Optimal Deterministic Sorting on Parallel Disks

Mark H. Nodine and Jeffrey Scott Vitter

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-08
August 1992

Optimal Deterministic Sorting on Parallel Disks

Mark H. Nodine and *Jeffrey Scott Vitter*

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

August 31, 1992

Abstract

We present an optimal deterministic algorithm called Balance Sort 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 of Vitter and Shriver as well as the (non-optimal) commonly-used technique of disk striping. It also improves upon our earlier merge-based sorting algorithm in that it has smaller constants hidden in the big oh notation, it is possible to implement using only striped writes (but independent reads), and it has application to parallel memory hierarchies.

1 Introduction

Input/Output communication (I/O) between primary and secondary memory is a major bottleneck in many important computations, and is especially prevalent when parallel processors are used. Of particular importance is external sorting, in which the records to be sorted are too numerous to fit in internal memory and instead are kept in secondary storage, typically made up of one or more magnetic disks. Data are usually 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 disk read/write head is positioned. An increasingly popular way to get further speedup is to use many disk drives working in parallel [GHK, GiS, Jil, Mag, PGK, Uni].

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

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

where $M < N$, and $1 \leq DB \leq M/2$. In each I/O, D blocks of B records can be transferred simultaneously, as illustrated in Figure 1. This model generalized the initial work on I/O of Floyd [Flo] and Hong and Kung [HoK]. Aggarwal and Vitter proved that the average-case and worst-case number of I/Os required for sorting is[1]

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

Their lower bound is based solely on routing arguments, except for the pathological case in which M and B are extremely small, in which case the comparison model is

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

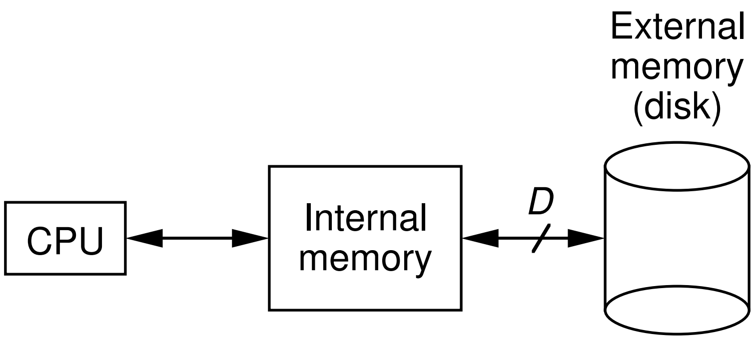

Figure 1: A simple D-parallel two-level memory model.

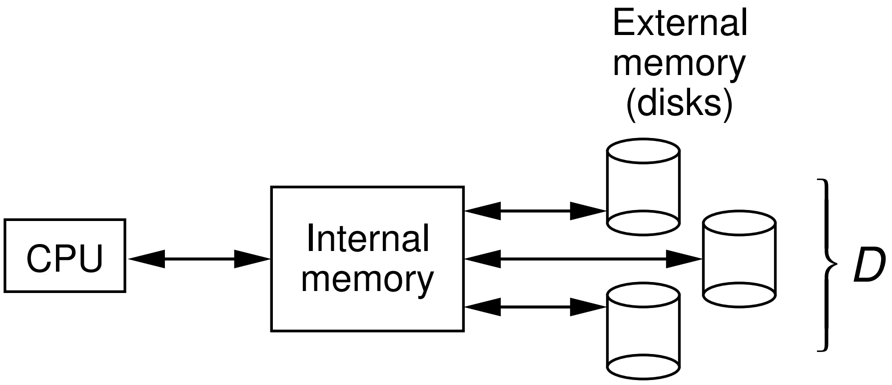

Figure 2: Model of parallel disks

used. They gave two algorithms, a modified merge sort and a distribution sort, that each achieved the optimal I/O bounds.

