Greed Sort: Optimal Deterministic Sorting on Parallel Disks

MARK H. NODINE

Motorola Cambridge Research Center, Cambridge, Massachusetts

AND

JEFFREY SCOTT VITTER

Duke University, Durham, North Carolina

Abstract. We present an algorithm for sorting efficiently with parallel two-level memories. Our main result is an elegant, easy-to-implement, optimal, *deterministic* algorithm for external sorting with D disk drives. This result answers in the affirmative the open problem posed by Vitter and Shriver of whether an optimal algorithm exists that is deterministic. Our measure of performance is the number of parallel input/output (I/O) operations, in which each of the D disks can simultaneously transfer a block of B contiguous records. We assume that internal memory can hold M records. Our algorithm sorts N records in the optimal bound of $\Theta((N/BD)\log(N/B)/\log(M/B))$ deterministically, and thus it improves upon Vitter and Shriver's optimal randomized algorithm as well as the well-known deterministic but nonoptimal technique of disk striping. It is also practical to implement.

Categories and Subject Descriptors: B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids—*worst-case analysis*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*; E.5 [**Files**]: *sorting/searching*

General Terms: Algorithms

Additional Key Words and Phrases: *I/O* complexity, parallel disks, parallel *I/O*, merge sort

---

The work of M. H. Nodine was supported in part by an IBM Graduate Fellowship and by a National Science Foundation Presidential Young Investigator Award CCR 90-47466 with matching funds from IBM Corporation.

The work of J. S. Vitter was supported in part by a National Science Foundation Presidential Young Investigator Award CCR 90-47466 with matching funds from IBM Corporation, by National Science Foundation grant CCR 90-07851, by the U.S. Army Research Office under grant DAAL03-91-G-0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052 and ARPA order 8225.

Authors address: M. H. Nodine, Motorola Cambridge Research Center, One Kendall Square, Building 200, Cambridge, MA 02139, e-mail: nodine@mcrc.mot.com; J. S. Vitter, Department of Computer Science, Duke University, Durham, NC 27708-0129, e-mail: jsv@cs.duke.edu

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1995 ACM 0004-5411/95/0700-0919 $03.50

1. *Introduction*

Sorting is reputed to consume roughly 20 percent of computing resources in large-scale installations [Knuth 1973; Lindstrom and Vitter 1985]. Although we can argue about the exact percentage, there is no doubt that sorting and related operations are significant components of information processing. 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 must be stored on disk. The bottleneck in external sorting is the time needed for the input/output (I/O) operations. The reason that the I/O becomes the bottleneck is that *CPU* speeds are much faster than disk speeds, and moreover have been growing at a much faster rate than disk speeds over the last decade. A tendency to use parallel processors in large-scale applications further exacerbates the mismatch between the computational and I/O capabilities.

There are several approaches that are taken to mitigate the I/O bottleneck. The first avenue, used in almost every system, is to transfer data in large units or *blocks*; this blocking takes advantage of the fact that the seek time is usually much longer than the amount of time needed to transfer a record of data once the disk's read/write head is in position. A second increasingly popular (and necessary!) way to alleviate the I/O bottleneck is to use many disk drives working in parallel.[1] This method greatly increases the bandwidth to the I/O subsystem (by about a factor of the number of disks) while not appreciably affecting the latency. The challenge, therefore, in using parallel disks is to take advantage of the increased bandwidth by making certain that the items to be read and written during I/Os are evenly distributed over the disks.

Initial work in the use of parallel block transfer for sorting was done by Aggarwal and Vitter [1988]. They considered a two-level memory model in which D physical blocks, each consisting of B contiguous records, can be transferred simultaneously in a single I/O into a primary memory capable of holding M records (see Figure 1). Their model generalized the initial work of Floyd [1972] and Hong and Kung [1981]. Aggarwal and Vitter derived matching upper and lower bounds for their model, finding that the number of parallel I/Os required to sort N numbers is

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

They showed that the lower bounds are the same as those for permuting N numbers and hold for an arbitrarily powerful adversary, except for the case when M and B are extremely small, in which case they used the comparison model to get the lower bounds. Aggarwal and Vitter [1988] showed the lower bounds for permuting by computing the maximum of permutations that can be accomplished in t I/O steps, and then computing the minimum number of steps that are required to attain all $N!$ permutations of N numbers. They achieved the sorting upper bounds with two different algorithms, one based on Merge Sort and the other based on Distribution (Bucket) Sort. For a treatment of Merge Sort and Distribution Sort, see Knuth [1973].

[1] See, for example, Gibson et al. [1988], Gifford and Spector [1984], Jilke [1986], Maginnis [1987], Patterson et al. [1988], and University of California at Berkeley [1989].

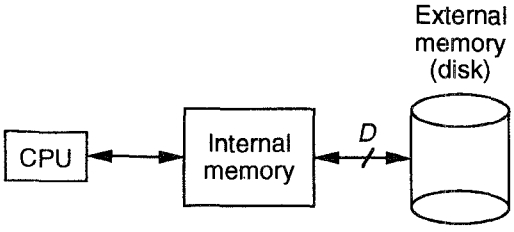

FIG. 1. Aggarwal and Vitter's memory model. The CPU is connected to the internal memory, which in turn has a D-way connection to the disk.

