ELSEVIER

Information Processing Letters 75 (2000) 159–163

Information Processing Letters

www.elsevier.com/locate/ipl

An efficient external sorting algorithm

Fang-Cheng Leu [a], Yin-Te Tsai [b], Chuan Yi Tang [a,*]

[a] *Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan*

[b] *Department of Computer Science and Information Management, Providence University, Shalu, Taiwan*

Received 15 January 1999; received in revised form 5 May 2000

Communicated by F.Y.L. Chin

Abstract

This paper presents an optimal external sorting algorithm for two-level memory model. Our method is different from the traditional external merge sort and it uses the sampling information to reduce the disk I/Os in the external phase. The algorithm is efficient, simple and it makes a good use of memory available in the recent computer environment. Under the certain memory constraint, this algorithm runs with optimal number of disk I/Os and each record is exactly read twice and written twice.

Keywords: External sorting; Sorting; Algorithms

1. Introduction

The problem of how to sort data efficiently has been widely discussed. Nowadays, to sort extremely large data is becoming more and more important for large corporations, banks, and government institutions, which rely on computers more and more deeply in all aspects. In [8], the authors confirmed that sorting has continued to be counted for roughly one-fourth of all computer cycles. Most of the time, sorting is accomplished by external sorting, in which the data file is too large to fit into main memory and must be resided in the secondary memory. The external sorting is also equivalent in I/O complexity to permuting, transposing a matrix and several combinatorial graph problems. The external sorting algorithm first generates some sorted subfiles called "runs" and then tries to merge them into a sorted file stored in the secondary memory. The number of I/Os is a more appropriate measure in the performance of the external sorting and the other external problems, because the I/O speed is much slower than the CPU speed.

Vitter and Shriver [15] considered a D-disk two-level memory model in which the secondary memory is partitioned into D physically distinct and independent disk drives or read/write heads that can simultaneously transmit a block of data. To be more precise, D is the number of blocks that can be transmitted in parallel at the speed at which data comes off or goes onto the magnetic disk media. The problem of external sorting in the D-disk model has been extensively studied. Barve et al. [2] offered a simple randomized mergesort which is a practical method. Each run is stripped across the disks, but with a random starting point. During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" to free up space. The expected performance of simple

* Corresponding author.

E-mail addresses: dr814322@cs.nthu.edu.tw (F.-C. Leu), yttsai@pu.edu.tw (Y.-T. Tsai), cytang@cs.nthu.edu.tw (C.Y. Tang).

0020-0190/00/$ – see front matter
PII: S0020-0190(00)00098-3

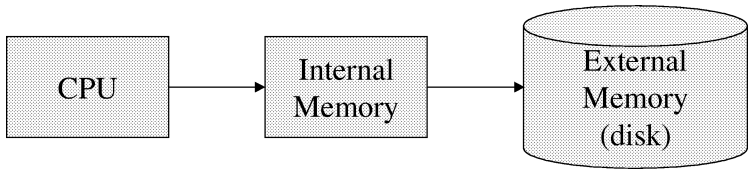

Fig. 1. Two-level memory model with $D = 1$.

randomized mergesort is not optimal for some parameter value, but it outperforms the use of disk stripping for reasonable values of the parameters. Nodine and Vitter [11] presented an optimal deterministic sorting on parallel disks and the algorithm reads each record three or four times (once for producing runs, once for updating the data structure and up to twice for merging runs) and writes each record twice. Fig. 1 depicts the memory hierarchy for one disk model, namely, $D = 1$.

Researchers have concentrated on the external sorting for one disk model. The external mergesort has been the most popular choice, as described by Knuth [6], Singh [12], Kwan [7], and the others. Let N, M and B denote the number of records to be sorted, the number of records that can fit into main memory, and the number of records that can be transferred into a single block, respectively, where $1 \leqslant B \leqslant M \ll N$. The total number of runs is N/M and the total number of blocks is N/B. Each block transfer is then allowed to access any contiguous group of B records on the disk. There are other external sorting algorithms such that the external bubble sort was presented in [4] with $O(N^2 \log N)$ disk I/Os. The external quicksort was proposed in [9,13]. Another external sorting algorithm bases directly on quicksort, designed by Monard [9], and presented in [5], which leads to $N/B(\log_2 N/M) - 0.924$ fetches. The sorting itself is performed in an ordinary quicksort manner. Furthermore, the external sorting algorithm based on the shuffle sort presented in [10] observes that the number of read operations needed during the execution is $N/B(1 + 2\ln((N + 1)/4B))$. Other types of sorting algorithms are also advocated for the external sorting problem, for example, the distributive partitioning, bucketsort, and binsort (see [3,6,8,14,16]). They have corresponding phases as mergesort does, but they are performed in the opposite order. The disk I/O complexity of k-way merge sort is $O(N \log_k N/M)$ as shown in [7]. The k-way merge-sort consists of two phases. In phase 1, it takes N/B disk I/Os (N/B reads and N/B writes) and produces N/M runs for phase 2. In phase 2, the k-way merge-sort merges k runs at a time to form larger runs and it takes $N/B\lceil \log_k N/M \rceil$ disk I/Os. Thus the k-way merge-sort takes $N/B + N/B\lceil \log_k N/M \rceil$ disk I/Os in total.