Vitter and Shriver considered a more realistic *D-disk model*, in which the secondary storage is partitioned into D physically distinct disk drives [ViSa], as in Figure 2. (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 D disks can simultaneously transfer one block of B records. Thus, D 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.

The measure of performance is the number of parallel I/Os required; internal computation time is ignored. In practice, though, the algorithms dealt with are very efficient in terms of internal processing. This model also applies to the case 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 CPUs 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 can be expected to be the I/O.

Disk striping is a commonly-used technique in which the D disks are synchronized, so that the D 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 $B' = DB$. 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 distribution sort, the algorithm partitions the set to be sorted into S sets of approximately equal size called *buckets*. Each bucket will consist of consecutive records in the total sorted list. The algorithm makes a pass over the data, separating the records into their respective buckets, and then sorts the buckets recursively. Finally, the buckets are appended to form a totally sorted list. Since at any level in the recursion the sets have approximately equal size, a total of $O(\log_S N)$ passes are required, each of which accesses $O(N)$ records to achieve a running time of $O(N \log_S N)$. The difficulty in implementing a version of distribution sort that works on a set of D parallel disks is making sure that each of the buckets can be read efficiently in parallel.

Vitter and Shriver presented a randomized version of distribution sort using two complementary partitioning techniques. Their algorithm meets the I/O lower bound (1) for the more lenient model of [AgV], and thus the algorithm is optimal. The randomization was used to distribute each of the buckets evenly over the D disks so that they could be read efficiently with parallel read operations. They posed as an open problem the question of whether there is an optimal algorithm that is deterministic. That question was answered in the affirmative by Nodine and Vitter using an algorithm called Greed Sort [NoVb]. That algorithm was based on merge sort, and was therefore not applicable to finding an optimal deterministic algorithm for parallel memory hierarchies, since there is no known way of using merge sort optimally on even a single memory hierarchy.

This report describes Balance Sort, the first known optimal and deterministic sorting algorithm based on distribution sort. Balance Sort is optimal for parallel disk sorting, *both in terms of the number of I/O steps and in terms of the amount of internal processing*. It provides a more practical and yet deterministic alternative to the optimal randomized algorithm of Vitter and Shriver [ViSa]. Previously there was one known optimal deterministic method for parallel disk sorting, called Greed Sort [NoVb], but Balance Sort has several important advantages over Greed Sort:

1. Balance Sort can be used as the basis of optimal deterministic algorithms for all defined parallel memory hierarchies, as described in the companion paper [NoVa]. By contrast, Greed Sort is based on merge sort, and in almost all cases there are no known optimal sorting methods for hierarchical memory based on merge sort, even when there is only a single hierarchy.

2. Balance Sort has smaller constants embedded in the big-oh notation.

3. Balance Sort can be implemented using only striped writes. This restriction is important in practical parallel disk systems, where striped writes may be required for maintaining redundancy information to make recovery possible in the event of a disk failure [CGK, PGK, Sch].

Section 2 describes how to balance the buckets in $O(N/DB)$ parallel I/Os in such a way that each bucket can be read efficiently in parallel at a later stage in the recursion The balancing subroutine is the heart of the overall sorting algorithm. Section 3 describes how to apply the balancing algorithm in such a way as to achieve optimal parallel sorting time. Section 4 explains the significance of this algorithm.

2 How to Balance

Distribution Sort is an algorithm that chooses a set of numbers that partition the elements to be sorted into ranges of values called *buckets*. Having the partition elements, we can then process the elements one at a time, placing each into the appropriate bucket, and sort each bucket. Finally, we concatenate the buckets in the right order to produce a completely sorted list. For example, if you needed to sort the checks from your bank statement, and the checks fall in the range 350–399, you might choose as your partition elements 360, 370, 380, and 390. You could then sort the checks into stacks (buckets). Once each stack is sorted, you then put the 350s first, followed by the 360s, etc., and the checks are fully sorted.

