (MB)	**Block Size, B (MB)**	**Ratio of Output Operations (in worst case)**	**Ratio of Output Operations (in average case)**	**Reduction of Output Operations (in worst case) (%)**	**Reduction of Output Operations (in average case) (%)**
200	32	0.0589	0.0379	94.11	96.21
320	32	0.0462	0.0273	95.38	97.27
512	32	0.0398	0.0223	96.02	97.77
1000	32	0.0354	0.0188	96.46	98.12

The total number of disk output of M. R. Islam *et al*.'s algorithm is:

$$N/B+[(N-2B+1)+(N-3B+1)+...+(2B-B+1)]$$
$$=N/B+[(N-2B)+(N-3B)+...+B]+(N/B-2)*1$$
$$=N/B+[(N-2B)+(N-3B)+...+B]+N/B-2$$
$$=B[(N/B-2)+(N/B-3)+...+1]+2N/B-2$$
$$=\tfrac{1}{2}B(N/B-2)(N/B-1)+2N/B-2$$
$$=\frac{N^2}{2B}+\frac{2N}{B}-\frac{3N}{2}+B-2\,.$$

Now, in the worst case, the number of output operations of the algorithm with in-place merging is: $\tfrac{1}{2}\left(N^2/B^2+N/B-2\right)$.

We assume, $\tfrac{1}{2}\left(N^2/B^2+N/B-2\right)=N^2/2B+2N/B-3N/2+B-2$

$$\Rightarrow N/2B+3N/2+\left(N^2/2B^2\right)=2N/B+N^2/2B+(B-1)\,.$$

Now from the above equation we assume that i) $N/2B=2N/B\Rightarrow 1/2=2$ where obviously $1/2<2$. Thus, $N/2B<2N/B$; ii) $3N/2=N^2/2B\Rightarrow 3=N/B\Rightarrow 3\le N/B$ where $\lceil N/B\rceil>2$. Thus, $3N/2\le N^2/2B$ for $\lceil N/B\rceil>2$; iii) the remaining quantity cannot change the less than relationship of the whole expression. Thus, $\tfrac{1}{2}\left(N^2/B^2+N/B-2\right)<N^2/2B+2N/B-3N/2+B-2$ for $\lceil N/B\rceil>2$. So, in the worst case; the number of output operations of the proposed algorithm is less than M. R. Islam *et al*.'s algorithm and in average case; the number of output operations of the proposed algorithm is $\frac{1}{4}\left(N^2/B^2\right)+\frac{3}{4}(N/B)-\frac{1}{2}$ which is also less than that of M. R. Islam *et al.*'s algorithm. For various external file sizes, the reduction of output (write) operations of the algorithm using in-place merging from the algorithm presented by M. R. Islam *et al.* is calculated for both cases (worst and average) and given in Table 1. Here ratio of output operations in the worst case $=\dfrac{\frac{1}{2}\left(N^2/B^2+N/B-2\right)}{\left(N^2/2B\right)+2N/B-3N/2+B-2}$ and in average case $=\dfrac{\frac{1}{4}\left(N^2/B^2\right)+\frac{3}{4}(N/B)-\frac{1}{2}}{\left(N^2/2B\right)+2N/B-3N/2+B-2}$.

4.2 Comparison of Time Complexity

The time complexity of the algorithm presented by M. R. Islam *et al.* is: $n\log_e n(N/B-1)+n\sum_{i=1}^{N/B-2} i$ and the time complexity of the algorithm with in-place merging in the worst case is: $n\log_e n(N/B-1)+n\sum_{i=1}^{N/B-2} i$. However, in an average case, the time complexity of the algorithm with in-place merging is: $n\log_e n(N/B-1)+n/2\sum_{i=1}^{N/B-2} i$.

