Greed Sort:
An Optimal External Sorting Algorithm for Multiple Disks

Mark H. Nodine and *Jeffrey Scott Vitter*

Technical Report No. CS–90–04
February 1990

Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks*

Mark H. Nodine and *Jeffrey Scott Vitter*

Dept. of Computer Science
Brown University
Providence, R. I. 02912–1910

Abstract

We present an optimal deterministic algorithm for external sorting on multiple disks. Our measure of performance is the number of input/output (I/O) operations. In each I/O, each disk can simultaneously transfer a block of data. Our algorithm improves upon the randomized optimal algorithm in [ViS] and the (non-optimal) commonly-used technique of disk striping. The code is simple enough for easy implementation.

*Support was provided in part by an NSF Presidential Young Investigator Award CCR–8906419 with matching funds from IBM, by NSF research grant DCR–8403613, and by ONR grant N00014–83–K–0146, ARPA Order No. 4786.

1 Introduction

Sorting consumes roughly 20 percent of computing resources in large-scale installations [LiV]. Of particular importance is external sorting, in which the records to be sorted are too numerous to fit in the processor's main memory and instead are placed on secondary storage. The bottleneck in external sorts is the input/output (I/O) operations. Typically data are transferred in units of *blocks*, which may consist of several kilobytes. This approach takes advantage of the fact that the seek time is usually much longer than the time needed for transferring a record of data once the head is positioned. An increasingly popular way to get further speedup is to use many disk drives working in parallel. This is exactly the approach taken, for example, in TWA's flight reservation system [GiS].

Initial work in the use of parallel block transfer for sorting was done by Aggarwal and Vitter [AgV]. In their model, they considered the parameters

$$\begin{aligned} N &= \#\text{ records in the file} \\ M &= \#\text{ records that can fit in internal memory} \\ B &= \#\text{ records per block} \\ P &= \#\text{ blocks transferred per I/O} \end{aligned}$$

where $1 \le B \le M < N$ and $2PB \le M$. In each I/O, P blocks of B records can be transferred simultaneously. They proved that the average-case and worst-case number of I/Os required for sorting is[1]

$$\Theta\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right). \tag{1}$$

Their lower bound is based on routing arguments. They gave two algorithms, a modified merge sort and a distribution sort, that each achieved the optimal I/O bounds. Likewise, they showed that the number of I/Os required to transpose a $p \times q$ matrix is

$$\Theta\left(\frac{N}{PB}\left(1+\frac{\log\min\{M,\min\{p,q\},N/B\}}{\log(M/B)}\right)\right). \tag{2}$$

Vitter and Shriver considered a more realistic *P-disk model*, in which the secondary storage is partitioned into P physically distinct disk drives [ViS]. (Note that each head of a multi-head drive could count as a distinct disk in this definition, as long as each could operate independently of the other heads on the drive.) In each I/O operation, each of the P disks can simultaneously transfer one block of B records. Thus, P blocks can be transferred per I/O, as in the [AgV] model, but only if no two blocks access the same disk. This assumption is very reasonable in light of the way real systems are constructed. Vitter and Shriver presented a randomized version of

[1] We use the notation $\log x$ to denote the quantity $\max\{1, \log_2 x\}$.

distribution sort using two interesting partitioning techniques. Their algorithm meets the I/O bound (1) for the more lenient model of [AgV], and thus the algorithm is optimal. They posed as an open problem the question of whether there is an optimal algorithm that is deterministic. They also showed that matrix transposition can be done optimally in the P-disk model, achieving the bound (2).

Disk striping is a commonly-used technique in which the P disks are synchronized, so that the P blocks accessed during an I/O are located at the same relative position on each disk. This technique effectively transforms the disks into a single disk with larger block size $B' = PB$. Merge sort combined with disk striping is deterministic, but the number of I/Os used can be much larger than optimal, by a multiplicative factor of $\log(M/B)$.

In the area of parallel computing, Leighton introduced the columnsort algorithm as an optimal sorting algorithm on linear-sized sorting networks [Lei]. It was pointed out in [AgV] that columnsort can be used for optimal sorting with respect to I/O if N and B are not too large. Since columnsort is a (non-adaptive) sorting network algorithm, its I/O schedule can be pre-specified independently of the data to take full advantage of parallel block transfer.

In this paper, we answer the open question posed in [ViS] and present an optimal deterministic sorting algorithm. The algorithm, which we call *Greed Sort*, is an interesting variant of merge sort. In each merge pass, a priority scheme guarantees that each record ends up sufficiently close to its correct position, so that another pass of columnsort can complete the merging.