Aggarwal and Vitter's model is somewhat unrealistic, however, because in practice, secondary storage devices cannot transfer any arbitrary set of D blocks simultaneously. Vitter and Shriver [1990] considered the more realistic *parallel disk model*, in which the secondary storage is partitioned into D physically distinct disk drives (see Figure 2). Each head of a multihead drive can count as a distinct disk in this definition, as long as each head can operate independently of the other heads on the drive. In a single (parallel) I/O operation, each of the D disks can simultaneously transfer one block of B records. Thus, D blocks can be transferred per I/O, as in Aggarwal and Vitter's model, but only if no two of the blocks access the same disk. This assumption is reasonable in light of the way real systems are constructed.

Vitter and Shriver presented a randomized version of Distribution Sort using two complementary partitioning techniques. Their algorithm meets the I/O lower bound given earlier for the more lenient model of Aggarwal and Vitter. Since the lower bound also applies to the more restrictive model, the algorithm is optimal. It can outperform the well-known deterministic technique of disk striping by a factor of about $\log M$ as measured by the number of I/Os. Vitter and Shriver posed as an open problem the question of whether there is an optimal algorithm that is deterministic.

From a historical perspective, the idea of sorting using parallel media is not a new one. Even [1974] proposed two algorithms for sorting using parallel tape drives. These algorithms are not very general, however, as they require a system with P processors and $4P$ tape drives. Bitton et al. [1984] later considered the problem of sorting with P processors and P disks, and they devised an elaboration of binary merge sort that uses $N/P \log N/P + \log P$ parallel read operations and the same number of parallel write operations. Each processor requires $O(1)$ internal memory cells in their algorithm, since they simulate a sequence of tree machines with a constant degree of 2. Our model, on the other hand, has only one processor and internal memory with a branching factor of D between internal and external memory. Dealing with this branching factor is what makes the problem hard.

In the next section, we answer the open question posed in Vitter and Shriver [1990] and present for the parallel disk model an optimal deterministic sorting algorithm called *Greed Sort*. It performs a Merge Sort in a greedy way, using a priority scheme during the first part of each merge process to do an "approximate merge" of the runs. A second part of the merge process completes the merging. Oddly enough, the intuitions of Vitter and Shriver [1990] suggested that merge sorting with D disks was particularly difficult to do, as opposed to distribution sorting.

Throughout this paper, the measure of performance is the number of parallel I/Os the algorithm does; we ignore internal computation time. In

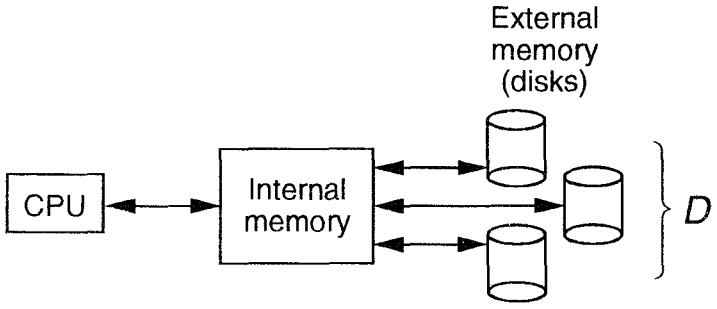

FIG. 2. The parallel disk model. The external memory is partitioned into D disks, all of which may transfer a block of B records in parallel.

practice, though, our algorithm is also very efficient in terms of internal processing. Our algorithm also applies to a model in which each of the D disks is controlled by a separate *CPU* with internal memory capable of storing M/D records, and the D *CPU*s are connected by a network that allows some basic operations (like sorting of the M records in the internal memories) to be performed quickly in parallel. The bottleneck in this model can also be expected to be the I/O.

2. *Greed Sort*

The parameters for our two-level memory model (or parallel disk I/O model) in Figure 2 are

$$N = \# \text{ records in the file;}$$
$$M = \# \text{ records that can fit in primary memory;}$$
$$B = \# \text{ records per block;}$$
$$D = \# \text{ disk drives;}$$

where $M < N$, and $1 \leq DB \leq \lfloor (M - M^{\beta})/2 \rfloor$, for fixed $\beta < 1$. The first constraint means that the problem is too large to fit in memory, and the second that the total number of records that can be transferred in a single parallel I/O, DB, cannot exceed about half of the internal memory locations. Our measure of performance is the number of parallel I/Os; during a parallel I/O, each disk can simultaneously transfer one contiguous block of data.

By a sorted list, we mean one in which the first block appears on track 1 of disk 1, the second block on track 1 of disk 2, and so on, until the Dth block appears on track 1 of disk D. The list then cycles back to the second block of disk 1, and so on, as suggested by Figure 3. In general, the ith block of the sorted list appears on track $\lfloor (i-1)/D + 1 \rfloor$ of disk $((i-1) \bmod D) + 1$. The B records within each block are numbered contiguously by their relative positions in the block.