Table 2: Comparison of time complexity (in average case) of the proposed algorithm with M. R. Islam *et al.*'s algorithm

External File Size, N (MB)	**RAM Size, M (MB)**	**Record Size (Byte)**	**Number of Records in RAM (n)**	**Ratio of Time Complexity (T_2/T_1) (in average case)**	**Reduction of Time Complexity (in average case) (%)**
100	64	4	16777216	0.9836	1.64
200	64	4	16777216	0.9434	5.66
300	64	4	16777216	0.9093	9.07
500	64	4	16777216	0.8547	14.53

Now we assume that, $n\log_e n(N/B-1)+n/2\sum_{i=1}^{N/B-2} i = n\log_e n(N/B-1)+n\sum_{i=1}^{N/B-2} i \Rightarrow n/2\sum_{i=1}^{N/B-2} i = n\sum_{i=1}^{N/B-2} i$

$\Rightarrow n/2 = n$. However, $n/2 \neq n$ (for n is a positive integer) and $n/2 < n.$ So in an average case, the algorithm with in-place merging is better than the algorithm proposed by M. R. Islam *et al.* and in the worst case, the time complexities are the same for both of the algorithms. Table 2 shows the reduction of time complexity of the proposed algorithm as compared to the algorithm presented by M. R. Islam *et al.*

Here $T_1 =$ time complexity of M. R. Islam *et al.*'s algorithm

$$
\begin{aligned}
&= n\log_e n(N/B-1)+n\sum_{i=1}^{N/B-2} i\\
&= n\log_e n(N/B-1)+[(N/B-2)+(N/B-3)+\ldots+1]n\\
&= n\log_e n(N/B-1)+\frac{n}{2}(N/B-2)(N/B-1)\\
&= (N/B-1)\left[n\log_e n+\frac{n}{2}(N/B-2)\right]\\
&= (N/B-1)[n\log_e n+n(n-2B)/2B]\\
&= (N/B-1)[n\log_e n+((N-M)/M)n]
\end{aligned}
$$

Here T_2 = time complexity of the proposed algorithm in average case

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

$$= (N/B-1)\left[n\log_e n + ((N-M)/M)\tfrac{n}{2}\right]$$

Thus, ratio of time complexity (in average case), $\dfrac{T_2}{T_1} = \dfrac{n\log_e n + \frac{n}{2}((N-M)/M)}{n\log_e n + n((N-M)/M)}$

5.0 CONCLUSION

We have presented an external sorting algorithm with no additional disk space using some special conditions and applying an in-place merging technique. Only the Quick sort and in-place merging are used in this algorithm. The algorithm uses probability concepts to reduce disk I/Os, especially the output (write) operations as well as the time complexity in average case and creates no extra file or any priority queue.

REFERENCES

[1] D. E. Knuth, "*Sorting and Searching: The Art of Computer Programming*", Addison-Wesley, Reading, MA, 1973, Vol. 3.

[2] B. Singh and T. L. Naps, "*Introduction to Data Structure*". West Publishing Co, St. Paul, MN, 1985.

[3] Fang-Cheng Leu, Yin-Te Tsai, Chuan Yi Tang, "An Efficient External Sorting Algorithm", revised in May 2000.

[4] W. R. Dufrene, F. C. Lin, "An Efficient Sorting Algorithm with No Additional Space". *Comput. J.* 35 (3) (1992).

[5] R. Islam, N. Adnan, N. Islam, S. Hossen, "A New External Sorting Algorithm with No Additional Disk Space". *Information Processing Letters*, 86 (2003) 229-233.

[6] B. C. Huang, and M. A. Langston, "Practical In-Place Merging". *Communications of the ACM*, March 1988, Vol. 31, No. 3.

BIOGRAPHY

Md. Rafiqul Islam obtained his Master of Science (M. S.) in Engineering (Computers) from the Azerbaijan Polytechnic Institute in 1987 and Ph. D. in Computer Science from Universiti Teknologi Malaysia (UTM) in 1999. His research areas include Design and Analysis of Algorithms, Information Security. He has a number of papers related to these areas published in national and international journals as well as conference proceedings. His research interest involves external sorting and bioinformatics.

S. M. Raquib Uddin has completed his final examination for graduation in Computer Science and Engineering from Khulna University. His research areas include algorithms with respect to external sorting.