This report describes a new algorithm called Balance Sort that is based on Distribution Sort. In Balance Sort, after finding the partitioning elements, we read D records into memory in parallel, partition them, and write them back in parallel, then read, partition, and write the next D records, and so on, until the whole file has been partitioned into buckets. We want to minimize the number of parallel reads that we will need during the next level of recursion to re-read each bucket. The heart of Balance Sort is in balancing all of the buckets evenly among the disks. This section describes the Balance routine, while the next section describes how to use Balance to achieve the optimal number of I/Os. For the time being, we will assume that the block size B is 1. Nothing in this section is dependent upon that assumption, and we show in Section 3.4 how to accommodate arbitrary block sizes.

Let x_{bd} be the number of records written to disk d from bucket b; $X = (x_{bd})$ is called the *histogram matrix*. The number of parallel reads that are needed to read bucket b during the next level of recursion is then $r_b = \max_{\mathbf{disks}\, d}\{x_{bd}\}$. So we want to minimize $\sum_{\mathbf{buckets}\, b} r_b$. We process the file a track at a time (all of the disks are synchronized for reading) and use a greedy strategy that attempts to minimize the increase of this sum. We do this by computing a matrix Y, called the *skew matrix*, that gives a measure of how unbalanced each bucket is. We define $y_{bd} = x_{bd} - x_b^{\min}$, where $x_b^{\min} = \min_{\mathbf{disks}\, d}\{x_{bd}\}$. Intuitively, the skew matrix element y_{bd} indicates how many records of bucket b will be on disk d after all the records in the bucket have been read on some other disk. We call y_{bd} the *skew* of bucket b on disk d. We attempt to minimize the elements of the skew matrix by computing a min-cost bipartite matching between blocks and disks. In particular, a block from bucket b will have cost y_{bd} to be assigned to disk d. Consider Algorithm 1.

In this algorithm, we assume that bucket(a, P) returns the number of the bucket to which a belongs based on the set of partitioning elements P. Note that this algorithm is an on-line algorithm in that it makes its decisions on where to place things based only on the records it has already seen (as codified in the arrays X and Y). The complexity of finding a permutation for the min-cost matching is $O(D^3)$, but for many architectures, this computation will be able to be done in parallel with I/O operations, and so will not result in any slowdown.

Algorithm 1 [Simple_Balance(P)]

```
{ P contains the partition elements }
{ Initialize }
for b := 1 to S
    for d := 1 to D
        x_bd := 0
        y_bd := 0

{ Process the file }
for t := 1 to ⌈N/D⌉
    read D records in parallel into array T
    { Compute the matching matrix Z }
    for i := 1 to D
        τ[i] := bucket(T[i], P)
        for d := 1 to D
            z_{i,d} := y_{τ[i],d}
    find a permutation π of T that is a min-cost matching with matching matrix Z
    for d := 1 to D
        increment x_{τ[π(d)],d}
        increment y_{τ[π(d)],d}
    decrement any row of Y that has no zeros
    write out T in the permuted order
```

There is an obvious fact about the disk sums in the histogram matrix.

Lemma 1 *The disk sums in the histogram matrix, $x_d = \sum_b x_{bd}$, are all equal.*

Proof: By definition, $\sum_b x_{bd}$ is the number of tracks written on disk d. Since we are reading and writing whole tracks at a time, the sum must be constant over all disks. □

It is not necessary from an algorithmic standpoint to have a skew matrix Y, since the min-cost matching will give exactly the same results if it consists of rows from the histogram matrix X. However, the skew matrix gives some insight into how unbalanced the buckets are. Intuitively, the skew matrix tells what will be left over after all the buckets have been read with all the full parallelism available to them. Assigning a bucket b to a disk d for which y_{bd} is zero is a good thing; it will never increase the number of reads required to process the entire bucket beyond the minimum needed. Alternatively, assigning a bucket b to a disk d for which y_{bd} is larger than $y_{bd'}$ for any other disk d' will always increase the total number of reads needed to process the bucket beyond what is strictly needed. In doing a min-cost matching, we are attempting to balance all of the buckets represented in the track simultaneously. The skew matrix has some useful properties, as evidenced by the following simple lemmas.