Our Greed Sort algorithm is a type of Merge Sort. A merge is a process that starts with two (or more) sorted input lists and produces a single completely sorted output list. A sorted list is sometimes called a "run" in this context. The traditional Merge Sort algorithm always merges pairs of lists that are approximately the same size. Thus, to start, the algorithm divides the N records into N

Disk 1	1	9	17	25	33	41
	2	10	18	26	34	42
Disk 2	3	11	19	27	35	43
	4	12	20	28	36	44
Disk 3	5	13	21	29	37	45
	6	14	22	30	38	46
Disk 4	7	15	23	31	39	47
	8	16	24	32	40	48

FIG. 3. An example of a sorted list with $D = 4$, $B = 2$.

lists of length 1, which it merges in pairs to form $\lceil N/2 \rceil$ lists of length at most 2 (all but possibly the last list will have length exactly 2). The next pass merges the lists of length 2 to produce lists of length 4, and so on, until a single list of length N finally results.

Greed Sort differs from the traditional Merge Sort in two ways:

(1) The initial lists have length M instead of length 1. We create these N/M initial input runs (sorted lists) of size M by repeatedly reading in a memoryload of M unprocessed elements, sorting it, and writing it back to the disks.

(2) Traditional Merge Sort merges runs in pairs. Greed Sort merges $R = \sqrt{M/B}/2$ input runs at a time to form larger runs, which are used as input runs for the next pass.

Each pass will be shown to take $O(N/DB)$ I/Os, giving us a total I/O bound of[2]

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

which is optimal, by the lower bound of Aggarwal and Vitter [1988]. The complete analysis is presented in Section 2.2.

THEOREM 2.1. *Greed Sort deterministically sorts $N \geq M$ records with*

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

parallel I/Os, which is optimal.

The key to the success of Greed Sort is being able to do the R-way merge. Let us assume that each of the R runs to be merged is stored consecutively on disk, beginning on disk 1 and cycling through the D disks, as previously described and illustrated in Figure 3. In each parallel read operation, the one or two "best" available blocks from each disk are read into primary memory. We define the best available blocks to be *the block with the smallest minimum key value* and *the block with the smallest maximum key value*. In this phase of the algorithm, *we do an approximate merge on each of the D disks independently*; all of the approximate merges are taking place in parallel. With respect to any disk, we generally read two blocks at each step, comprising two sorted sets of B records. We merge the two blocks in primary memory and write the smallest B records to the output list that we're forming for that disk; we put the largest B records at the front of the run from which the smallest minimum was taken

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

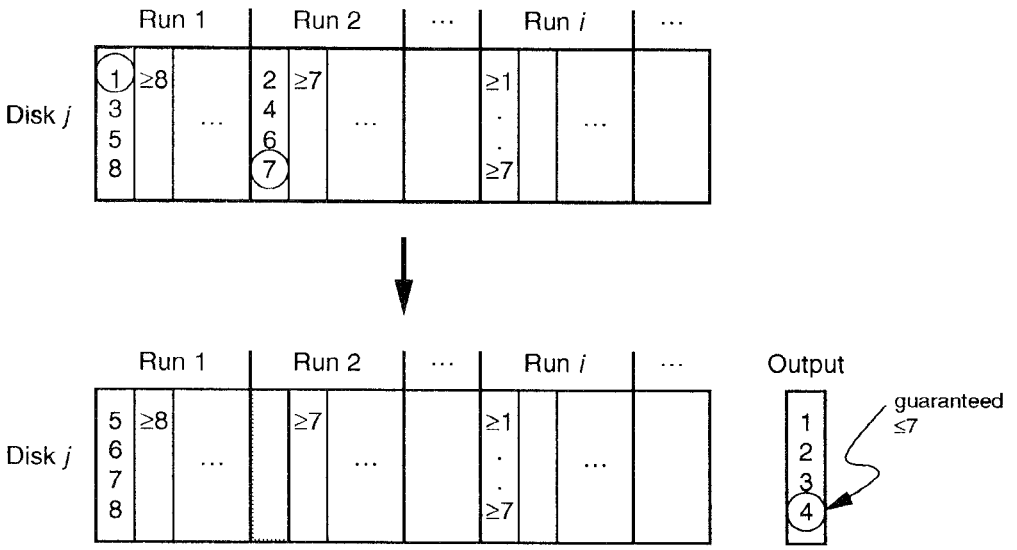

FIG. 4. Assume that Run 1 contains the block with the smallest minimum and Run 2 that with the smallest maximum on some disk j. Then this figure shows what the situation will be after the blocks have been processed.

(note that this run remains sorted after this operation). For example, in Figure 4, Run 1 contains the block with the smallest minimum on disk j and Run 2 the block with the smallest maximum. Merging those two blocks together gives all the numbers from 1 to 8, of which we write the first four to the output, and the other four back to Run 1, which is the run that had the smallest minimum element. Ties are broken arbitrarily. If the blocks with the smallest minimum and the smallest maximum are the same block, we read only the one block and copy it directly to the output list.