In this paper, we propose an external sorting algorithm in one disk model with $2N/B$ reads and $2N/B$ writes for $M \geqslant N/B + (NB)^{1/2}$. The internal phase here performs the same internal sorting as in the external merge sort. In the meantime, the priority queues are used to keep the information of blocks in the main memory for reducing the I/O operations of the external phase. Our sorting algorithm totally reads each record twice (once for producing runs and once for merging) and writes each record twice. We shall show that our algorithm has the optimal disk I/O complexity. The essential problem is to select the parts of input file that are kept in the main memory at each time. For this purpose, the block is fetched as it is needed and removed from memory only when its final position is known.

Our external sorting algorithm has the merge process but it differs from k-way merge-sort and the traditional merge sort in two ways.

(1) Our algorithm constructs the priority queues for external phase, then the block can be fetched in priority order to reduce disk I/O in external phase.
(2) Traditional merge sort merges runs in pairs.

The k-way merge-sort merges k runs at a time to form larger runs but our algorithm merges the blocks by using the information in the priority queues.

Table 1 will offer an idea for the difference between our algorithm and k-way merge-sort by comparing the number of disk I/Os where the record size is 4 bytes. We assume $N = 1000$ (MByte), $M = 5$ (MByte) and $B = 1$ (KByte). The relation between N, M, and B fits the memory constrain of our algorithm.

The k-way merge-sort will approach the same number of disk I/Os only when $k = N/M$. In external

Table 1
Comparing the number of disk I/O between our algorithm and k-way merge-sort

	The number of disk I/O
Our algorithm	2×10^6
2-way merge-sort	9×10^6
5-way merge-sort	5×10^6
10-way merge-sort	4×10^6
15-way merge-sort	3×10^6
20-way merge-sort	3×10^6
30-way merge-sort	3×10^6
40-way merge-sort	3×10^6
50-way merge-sort	3×10^6

problem, the file N is several orders of magnitude larger than M, so the number of runs, N/M, is very large. In [7], the analysis demonstrated that the total seek time grows linearly in k. Minimizing the number of level by selecting k as large as possible is not a good idea because the k-way merge-sort will become impractical. The data is sorted logically means that the kth element can be selected in constant time according to the information in main memory but the file is not sorted physically in the disk. Our algorithm takes advantages if the data ranges of the blocks are mutually non-overlapping (say reversed order, sorted and the others) after the internal phase. Then the data can be sorted logically by merging the priority queues in main memory. In such cases, our algorithm takes only few internal works and disk I/Os in external phase, on the contrast the k-way merge-sort does not take any advantage.

The remainder of this paper is organized as follows. Section 2 is devoted to the new sorting algorithm. In Section 3, the correctness and analysis of our algorithm are shown. Finally, concluding remarks are provided in Section 4.

2. The algorithm

Our sorting algorithm includes two phases—the internal phase and the external phase. In the internal phase, it divides input files into subfiles that can fit into the main memory and sort them by using some internal sorting algorithm. We call each sorted subfile as a run and the total number of runs is N/M. After generating run i, the pointers of the blocks are kept in a min-priority queue Q_i using the blocks' smallest values as their keys, where $1 \leqslant i \leqslant N/M$. The information in the priority queues will be very useful for the external phase to reduce the number of disk I/Os. According to the priority queues, the blocks can be read in order from the disk and written to their sorted position in the external phase. Aggarwal and Vitter [1] proved that the optimal disk I/Os for external sort is

$$\mathrm{O}\left(\frac{N\log(N/B)}{B\log(M/B)}\right)$$

in one disk model. If $M \geqslant (NB)^{1/2}$, the optimal disk I/Os become $\mathrm{O}(N/B)$. Our internal phase needs N/B reads and N/B writes and the external phase also needs N/B reads and N/B writes. Our sorting algorithm takes $2N/B$ reads and $2N/B$ writes and it is also efficient in internal work.