Lemma 2 *Every bucket b contains at least one zero in the skew matrix.*

Proof: By definition of the skew matrix, those disks d for which $x_{bd} = \min_d \{x_{bd}\}$ will have $y_{bd} = 0$. □

Lemma 3 $y_{bd} \geq 0$ *for all b and d.*

Proof: By definition of constructing the skew matrix, we subtracted off the minimum value for each bucket; hence all the other values in the bucket were at least as large as the one subtracted. □

Lemma 4 *The disk sums in the skew matrix, $y_d = \sum_b y_{bd}$, are all equal.*

Proof: This fact is a simple consequence of Lemma 1 and the fact that forming the skew matrix from the histogram matrix subtracts the same number from each disk for each bucket. □

We need a little information about how the skew matrix evolves as a function of the number of tracks processed.

Definition 1 Let Y be a skew matrix and let Y' be the same skew matrix after a track has been processed using the min-cost matching to assign buckets to disks and after any rows that have no zeroes have been decremented. If $y'_{bd} > y_{bd}$ then we say that bucket b has been *promoted* on disk d.

Lemma 5 *At most one bucket promotes on any given disk on any track. Furthermore, that bucket can increase its skew by at most one on the disk.*

Proof: Since exactly one bucket has its value in the histogram matrix incremented on the given disk, it is the only one that could have its value increase in the skew matrix on that disk. Since the value in the histogram matrix was incremented by one, one is also the maximum possible increase in its skew. □

The following lemma about changes to the skew matrix is pivotal in proving the optimal behavior of our algorithm.

Lemma 6 *Assume that the largest element of skew matrix Y is* $\max$ *and that some min-cost matching for a matching matrix composed of rows of Y results in a skew matrix Y' for which several buckets have a skew of* $\max+1$*. Then all the buckets that reached the larger skew must have had a skew of* $\max$ *in Y on every disk for which the skew increased to* $\max+1$*.*

Proof: Assume that bucket b reaches a skew of $\max + 1$ on disk d and bucket b' reaches a skew of $\max + 1$ on disk d'. Then in the partial matching, we have

$$\begin{array}{c|cc} & d & d' \\ \hline b & \underline{\max} & i \\ b' & j & \underline{\max} \end{array}$$

where underlines indicate that the elements are part of a min-cost matching. Hence, it must be the case that $i + j \geq 2 \max$. Since no element of Y is greater than max, then it must be the case that $i = j = \max$. Since this fact is true for every pair of buckets that achieve a new maximum skew, the result follows. $\square$

The upshot of Lemma 6 is that if several buckets are promoted to a new maximum skew value, there must have been a "block of maxes" in the matching matrix. That is, the matching matrix must in part (up to permutations of the rows and columns) have looked like this:

$$\begin{array}{cccc} \underline{\max} & \max & \ldots & \max \\ \max & \underline{\max} & \ldots & \max \\ \vdots & \vdots & \ddots & \vdots \\ \max & \max & \ldots & \underline{\max} \end{array}$$

The difficulty with Algorithm 1 is that, although there is much empirical evidence to suggest that it is always within a factor of 2 of the optimal time for reading all the buckets back in, it has been difficult to prove this conjecture in the general cases. Intuitively, the algorithm always tries to place the buckets on the disk that conflicts least with what has been previously placed. If it could be shown that the maximum skew grows sufficiently slowly (for example, that it takes at least a quadratic number of tracks to attain any given skew value) or that the maximum skew is bounded linearly by the number of buckets available, then it would be possible to show that Algorithm 1 could obtain optimal results for balancing.