Once all the elements in every run on all of the disks have been written to their corresponding output list, we wind up with an "approximately merged" list. The crucial observation, which we prove in Theorem 2.1.2, is that the records are within $RDB = D\sqrt{MB}/2$ positions of their correct sorted locations. By an appropriate use of clustering (or partial striping) throughout the course of the algorithm, we can complete the merge of this approximately merged list by a single pass consisting of several applications of the Columnsort algorithm of Leighton [1985] applied to subfiles of size $D\sqrt{MB}$. We use Columnsort since it is able to sort $N = O(M^{3/2})$ elements with a linear number of I/Os. Then, the next merge begins.

Columnsort is easiest to visualize as sorting into column-major order a matrix with r rows and c columns. For technical reasons, there is a requirement that c divides r and $r > 2(c-1)^2$. Columnsort has eight steps, of which the odd-numbered steps are all the same: sort all the records in each column. In his original article, Leighton [1985] used the (impractical) AKS sorting network [Ajtai et al. 1983] to do the sorts, since he was interested in establishing the existence of a bounded-degree sorting network to sort N numbers in $O(\log N)$ time. The correctness of Columnsort, however, is independent of what algorithm is used to sort the columns. Steps 2 and 4 are a transpose

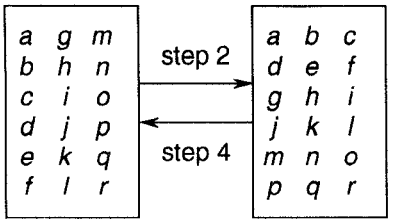

FIG. 5. The transpose used by Steps 2 and 4 of Columnsort.

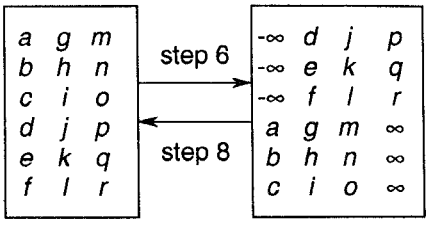

FIG. 6. The shift operation used in Steps 6 and 8 of Columnsort.

operation in which the rows of the transposed matrix are wrapped to maintain the same shape as the original matrix, as shown in Figure 5. This transpose operation leads to the requirement that c divides r. Steps 6 and 8 amount to a shift by $r/2$ in column-major order, with padding by $-\infty$ and ∞ before the first element and after the last element, respectively. This shift operation is illustrated in Figure 6.

For reference, we give the pseudocode for the Greed Sort algorithm in Figure 7. Recall that a sorted list cycles through all the disks. If we look at the "slice" of any run that appears on some particular disk, it is easy to see that the slice is a sorted sublist. Thus, we only need to examine the first unprocessed block on a given disk in any run to find the block with the smallest minimum (or maximum) on that disk from that run. The collection of first unprocessed blocks, one for each (run, disk) pair, is called the *candidate set* of blocks, and it is from this candidate set that we will select the best available blocks. To aid in this process, we can store pointers to the blocks in two priority queues: one using the blocks' largest value as its key and the other using the blocks' smallest value. The implementation details are unimportant from a theoretical standpoint, however, since our model assumes that internal processing time is free. Hence, the amount of time needed to compute the best available blocks does not matter, as long as we have enough room to keep all the information about the candidate set in memory. There are D disks and R runs, so the candidate set has cardinality DR. We only need to keep two elements per block in the candidate set, the largest and the smallest, so we need $O(DR)$ storage. We show in Section 2.2 that there is room for these elements.

We assume for convenience in the pseudocode that the runs are separated on each disk by blocks containing some dummy key value $+\infty$ that is larger in value than any key. The algorithm uses $next[i, j]$ to keep track of the next block of run i to be read from disk j. Thus, the set of all blocks $next[i, j]$ comprises the candidate set. The maximum and minimum key fields of block $next[i, j]$ are stored in $biggest[i, j]$ and $smallest[i, j]$, respectively. We do our I/O operations into the buffers $b1$ and $b2$, which each consist of D smaller buffers, one for each disk. Buffers $b1[j]$ and $b2[j]$, for $1 \leq j \leq D$, each hold B records from disk j; we denote their maximum and minimum keys by $\max(b1[j])$, $\min(b1[j])$,

```
algorithm Greed Sort;
{ Create the initial runs }
repeat N/M times
    read the next M records into primary memory, DB at a time
    sort the M records internally
    write the M records back onto disk, DB at a time

repeat until only 1 run is left
    { Merge together R = √(M/B)/2 input runs at a time }
    R := √(M/B)/2
    { The output runs of the previous stage become the input runs for this stage }
    repeat until all input runs have been processed
        { Assume the next R runs to process are numbered 1, ..., R }
        for i := 1 to R do   { Initialize }
            read in parallel the first D blocks of run i into buffer b1
            for j := 1 to D do
                next[i, j] := 1
                biggest[i, j] := max(b1[j])
                smallest[i, j] := min(b1[j])

        { Do an approximate merge of the R runs }
        repeat until all records of the R runs have been processed
            for j := 1 to D do in parallel
                { Find the best one or two blocks to read from each disk }
                bestrun1[j] := i such that biggest[i, j] is a minimum
                bestrun2[j] := i such that smallest[i, j] is a minimum
                read block next[bestrun1[j], j] of run bestrun1[j] from disk j into buffer b1[j]
                if bestrun1[j] ≠ bestrun2[j] then
                    read block next[bestrun2[j], j] of run bestrun2[j] from disk j into buffer b2[j]
                    merge b1[j] and b2[j] in place internally with the smallest elements in b1
                    write b2[j] to block next[bestrun2[j], j] of run bestrun2[j] on disk j
                write b1[j] to disk j in the output list
                read block next[bestrun1[j], j] + 1 of run bestrun1[j] from disk j into buffer b1[j]
                { Update data structures }
                next[bestrun1[j], j] := next[bestrun1[j], j] + 1
                biggest[bestrun1[j], j] := max(b1[j])   { May be +∞ }
                smallest[bestrun1[j], j] := min(b1[j])   { May be +∞ }
                if bestrun1[j] ≠ bestrun2[j] then
                    smallest[bestrun2[j], j] := min(b2[j])   { The biggest hasn't changed }

        { Do a restorative pass to turn the approximate merge into a complete merge }
        L := RDB
        T := total # of records in the R runs
        for t := 0 to ⌈T/L⌉ − 2 do
            use Columnsort to sort records tL + 1, tL + 2, ..., tL + 2L of the output list
```