Overall, our algorithm is based upon the above idea. Without loss of generality, we assume that each record of the file is distinct and our goal is to sort these records in ascending order. In the internal phase, M/B blocks are read from disk to main memory then the algorithm selects an internal sorting method, such as quick sort or bubble sort, to sort these blocks to produce runs. Each run consists of M/B blocks. Before writing a run to the disk, the algorithm constructs the priority queue Q_i for each run i. Let $\mathit{Front}(Q_i)$ denote the smallest key value in Q_i. The number of priority queues is equal to the total number of runs, N/M. In the external phase, the algorithm uses $\mathit{Front}(Q_i)$ to decide the order of blocks to be read into main memory. The corresponding block of $\min_i\{\mathit{Front}(Q_i)\}$ will be read into main memory first. When one block is read into main memory, the algorithm chooses the smallest record of this block as a pivot and compares the pivot with the records already exist in main memory. The records in main memory which are smaller than the pivot can be written back to the disk in block. The algorithm then continues to find the next block by using $\mathit{Front}(Q_i)$ and to read the block into main memory. In addition, the pivot is chosen; some records in final positions are written to disk and then the merge process is performed until all the blocks are processed. During the external phase,

each block has to be read into main memory once, thus there are N/B reads in the external phase.

Our algorithm is stated as follows:

Begin
For $i = 1$ **To** N/M **Step** 1 /* Internal Phase */
  Read next M records into main memory;
  Use internal sorting to produce run i;
  Construct a priority queue Q_i with M/B keys
    which are the smallest records of M/B
    blocks in run i, respectively;
  Write run i to the disk;
End For
Clear buffer $b1$; /* External Phase */
Repeat
  Let $K_{\min} = \min_i\{\mathit{Front}(Q_i)\}$;
  Let $B_{\min}$ = the corresponding block of $K_{\min}$;
  Let $Q_{\min}$ = the queue with the key $K_{\min}$;
  Read block $B_{\min}$ into buffer $b2$;
  Delete the smallest element of $Q_{\min}$;
  Let *pivot* = the smallest record of $b2$;
  Let $S = \{X \mid X \leqslant \mathit{pivot}$ and $X \in b1\}$;
  While $|S| \geqslant B$
    Let S' = the first B smallest records of S
      in the ascending order;
    Write S' to disk;
    $S = S - S'$;
  End While
  Merge the records in $b1$ and $b2$, and
    place the results into $b1$;
Until all Q_i are empty
  Write $b1$ to disk;
End

3. Analysis of the algorithm

In this section, we shall prove the correctness, I/O complexity and memory requirement of our algorithm. Below we will show the reasons why this algorithm works.

Lemma 1. *The sorting algorithm takes* $2N/B$ *reads and* $2N/B$ *writes when provided sufficient memory resources.*

Proof. To generate one run needs M/B reads and M/B writes because there is M records in one run. In the internal phase, the total number of reads for generating N/M runs is $(M/B) \times (N/M) = N/B$. In the external phase, one block is read into main memory in one iteration and in the ascending order of the key of the min-priority queue. Each block needs to be read exactly once in external phase, thus it is easy to know that the external phase performs N/B reads and N/B writes. Thus, our sorting algorithm takes $2N/B$ reads and $2N/B$ writes. □

Lemma 2. *After the external phase, all of the records in the disk are sorted when provided sufficient memory resources.*

Proof. The records in each run are in the ascending order after the internal sorting. In the external phase, the algorithm considers a block $B_{\min}$ in an iteration. All of the records in memory buffer $b1$, which are smaller than in pivot are placed in the disk and in their final positions after iteration. We know that the pivot of the nth iteration is smaller than the pivot of $(n+1)$th iteration in the external phase. After each block is read into main memory and proceed in the external phase, all of the records will be written to disk and in their final positions. □

Next we will show the memory requirement of the algorithm.

Lemma 3. *The amount of main memory needed for our sorting algorithm is at least* $N/B + (NB)^{1/2}$.

Proof. First of all, we consider the memory requirement for the priority queue Q_i. There are N/M queues and each queue stores M/B records. The total memory space for N/M queues is N/B. Therefore, M will be at least N/B. We can show later that there are at most N/M blocks kept in main memory for the external phase, we get $M \geqslant (N/M) \times B$ for the external phase. It turns out that $M \geqslant (NB)^{1/2}$. Based upon the above discussion, we can conclude that $M \geqslant N/B + (NB)^{1/2}$.