Our alternative and provably optimal algorithm makes use of two key ideas:

1. We rebalance occasionally if the min-cost operation skews the balance too badly.

2. We only need to use some (preferably large) constant fraction of the disks efficiently.

The point behind rebalancing is that when some bucket b gets too skewed, that is, when $x_{bd} - x_{bd'} > c$ for some disks d and d', where $c > 0$ is some skew threshold, we can read a block from bucket b on disk d and write it to disk d'. In fact, any number of buckets (up to $\lfloor D/2 \rfloor$) can be rebalanced in a single parallel I/O operation, as long as the swaps all take place on distinct disks.

Saying that we need to use only some constant fraction α of the disks efficiently for any given bucket means that the number of tracks needed to read that bucket will expand by a factor of only about $1/\alpha$ above the optimal. Our algorithm will use $\alpha = \frac{1}{2}$.

Algorithm 2 [Balance(P)]

{ P contains the partition elements }
{ Initialize }
for $b := 1$ to S
    for $d := 1$ to D
        $x_{bd} := 0$
        $\underline{a_{bd} := 0}$

{ Process the file }
for $t := 1$ to $\lceil N/D \rceil$
    read D records in parallel into array T
    { Compute the matching matrix Z }
    for $i := 1$ to D
        $\tau[i] :=$ bucket($T[i], P$)
        for $d := 1$ to D
(1) $\underline{z_{i,d} := a_{\tau[i],d}}$
    find a permutation π of T that is a min-cost matching with matching matrix Z
    write out T in the permuted order
    for $d := 1$ to D
        increment $x_{\tau[\pi(d)],d}$
(2) $\underline{A := \text{ComputeAux}(X)}$
(3)   **if** there is a 2 in A
(4)   Rebalance(A, X)
        $\underline{A := \text{ComputeAux}(X)}$

Our algorithm is given in Algorithm 2. In addition to keeping a histogram matrix X, the algorithm keeps an auxiliary matrix A to ensure that after processing each track, no bucket is too far out of balance. Specifically, if m_b is the median number of blocks that bucket b has on all the disks (m_b is the median of $x_{b1}, \ldots, x_{bD}$),[2] we define $a_{bd} := \max\{0, x_{bd} - m_b\}$. Invariant 1 about A below will guarantee that $X_{bd} \leq m_b + 1$ for all $1 \leq d \leq D$.

The differences between Algorithm 1 and Algorithm 2 are underlined. This algorithm requires two subroutines to complete its work: ComputeAux and Rebalance, given in Algorithms 3 and 4, respectively. We define these routines so as to maintain two invariants on the matrix A at step (1) in the algorithm:

1. A is a binary matrix; that is, all of its elements are either 0 or 1.

[2]We use the convention that the median is always the $\lceil D/2 \rceil$th smallest element, rather than the convention in statistics that it is the average of the two middle elements when D is even.

Algorithm 3 [ComputeAux(X) returns A]

> **for** $b := 1$ to S
> $m :=$ median of $x_{b1}, \ldots, x_{bD}$ (the $\lceil D/2 \rceil$th smallest element)
> **for** $d := 1$ to D
> $a_{bd} := \max(0, x_{bd} - m)$

Algorithm 4 [Rebalance(A, X)]

```
    { Create a bipartite matching problem }
    U := {d | a_bd = 2 for some b ∈ 1,...,S}
    V := {1,...,D} − U
    E := ∅
    for i := 1 to |U|
        d := U_i
(1)     b := (unique) bucket such that a_bd = 2
        for j := 1 to |V|
            d' := V_j
            if a_bd' = 0
                E := E ∪ (U_i, V_j)
    { We will swap a pair of blocks for every edge in the following match }
(2) Find a maximum bipartite matching on the graph G = (U ∪ V, E)
    { The array R will tell us what buckets to read from each disk }
    { The array W will tell us on what disk to write the block }
    for d := 1 to D
        r_d := 0
        w_d := 0
    for each match (U_i, V_j)
        d := U_i
        r_d := b
        w_d := V_j
        { Update X to reflect the swap }
        x_bd := x_bd − 1
        x_bd' := x_bd' + 1
    Read the last block of bucket r_d on disk d if r_d ≠ 0
    Write the block read from disk d onto disk w_d if r_d ≠ 0
```

2. Every row of A has at least $\lceil D/2 \rceil$ 0s in it (or equivalently, every row has no

more than $\lfloor D/2 \rfloor$ 1s in it).