FIG. 7. Pseudocode for the Greed Sort Algorithm.

$\max(b2[j])$, and $\min(b2[j])$. In the pseudocode, when we use the construct **do in parallel**, we mean that the I/O within the loop should be done in parallel, not that the actual computation needs to be done in parallel.

The overall structure of Greed Sort, like any Merge Sort algorithm, consists of an outer loop that establishes what size lists are being merged, and an inner loop that merges together lists of that size R at a time. The outer loop terminates when the size of the lists to be merged reaches N.

2.1. PROOF OF CORRECTNESS. The correctness of Merge Sort algorithms in general is easy to establish, since each merge by definition produces a sorted list. Thus, showing the correctness of Greed Sort depends on showing that each merge pass correctly merges the $R = \sqrt{M/B}/2$ runs into a single sorted run.

THEOREM 2.1.1. *Each sequence of an approximate merge followed by Columnsorts in the Greed Sort algorithm correctly merges R runs.*

Theorem 2.1.2 below shows that each record in the approximately merged output list formed from the R runs is at most $L = RDB$ locations from its correct sorted location. Theorem 2.1.13 proves that the Columnsorts on successive overlapping subfiles of size $2L$ complete the sorting. Together, Theorems 2.1.2 and 2.1.13 establish Theorem 2.1.1.

THEOREM 2.1.2. *In the approximately merged output list formed from merging R runs, each record is at most RDB locations from its correct sorted location.*

This theorem is proved using the lemmas below. The main lemmas are Lemma 2.1.11, which limits how far any record can occur in the approximately merged list after a record that has a larger key, and Lemma 2.1.12, which shows that each element is close to its correct sorted location in the approximately merged list.

For notational convenience, we will identify a record with the value of its key. We are justified in doing this since, as mentioned above, the amount of time needed for sorting is the same as that needed for permuting. In other words, the difficulty in sorting is not in establishing the correct permutation for the records, but in doing the actual routing. From this, it follows that doing "key sorting" followed by the actual permutation does not affect the number of I/Os by more than a constant factor, so we might as well cart around the entire record.

In our arguments, we will often speak of the "tth smallest minimum record on disk i" or the "tth smallest maximum record on disk i". By these, we mean the tth smallest record in the set of minimum (respectively, maximum) records in all the blocks in all the runs (before the merging begins). We will denote these elements by $y_{i,t}^{\min}$ and $y_{i,t}^{\max}$, respectively. It would appear from this terminology that we assume the key values are distinct. The proofs, however, work even if there are duplicate keys. If there are many records of the same key value, it does not matter in what order we consider the duplicate values to occur.

LEMMA 2.1.3. *On any disk, the blocks are vacated from the input runs and moved (in possibly altered form) to the output run in the order of their maximum records.*

PROOF. At each step, the block with the smallest maximum is written to the output, although it may contain records that have been merged into that block from other blocks. Furthermore, no step modifies the maximum record of any block that is not written to the output. □

COROLLARY 2.1.4. *The largest record written to the output at step t on disk i is no larger than* $y_{i,t}^{\max}$.

PROOF. This corollary follows from Lemma 2.1.3 and the fact that at least B records no larger than $y_{i,t}^{\max}$ are read at the tth step. □

Now that we have limited how soon $y_{i,t}^{\max}$ can be written to the output, we turn our attention to $y_{i,t}^{\min}$.