In the following, we show that in the external phase, at most N/M blocks are needed to keep into main memory. Suppose the new pivot is from run j and one of the blocks kept in the memory is also from run j. This block will be written back to disk because the new pivot is larger than all records in this block. It

turns out that there is only one block from run j kept in the memory. Thus these blocks kept in the memory must be from different runs. Therefore at most N/M blocks will be left in the main memory. □

Theorem. *The algorithm performs optimal* $O(N/B)$ *disk I/Os for* $M \geqslant N/B + (NB)^{1/2}$.

Proof. The optimal of disk I/Os for external sort is

$$O\left(\frac{N \log(N/B)}{B \log(M/B)}\right)$$

in one disk model (see [1]). If $M \geqslant (NB)^{1/2}$, it becomes $O(N/B)$. By Lemmas 1 and 2, we get that our sorting algorithm takes $2N/B$ disk I/Os for $M \geqslant N/B + (NB)^{1/2}$. Thus, our algorithm is optimal. □

4. Conclusion

We have proposed an optimal external sorting algorithm by using the elegant sampling technique to optimize disk I/Os. The algorithm takes exactly $2N/B$ reads and $2N/B$ writes and we have proved the algorithm sorts all records. Table 2 lists the file size and corresponding memory size for running our algorithm, where the block size and record size are 1 K bytes and 4 bytes, respectively. Our external algorithm takes less I/O operations than k-way merge-sort for $M \geqslant N/B + (NB)^{1/2}$ but it can be only performed on one-disk model. It is a trade-off. Our algorithm also has a restrict relation between N and M as shown in Table 2 which is available in the computer environment today.

Table 2
The file size and corresponding memory size for running our algorithm

N/file size (MB)	M/memory size (MB)
10	$\geqslant$ 0.14
100	$\geqslant$ 0.72
500	$\geqslant$ 2.70
1,000	$\geqslant$ 5.00
2,000	$\geqslant$ 9.41
5,000	$\geqslant$ 22.23
10,000	$\geqslant$ 43.16

The interesting open problems are whether the idea of the algorithm is helpful for parallel disk model (PDM) and for parallel external sorting in distributed memory environment to overlap the disk I/O and communication.

References

[1] Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM 31 (9) (1988) 1116–1126.
[2] R.D. Barve, E.F. Grove, J.S. Vitter, Simple randomized mergesort on parallel disks, in: Proc. 8th Symposium on Parallel Algorithms and Architectures, Padua, Italy, ACM Press, New York, June 1996, pp. 109–118.
[3] W. Dobosiewicz, Sorting by distributive partitioning, Inform. Process. Lett. 7 (1) (1978) 1–6.
[4] W.R. Dufrene, F.C. Lin, An efficiency sort algorithm with no addition space, Comput. J. 35 (3) (1992).
[5] G.H. Gonnet, Handbook of Algorithms and Data Structures, Addison-Wesley, Reading, MA, 1984, pp. 160–162.
[6] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[7] S.C. Kwan, J. Baer, The I/O performance of multiway mergesort and tag sort, IEEE Trans. Comput. C-34 (4) (1985) 383–387.
[8] E.E. Lindstorm, J.S. Vitter, The design and analysis of Bucket-Sort for bubble memory secondary storage, IEEE Trans. Comput. C-34 (3) (1985) 218–233.
[9] M.C. Monard, Projecto e Analise de Algorithm de Classificacao Externa Baseados na Estrategia di Quicksort, Ph.D. Thesis, Pontificia Univ. Catolica, Rio de Janeiro, Brazil, 1980.
[10] D. Motzkin, C. Hansen, An efficient external sorting with minimal space requirement, Internat. J. Comput. & Inform. Sci. 11 (6) (1982) 391–392.
[11] M.H. Nodine, J.S. Vitter, Greed sort: An optimal sorting algorithm for multiple disks, J. ACM 42 (4) (1995) 919–933.
[12] Singh, T.L. Naps, Introduction to Data Structure, West Publishing Co., St. Paul, MN, 1985.
[13] I. Verkamo, External quicksort, Performance Evaluation 8 (1988) 271–288.
[14] I. Verkamo, Performance comparison of distributive and mergesort as external sorting algorithms, J. Systems Software 10 (1989) 187–200.
[15] J.S. Vitter, E.A.M. Shriver, Algorithm for parallel memory, I: Two-level memories, Algorithmica 12 (2–3) (1994) 110–147.
[16] B.W. Weide, Statistical methods in algorithm design and analysis, Technical Report CMU-CS-78-142, Carnegie-Mellon University, 1978, pp. 3-30–3-39.