It is easy to see that Invariant 2 is maintained, since the smallest $\lceil D/2 \rceil$ elements all become zero, by our definition of the median.

To show Invariant 1, it is necessary to understand a bit more about how the min-cost matching affects the auxiliary matrix and how effective the Rebalance subroutine is. We use induction to show that the variant is maintained. Auxiliary matrix A is clearly binary initially (it is all 0s).

To show that the invariant is maintained, we first need a preliminary lemma. Next we will show a couple of lemmas that assume the invariant holds at line (1) of Algorithm 2 and show what happens to A at later points in the iteration. After this, we show some lemmas needed to demonstrate that Rebalance is able to remove any 2s introduced as a result of the min-cost matching. Finally, Theorem 1 establishes the invariant.

Lemma 7 *The only entries of A that can be larger at line (3) of Algorithm 2 than they were at line (1) of the same iteration are those matched in the min-cost matching. Furthermore, any element of A that increases can increase by only one.*

Proof: Certainly, any elements of X not involved in the min-cost matching are the same at line (3) as they were at line (1), so the crux of the proof is to show that the call to ComputeAux in line (2) maintains this property of A. But since the median for any bucket can only go up as a result of the matching, the difference between any element not matched and the median can only go down. The elements of A involved in the match increase by only one, and again, since the median can only increase, the difference between them and the median can go up by at most one. □

Now we can show that lemmas similar to Lemmas 5 and 6 for skew matrices will hold for the auxiliary matrix.

Lemma 8 *Assuming that A is binary at line (1) of Algorithm 2, at most one bucket will have a 2 on any given disk at line (3) of the algorithm during the same iteration. No buckets will have any elements larger than 2.*

Proof: By Lemma 7, only those elements involved in a match could have increased and even then, only by one. Since A was binary prior to the match, the largest element is no greater than 2 at line (3). Furthermore, by definition of the match, exactly one element of X is incremented on each disk; that element is the only one that could become a 2 on that disk. □

Lemma 9 *Assume that A is binary at line (1) of Algorithm 2 and let $\mathcal{D}$ be the set of disks having a 2 in the auxiliary matrix at line (3) during some iteration. Then every bucket that has a 2 in must have had a 1 on every disk in $\mathcal{D}$ back in line (1) of the algorithm during this iteration.*

Proof: The argument used in the proof of Lemma 6 works here as well, except that in this case, max = 1 and some of the 2s may have reverted to 1s during the call to ComputeAux at line (2) of the algorithm, if the median for that bucket happened to increase. However, the "block of 1s" is still a necessary (though not sufficient) condition for getting 2s on several buckets. □

Every bucket in A at line (3) of Algorithm 2 has at least $\lceil D/2 \rceil$ zeroes in it. The call to ComputeAux at line (2) guarantees this fact. We need one more fact about A before we can understand the Rebalance routine.

Lemma 10 *Assuming that A is binary at line (1) of Algorithm 2, at most $\lfloor D/2 \rfloor$ disks will have 2s in A at line (3) on the same iteration.*

Proof: By Lemma 9, a necessary condition for the introduction of 2s on k different disks is to have a "block of 1's" in the matching matrix. But each bucket has at least $\lceil D/2 \rceil$ zeroes, which means that the maximum width of the block of 1s is $\lfloor D/2 \rfloor$. □

Now that we know what the matrix A looks like at the call to Rebalance at line (4) of Algorithm 2, we need to show some facts about the Rebalance subroutine. First, note that Lemma 8 establishes that the bucket at line (1) of Algorithm 4 is unique. Also, note that the following lemmas are assuming that the induction hypothesis holds (i.e., that A was binary back on line (1) of Algorithm 2), even though we do not state that assumption as part of the lemmas.