2 The Greed Sort Algorithm

We start by defining an order on the locations on the disks. Let $B_{i,j}$ denote the ith block stored on disk j, and let $R_{i,j,k}$ denote the kth record in $B_{i,j}$. We order the locations so that

- Record $R_{i,j,k}$ precedes $R_{i,j,k+1}$ for all i, j, k.
- Block $B_{i,j}$ precedes block $B_{i,j+1}$ for all i, j.
- Block $B_{i,P}$ precedes block $B_{i+1,1}$ for all i.

Now we can describe the sorting problem.

Problem instance: A series of N records, each containing a key field, located in the first N locations on the disk.

Goal: To permute the records so that the key fields are in non-decreasing order in the first N locations on the disk.

Disk 1	1	26	42	71	83	94
Disk 2	4	29	47	77	85	96
Disk 3	7	31	60	79	89	98
Disk 4	14	32	68	80	91	99

Figure 1: An example of a sorted list with $B = 1$.

Figure 1 gives an example of a sorted list with $B = 1$.

Our Greed Sort algorithm is based on merge sort. We create initial runs of size N/M by repeatedly reading in a memoryload, sorting it, and writing it out again to the disks. In each subsequent pass, we merge together at least $\sqrt{M/B}$ sorted lists, called *runs*, to form a single sorted list. Each pass will be shown to take $O(N/PB)$ I/Os, giving us a total I/O bound of

$$O\left(\frac{N}{PB}\left(1+\log_{\sqrt{M/B}}(N/M)\right)\right) = O\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right),$$

which is optimal, by (1). The complete analysis follows in Section 4.

The basic idea behind greed sort is the following: Let us assume that each run to be merged is stored consecutively on disk, each beginning on disk 1. In each read operation, the "best" available block on each disk is read into internal memory. The set of blocks is then sorted and output in parallel. This operation results in an "approximately merged" list. The crucial observation is that the records are within $P\sqrt{MB}$ positions of their correct locations. A single pass over the approximately merged list using columnsort, which we describe later, produces a totally sorted list.

Since all the elements within a run are sorted, consecutive blocks within a run on the same disk are in non-decreasing order. Thus, the best available block on a given disk must be the next unread block on that disk from some run. The collection of next unread blocks, one for each (run, disk) pair, is called the *candidate set* of blocks. There are P disks and $\sqrt{M/B}$ runs, so the candidate set has cardinality $P\sqrt{M/B}$.

We assume for convenience that the runs are separated on each disk by blocks containing a number larger than any being sorted. The algorithm uses $mark[i,j]$ to keep track of the next block to be read for run i on disk j. The set of all blocks $mark[i,j]$ comprises the candidate set. The maximum and minimum key fields of block $mark[i,j]$ are stored in $tops[i,j]$ and $bots[i,j]$. We do our I/O operations into the buffer b, which consists of P smaller buffers. Buffer $b[j]$, for $1 \le j \le P$, holds B records from disk j. We use the notation "|| write" and "|| read" to refer to parallel transfers of PB records to and from the disks, respectively.

The algorithm uses columnsort to complete each merge pass. Columnsort is easiest to visualize as sorting a matrix with r rows and c columns, with the requirement that

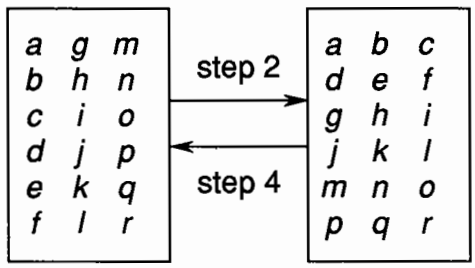

Figure 2: The transpose used by Steps 2 and 4 of columnsort.

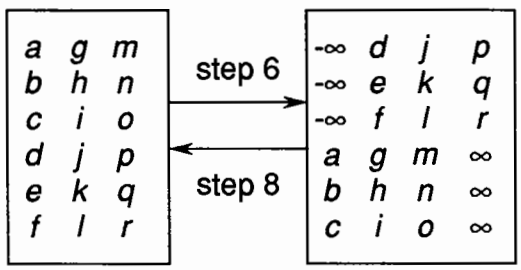

Figure 3: The shift operation used in Steps 6 and 8 of columnsort.

c divides r and $r > 2(c-1)^2$. It has eight steps, of which the odd-numbered steps are all the same: sort all the elements in each column. Steps 2 and 4 are a transpose operation as shown in Figure 2. Steps 6 and 8 amount to a cyclical shift by $r/2$, and are shown in Figure 3. The values ∞ and $-\infty$ refer to a elements chosen to be larger than and smaller than any being sorted, respectively.