LEMMA 2.1.5. *There are at most $t-1$ blocks on disk i that contain records smaller than $y_{i,t}^{\min}$.*

PROOF. In order to contain records smaller than $y_{i,t}^{\min}$, a block must have a smaller minimum than the block containing $y_{i,t}^{\min}$. But by definition, there are at most $t-1$ such blocks. (There could be more than $t-1$ blocks containing records at least as small as $y_{i,t}^{\min}$). □

LEMMA 2.1.6. *As long as there remain blocks on disk i that contain records smaller than $y = y_{i,t}^{\min}$, the number of such blocks decreases by at least one at each step.*

PROOF. Let us assume that there are still records less than y and look at any time step. There are two cases to consider: the algorithm reads one block on disk i or the algorithm reads two blocks. In the former case, the block contains records less than y, since it has a smaller minimum, and all its records are written to the output. In the latter case, the only way that the number of blocks containing records less than y could not decrease is if only one of the two blocks contains records less than y and some of those records are among those written back to the input. The block that was chosen as having the smallest minimum has records less than y. If there are at most B records of value less than y in the two blocks, then all these records will be output. Otherwise, if there are more than B records of value less than y from the two blocks, then each block contains at least one such record, and only one such block remains after output. □

Combining Lemmas 2.1.5 and 2.1.6, we get the following corollary.

COROLLARY 2.1.7. *All records smaller than $y_{i,t}^{\min}$ in the input runs on disk i are written to the output on disk i before the tth output block.*

Now that we have limits on when the maxima and minima are written onto the output for disk i, we are ready to consider pairs of disks. First, we look at disks after disk i.

LEMMA 2.1.8. *Let $1 \le i < j \le D$. Then $y_{i,t}^{\max} \le y_{j,t}^{\min}$ and $y_{D,t}^{\max} \le y_{1,t+1}^{\min}$.*

PROOF. Consider the t smallest minimum records on disk j. For each of these records, $y_{j,k}^{\min}$, $1 \le k \le t$, the block on the same track of the same run on disk i has a maximum no greater than $y_{j,k}^{\min}$. Since each of these maximum records occurs in a distinct block on disk i, and $y_{j,k}^{\min} \le y_{j,t}^{\min}$ for all $1 \le k \le t$, there are at least t maximum records on disk i that are no greater than $y_{j,t}^{\min}$. □

LEMMA 2.1.9. *Let y be the largest record written to the output on disk i at step t. Then all the records less than y on any disk $j > i$ are written to the output before step t.*

PROOF. By Corollary 2.1.7, any record not written to the output before step t on disk j is at least as large as $y_{j,t}^{\min}$. By Lemma 2.1.8, $y_{j,t}^{\min} \ge y_{i,t}^{\max}$. By Corollary 2.1.4, $y_{i,t}^{\max} \ge y$. Thus, all elements less than y have already been written on disk j before step t. □

Finally, we have gotten to the point that we can prove our first main lemma. However, before doing so, we need a definition.

Definition 2.1.10. A sequence is called *L-regressive* if for any two records $x < y$, y does not precede x by more than L records in the sequence.

LEMMA 2.1.11. *The approximately merged output list is RDB-regressive.*

PROOF. When a record y is written on disk i at step t, we do not have to worry about records less than y appearing later in the approximately merged list on any disks $j > i$, by Lemma 2.1.9. So we only need to consider those disks $j \leq i$. But, in a sense, we can consider disks $j \leq i$ as coming after disk i with a one block offset in each run. Thus, if $j \leq i$, then block b of run r on disk i precedes block $b + 1$ of run r on disk j. Consequently, the same argument that was used to show Lemma 2.1.9 applies in this case. In other words, if we removed the first block of each run on all disks $j \leq i$, no record on disk j would follow a larger record on disk i. The first blocks of those runs can push smaller records than y to later blocks of the output on disk j, but the position of such smaller records can only shift by a number of blocks equal to the number of runs, that is, by R blocks. Since each striped block in the output contains DB records, it follows that a larger record can precede a smaller one in the approximately merged list by at most RDB records. □

Now we show what RDB-regressive means in practical terms.

LEMMA 2.1.12. *If a list is L-regressive, then every record is at most L locations from its correct sorted location.*

PROOF. For the moment, let us suppose that all the records have unique keys. We will lift this assumption later. Let y be the jth smallest record and assume by way of contradiction that it occurs at position i where $i < j - L$. We derive a contradiction by showing that there exists an $x < y$ that succeeds y by more than L records and hence the list is not L-regressive. Since y is the jth smallest record, there are $j - 1$ records less than y. In order to meet the L-regressive condition, all the elements less than y must occur in the range of locations $1, \ldots, i + L$. There are $i + L$ locations up to the point that is L records beyond y, of which y is occupying one. So we have a total of j records to be fit into $i + L$ slots. By assumption, $i + L < j$, meaning that at least one record less than y must be out of the desired range, contradicting the fact that the list was assumed to be L-regressive. This same argument also shows that the jth record cannot be at any location $i > j + L$. When duplicate keys are allowed, the argument gets a little more complicated, since we cannot talk about the jth record exactly. Saying that every record is at most L locations from its correct sorted location means that there exists a permutation that sorts the list and also moves no element by more than L locations. The above argument shows that every record is within L of the closest location it can acceptably go, from which it follows that such a permutation exists. □

Lemmas 2.1.11 and 2.1.12 directly prove Theorem 2.1.2. Finally, we demonstrate that the final pass of Columnsorts finishes the sort.