In the code below, we compute *bestrun* by minimizing over pairs of numbers using a lexicographic ordering. In a lexicographic ordering, $(a_1, b_1) < (a_2, b_2)$ if $a_1 < a_2$ or if both $a_1 = a_2$ and $b_1 < b_2$.

```
{ Create the initial runs }
repeat N/M times
    || read the next M records into internal memory, PB at a time
    sort the M records internally
    || write the M records back onto disk, PB at a time
end repeat
```

```
repeat until only 1 run is left
    { Merge together √(M/B) runs at a time }
    repeat until all runs have been processed
        pick the first √(M/B) runs that have not been processed
        T = total # of records in the √(M/N) runs
        for i := 1 to √(M/B)
            || read the first P blocks of run i into buffer b
            for j := 1 to P
                mark[i,j] := 1
                tops[i,j] := max(b[j])
                bots[i,j] := min(b[j])
            end for
        end for

        repeat until all elements of all runs have been processed
            { Do an approximate merge: find the best run to read on each disk }
            bestrun[j] := i such that (tops[i,j], bots[i,j]) is a lexicographical minimum

            || read block mark[bestrun[j],j] from run bestrun[j]
            sort the records internally
            || write the sorted records

            || read block mark[bestrun[j],j]+1 from run bestrun[j] into buffer b[j]
            for j := 1 to P
                mark[bestrun[j],j] := mark[bestrun[j],j]+1
                tops[bestrun[j],j] := max(b[j])
                bots[bestrun[j],j] := min(b[j])
            end for
        end repeat

        { Do a restorative pass to put everything into sorted order }
        L := P√(MB)
        for n := 0 to 2T/L − 1
            use columnsort to sort elements (n/2)L + 1, (n/2)L + 2, ..., (n/2)L + L of the run
        end for
    end repeat
end repeat
```

3 Proof of Correctness

The proof of correctness of the Greed Sort algorithm is based on the well-known 0-1 Sorting Lemma. This lemma can be found in several souces [Akl, Knu], and is reproduced here for convenience.

Lemma 1 (0-1 Sorting) *A sorting algorithm using only comparison and exchange operations is valid if it is valid for the special case in which the key values are taken from the set* $\{0, 1\}$.

Proof: The proof is by contradiction. Assume that the algorithm sorts 0s and 1s correctly, but there is some permutation of numbers for which it fails. Let i be the first record that is out of place. Then there is some number j occurring after i with $j < i$. Now apply the function

$$f(n) = \begin{cases} 0 & n < i; \\ 1 & n \geq i, \end{cases}$$

to the original set of numbers. The exchanges done by the algorithm on this set of 0s and 1s correspond to those done by the algorithm applied to the original numbers. Swapping two 0s or two 1s is immaterial, but swapping a 0 with a 1 corresponds to swapping a smaller number with a larger one. It follows that the algorithm will fail to sort these 0s and 1s correctly, contradicting our original assumption. □

Now we are in a position to look at how the 0-1 Sorting Lemma applies to the merge operation.

Theorem 1 *Each stage of merging in the algorithm correctly merges* $\sqrt{M/B}$ *lists.*

Proof: Figure 4 shows a typical scenario for our greedy merging algorithm. In the figure, each "0" refers to an entire block of B 0s and each "1" refers to a block containing at least one 1. Let T be the total number of records in all the runs and let the ith run contain α_i 0s followed by β_i 1s. Let $R = \sqrt{M/B}$ be the number of runs. The algorithm starts by writing $\sum_{1 \leq i \leq R} (PB)\lfloor \alpha_i/(PB) \rfloor$ 0s. At this point, no run has as many as PB 0s remaining. The merge process then produces a stretch of mixed 0s and 1s. The lexicographic ordering ensures that any block containing a 0 will be read before any block that is full of 1s on a given disk. Since there are at most $\sqrt{M/B}$ blocks containing 0s on any disk, the next $\sqrt{M/B}$ parallel I/Os will read and write all the remaining 0s. It follows that the unsorted stretch has a length $L < PB\sqrt{M/B} = P\sqrt{MB}$. After this unsorted stretch, the algorithm will output $\sum_{1 \leq i \leq R} (PB)\lfloor \beta_i/(PB) \rfloor$ 1s to complete the approximate merge. Thus if we let $L = P\sqrt{MB}$ and sort elements $\frac{n}{2}L + 1, \ldots, \frac{n}{2}L + L$ for all $0 \leq n < 2T/L$, then any set of 0s and 1s will be sorted. Thus, by the 0-1 sorting lemma, any set of elements can be sorted, and we have demonstrated the correctness of the merge. □