THEOREM 2.1.13. *If every element in a list is within a distance of L of its sorted location, then a series of sorts of size* $2L$, *beginning at every Lth location, will suffice to complete the sort.*

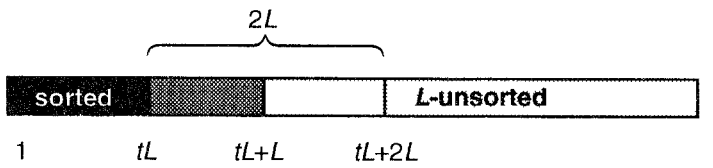

FIG. 8. A series of sorts of size $2L$ suffices to fix up an L-regressive list.

PROOF. Let there be N total records, so that we perform a total of $\lceil N/L \rceil - 1$ sorts of size $2L$ (the last sort may be smaller than size $2L$). The proof proceeds by induction on the number of sorts performed. During step $t \geq 0$ we sort records $tL + 1, tL + 2, \ldots, tL + 2L$. We have the following invariants at the beginning of step t (see Figure 8) and show that each step maintains the invariants:

(1) All the records $1, \ldots, tL$ are completely sorted.
(2) No record in $tL + 1, \ldots, N$ is more than $2L$ before or L after its final position.

The invariants are clearly met at the beginning since (1) refers to the null set and we have assumed something stronger than (2), namely that everything is within L of its correct sorted location.

Now we can demonstrate that each step in the range $t = 0, \ldots, \lceil N/L \rceil - 2$ preserves the invariants. To preserve Invariant (1), we need to demonstrate that the sort moves records that belong in locations $tL + 1, \ldots, tL + L$ to their final locations. By Invariant (2), every record that belongs in this range must fall within the $2L$ records we are sorting: the record belonging at $tL + L$ can be off by at most L, placing it at $tL + 2L$, which is within the sorting range. Thus, Invariant (1) is preserved. Similarly, Invariant (2) is preserved, since none of the records moved to the range $tL + L + 1, \ldots, tL + 2L$ belonged in the block $tL + 1, \ldots, tL + L$ (by preservation of Invariant (1)), so that they must be at most $2L$ before or L after their intended location.

The last step by definition sorts the last up to $2L$ records, so that at its conclusion, the whole list is sorted. □

2.2. ANALYSIS OF THE ALGORITHM. Now that we have shown the correctness of Greed Sort, we substantiate our claim that its performance is optimal. Before proving Theorem 2.1, we first show by a clustering technique that the amount of storage space required for the data structures is small enough.

THEOREM 2.2.1. *The amount of primary memory space needed for the data structures of Greed Sort is $O(M^{\beta})$, for fixed $\beta < 1$.*

PROOF. The number of runs that we must merge in order to obtain optimal performance is $\sqrt{M/B}/2$. As we pointed out in Section 2, the candidate set requires a total of $D\sqrt{M/B}$ key fields to be kept in primary memory to decide what blocks to read next, plus, in practice, any auxiliary structures used for implementing a priority queue. At first glance, it seems if $D = O(M)$ and $B = 1$ that $\Omega(M^{3/2})$ storage space will be required in primary memory, which

is clearly impossible. However, we can use a partial disk striping method throughout the course of the algorithm, while giving up only a constant factor in performance. Assume that $D = D(M)$ grows faster than M^{α}, for some fixed $0 < \alpha < 1/2$. We can cluster our D disks into clusters of $D' = M^{\alpha}$ clusters of D/D' disks synchronized together. Each of the D' clusters acts like a logical disk with block size $B' = BD/D'$. Thus, the number of primary storage locations we need is at most

$$D'\sqrt{M/B'} \leq M^{\alpha}\sqrt{M/B'} = O(M^{\alpha+1/2}).$$

The expression for the number of I/Os remains the same, namely,

$$k\frac{N}{D'B'}\frac{\log N/B'}{\log M/B'} = O\left(\frac{N}{DB}\frac{\log N/B}{\log M/B}\right).$$

We set $\beta = \alpha + 1/2$. The amount of memory needed for the buffers is $2DB$, so the total memory needed is at most

$$2DB + M^{\beta} \leq M,$$

since $DB \leq \lfloor (M - M^{\beta})/2 \rfloor$. Note that for efficiency, an additional $O(\log M)$ memory can be set aside for keeping priority queues without running into memory problems. □

In order to demonstrate that Greed Sort has an optimal I/O bound, we need to analyze the I/O efficiency of the Columnsort subroutine.

THEOREM 2.2.2. *Columnsort sorts $N \leq D\sqrt{MB}$ records with $O(N/DB)$ parallel I/Os.*

PROOF. First we show that Columnsort produces a correctly sorted sequence when $N \leq D\sqrt{MB}$. We define the number of rows in the matrix to be $r = M$, so that each column can be sorted internally. We have

$$N \leq D\sqrt{MB} \leq DB\sqrt{M} \leq \frac{M^{3/2}}{2} < \frac{M^{3/2}}{\sqrt{2}},$$

making use of our assumption that $2DB \leq M$. Thus, the number of columns in our application of Columnsort is $c = N/r \leq \sqrt{M/2}$, and the Columnsort requirement that $r \geq 2(c-1)^2$ is met. Thus, the Columnsort works correctly.