Corollary 1 *Greed sort correctly sorts any set of elements.*

Proof: This corollary follows from Theorem 1 and the fact that it is a classical merge sorting algorithm. □

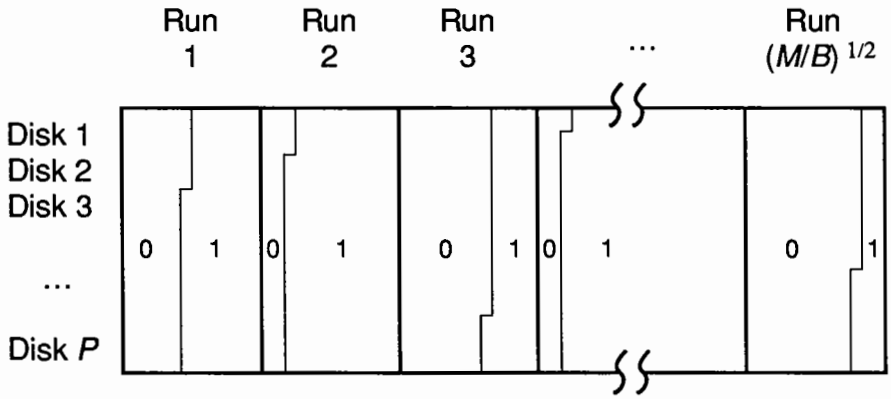

Figure 4: A typical layout of 0s and 1s in runs for greedy merging.

4 Analysis of the Algorithm

First, we show that the amount of internal storage space required to keep the arrays *mark*, *tops*, and *bots* is not excessive.

Theorem 2 *The amount of internal memory space needed for the data structures of greed sort is* $O(M^{3/4})$.

Proof: The number of runs that we must merge together in order to obtain optimal performance is $\sqrt{M/B}$. As we pointed out in Section 2, the candidate set requires a total of $P\sqrt{M/B}$ key fields to be kept in internal memory to decide what blocks to read next. At first appearance, it seems that if $P = M/2$ and $B = 1$, obeying the restriction that $2PB \leq M$, then this will require $O(M^{3/2})$ elements to be stored, which is clearly impossible. However, we can safely assume that $P \leq \sqrt{M}$ because the optimal bounds given in (1) with respect to B and P are the same as those with respect to $P' = \sqrt{M}$, $B' = BP/P'$, namely,

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

In other words, we can cluster our P disks into BP/P' clusters of $\sqrt{M}$ disks synchronized together. Each cluster acts like a logical disk with block size $B' = BP/P'$. Thus, we need at most

$$P\sqrt{M/B} \leq \sqrt{M}\sqrt{M/\sqrt{M}} = M^{3/4}$$

internal storage space. Technically, each memory location holds an entire record, whereas we only need to store key fields in *tops* and *bots* and numbers from 1 to $N/\sqrt{M/B}$ in *mark*, but since a record is at least as large as a key field, we still get a storage that is $O(M^{3/4})$. □

In addition to requiring space to store the maximum keys in the candidate set, we need enough room to read at least PB records. But $PB \leq M/2$, so we can use half the space for our buffers and half for the data structures, for $M \geq 16$.

In order to demonstrate that greed sort has an optimal I/O bound, we need to analyze the I/O efficiency of columnsort, which we use as a subroutine.

Theorem 3 *Columnsort sorts N records with $O(N/PB)$ parallel I/O operations if $N \leq P\sqrt{MB}$.*

Proof: First we show that columnsort produces a correctly sorted sequence for this value of N. We consider $r = M$ to be the number of rows in the array to be sorted. If there are $c = \sqrt{M/2}$ columns, the columnsort requirement that $r \geq 2(c-1)^2$ is met. So we have a total of of $M^{3/2}/\sqrt{2}$ records. But

$$N \leq (P\sqrt{B})\sqrt{M} \leq PB\sqrt{M} \leq \frac{M^{3/2}}{2} < \frac{M^{3/2}}{\sqrt{2}},$$

making use of our assumption that $2PB \leq M$. Thus, the columnsort works correctly.

Now we find the number of parallel block transfers needed to do the columnsort. Steps 1, 3, 5, and 7 (the column sorting steps) can each be done by reading in the M records of a column PB at a time, sorting them, and writing them back out PB at a time. Each record is read and written once, so the total number of I/O operations for all of these steps is $8N/PB$.

Steps 6 and 8, although useful in describing the algorithm, are not necessary when implementing columnsort using parallel I/O. The whole effect of Steps 6–8 is to offset by $M/2$ the column boundaries as used by the sorting in Step 7; this can be done by reading a shifted set of M records from the disks, PB at a time, in Step 7. Steps 6 and 8 thus contribute nothing to the number of I/O operations.

Finally, we need to show that the transpose operation in Steps 2 and 4 can be done with $O(N/PB)$ I/Os. Although this transpose operation is not the usual matrix transposition (in which transposing a $p \times q$ matrix gives a $q \times p$ matrix, the actual placement of records on the disk is exactly the same as for the usual matrix transposition; we simply adjust our idea of where the column boundaries are. Thus, the algorithm of [ViS] that achieves the optimal I/O bound is applicable. Here we have $p = M$ and $q = N/M = P\sqrt{B/M}$. The resulting number of I/Os is

$$\begin{aligned} O\left(\frac{N}{PB}\frac{\log\min\left\{M, \min\left\{M, P\sqrt{B/M}\right\}, P\sqrt{MB}/B\right\}}{\log(M/B)}\right) \\ = O\left(\frac{N}{PB}\frac{\log\left(P\sqrt{B/M}\right)}{\log(M/B)}\right) \\ = O\left(\frac{N}{PB}\right). \end{aligned}$$

□

Corollary 2 *Each merging step requires $O(N/PB)$ I/Os.*

Proof: The greedy merge reads each record twice (once for updating *tops* and *bots* and once for merging) and writes each record once, taking full advantage of parallel block transfer. The columnsort routine is called $2N/k$ times, each time using $O(k/PB)$ I/Os, for a value of k that differs from pass to pass. Thus the total number of parallel I/Os is $O(N/PB)$. □

We are finally ready to prove the I/O bound for greed sort.

Theorem 4 *The number of parallel I/O operations used in greed sort is*

$$O\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right).$$

Proof: The algorithm uses $O(N/PB)$ parallel I/Os in creating the N/M initial runs each of size M. It then goes through a series of merging passes, each of which merges $R = \sqrt{M/B}$ runs into one longer run. The number of merging passes needed will clearly be $\log_R(N/M)$. Let N_k denote the size of the lists to be merged by the kth merging pass, with $N_1 = M$. Then the number of I/Os required by a single merge in the kth merging pass is $O(RN_k/PB)$. Since there are N/RN_k sets of R lists to be merged in the kth pass, the total number of I/Os is

$$\begin{aligned} O\Big(\frac{N}{PB} + \sum_{1\le k\le \log_R(N/M)} \frac{RN_k}{PB}\frac{N}{RN_k}\Big) &= O\Big(\sum_{1\le k\le \log_R(N/M)} \frac{N}{PB}\Big) \\ &= O\left(\frac{N}{PB}\left(1+\frac{\log(N/M)}{\log R}\right)\right) \\ &= O\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right). \end{aligned}$$

□

5 Conclusions

Greed sort is an optimal deterministic algorithm for external sorting applications that is easy to implement in practice. The advantages of greed sort are:

- It exhibits optimal I/O performance under realistic high-performance storage systems with multiple disks.
- It is deterministic with a worst-case guarantee on performance. Its performance can be substantially better than that of the commonly-used technique of disk striping.

- It is simple and straightforward to implement.

This Greed Sort algorithm also has applications to the parallel hierarchical memory systems considered in [ViS], which will be explored in a forthcoming extension of this paper.

References

[AgV] A. Aggarwal & J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* (September 1988), also appears in *Proceedings of 14th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987.

[Akl] S. G. Akl, *Parallel Sorting Algorithms*, Notes and Reports in Computer Science and Applied Mathematics #12, Academic Press, Inc., Orlando, 1985.

[GiS] D. Gifford & A. Spector, "The TWA Reservation System," *Communications of the ACM* 27 (July 1984), 650–665.

[Knu] D. Knuth, in *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Lei] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers* C-34 (April 1985), 344–354.

[LiV] E. E. Lindstrom & J. S. Vitter, "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Transactions on Computers* C-34 (March 1985), 218–233.

[ViS] J. S. Vitter & E. A. M. Shriver, "Optimal Disk I/O with Parallel Block Transfer," *Symposium on the Theory of Computing* (May 1990).