Steps 1, 3, 5, 6, 7, and 8 can be done easily with $O(N/DB)$ I/Os. The transpose-like operation in Steps 2 and 4 can be done with $O(N/DB)$ I/Os by the $p \times q$ matrix transpose algorithm of Vitter and Shriver [1990] for $p = M$ and $q = N/M \leq D\sqrt{B/M}$. Doing this transpose will put the records on the disks in exactly the right order; the only difference between the actual transpose and the transpose-like operation in Steps 2 and 4 is how large we consider the columns to be. The resulting number of I/Os for Steps 2 and 4 is

$$O\left(\frac{N}{DB}\frac{\log\min\left\{M, D\sqrt{B/M}, D\sqrt{MB}/B\right\}}{\log(M/B)}\right) = O\left(\frac{N}{DB}\frac{\log\left(D\sqrt{M/B}\right)}{\log(M/B)}\right)$$
$$= O\left(\frac{N}{DB}\right).$$

Since each of the eight steps can be done in $O(N/DB)$ I/Os, the overall Columnsort algorithm takes $O(N/DB)$ I/Os. □

The greedy merge reads each record at most three times (once for updating *biggest* and *smallest* and up to twice for merging) and writes each record at most twice, taking full advantage of parallel block transfer. The Columnsort routine is called $2\lceil N/k \rceil - 1$ times, each time using $O(k/DB)$ I/Os, for a value of k that differs from pass to pass. Thus, we have shown the following lemma.

LEMMA 2.2.3. *Each merging step requires $O(N/DB)$ parallel I/Os.*

By the remarks immediately before Theorem 2.1, this concludes the proof of Theorem 2.1.

3. *Conclusions*

We have presented the first optimal, deterministic external sorting algorithm for multiple disks, improving significantly the randomized algorithm of Vitter and Shriver [1990]. The Greed Sort algorithm is easy to implement and is efficient in terms of internal computations. An interesting open problem is whether a distribution-type sort can be implemented deterministically and optimally in terms of the number of parallel I/Os in the two-level model. Such an algorithm could have applications to optimal deterministic sorting in parallel versions of other memory hierarchies as well.

Recently, we have successfully developed a simple, deterministic distribution sort which, unlike Greed Sort, also extends to the parallel hierarchical memory models *P-HMM*, *P-BT*, and *P-UMH* [Nodine and Vitter 1993]. The algorithm is also optimal with respect to internal computation, even with parallel processors. The algorithm in Vitter and Shriver [1990] also has this property, but is randomized.

REFERENCES

AGGARWAL, A., AND VITTER, J. S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (Sept.), pp. 1116–1127.

AJTAI, M., KOMLOS, J., AND SZEMERÉDI, E. 1983. An $O(n \log n)$ sorting network. In *Proceedings of 15th Annual ACM Symposium on the Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, pp. 1–9.

BITTON, D., DEWITT, D. J., HSIAO, D. K., AND MENON, J. 1984. A taxonomy of parallel sorting. *Comput. Surv. 16*, 3 (Sept.), 287–318.

EVEN, S. 1974. Parallelism in tape-sorting. *Commun. ACM 17*, 4 (Apr.), 202–204.

FLOYD, R W 1972. Permuting information in idealized two-level storage. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, eds , Plenum, New York, pp. 105–109.

GIBSON, G., HELLERSTEIN, L., KARP, R. M., KATZ, R. H., AND PATTERSON, D. A. 1988. Coding techniques for handling failures in large disk arrays. UCB/CSD 88/477, (Dec.), Univ. Calif. at Berkeley, Berkeley, Calif

GIFFORD, D., AND SPECTOR, A. 1984. The TWA reservation system. *Commun. ACM 27*, 7 (July), 650–665.

HONG, J.-W., AND KUNG, H. T. 1981. I/O complexity: The red–blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing* (Milwaukee, Wis., May 11–13). ACM, New York, pp. 326–333.

JILKE, W. 1986. Disk array mass storage systems: The new opportunity. Tech. Rep. Amperif Corporation.

KNUTH, D. E. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., pp. 159–180.

LEIGHTON, T. 1985. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput. C-34*, 4 (Apr.), 344–354.

LINDSTROM, E. E., AND VITTER, J. S. 1985. The design and analysis of bucketsort for bubble memory secondary storage. *IEEE Trans. Computers C-34*, 3 (Mar.), 218–233.

MAGINNIS, N. B. 1987. Store more, spend less: Mid-range options around. *Computerworld* (November 16, 1987), 71–82.

NODINE, M. H., AND VITTER, J. S. 1993. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures* (Velen, Germany, June 30–July 2). ACM, New York, pp. 120–129.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Chicago, Ill., June 1–3). ACM, New York, pp. 109–116.

UNIVERSITY OF CALIFORNIA AT BERKELEY. 1989. Massive information storage, management, and use (NSF Institutional Infrastructure Proposal). Tech. Rep. *UCB/CSD* 89/493, Jan.

VITTER, J. S., AND SHRIVER, E. A. M. 1990. Optimal disk I/O with parallel block transfer. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (Baltimore, Md., May 12–14). ACM, New York, pp. 159–169.

RECEIVED JULY 1991; REVISED FEBRUARY 1994; ACCEPTED MARCH 1995