Lemma 11 *The maximum matching at line (2) of Algorithm 4 will include all vertices in U.*

Proof: By Lemma 10, $|U| \leq \lfloor D/2 \rfloor$. Also, each vertex in U has at least $\lceil D/2 \rceil$ edges coming out of it. Each edge that is chosen for the match can invalidate at most one possible edge for every other vertex of U. Since the minimum number of edges coming out of any vertex in U exceeds $|U|$, then the simple greedy algorithm of just choosing the first edge for the first vertex of U, the first remaining edge for the second vertex of U, and so on, will always result in a match that includes all the vertices of U. □

At long last, we are ready to prove that A is always binary at line (1) of Algorithm 2.

Theorem 1 *Invariant 1 (A is a binary matrix) is maintained at line (1) of Algorithm 2.*

Proof: The proof proceeds by induction on the number of tracks processed. Before any tracks have been processed, it is trivially true, since all the elements of A are zero.

Assume that the invariant holds at the beginning of an iteration. We now show that the invariant will hold at the end of the iteration to be carried back to the beginning of the next iteration. If there is no 2 in A at line (3) of Algorithm 2, then the invariant is clearly maintained. Assume that $a_{bd} = 2$ at line (3) of Algorithm 2. We know from Lemma 11 that the vertex corresponding to disk d in U will be matched to some disk d' for which $a_{bd'} = 0$. The rebalancing algorithm will read the last block of bucket b on disk d and write it to disk d'. This operation is guaranteed to make it so that the call to ComputeAux in line (5) of Algorithm 2 will result in $a_{bd} \le 1$ and $a_{bd'} \le 1$, since the median element could not have existed on disk d, so there is no chance of lowering the median. By Lemma 11, we can accommodate all of the 2s that appeared in A simultaneously, and since the matching problem had only one vertex per disk, they can all be balanced in the single parallel read/write operation. Thus, A is binary at the end of the loop and does not change before line (1) at the next iteration (or at the end of the algorithm, if this was the last iteration). □

We now have a theorem that tells how well each bucket is balanced.

Theorem 2 *Any bucket b will take no more than a factor of 3 above the optimal number of tracks to read.*

Proof: Let m_b denote the median element in bucket b. Then, by the Invariant 1, which is also maintained at termination of the algorithm, the number of tracks needed to read bucket b is no more than $m_b + 1$. But we also know that the bucket has at least $\lceil D/2 \rceil m_b$ elements in it, so it requires at least

$$\left\lceil \frac{\lceil D/2 \rceil m_b}{D} \right\rceil \ge \left\lceil \frac{m_b}{2} \right\rceil$$

tracks to read it. Thus, we are a factor of at most

$$\frac{m_b + 1}{\lceil m_b/2 \rceil} \le 3$$

from the optimal. (The expression could be as large as 3 if D is even and $m_b = 2$.) □

It should be noticed that the Algorithm 2 assumes that it reads D blocks on every track, whereas the previous phase does not guarantee that every track up to the last is fully utilized. Adjusting the algorithm for partial tracks is simple: if no block is available to be read on some disk d, we pretend that we read from a dummy bucket 0. Bucket 0 will be have the characteristic that $x_{0d} = 0$ for all $1 \le d \le D$, so in particular, that row of the matching matrix will be all zeroes. The part of the algorithm that increments the values of X should ignore bucket 0.

Finally, we can bound the total number of parallel I/Os needed for balancing N records into S buckets.

Theorem 3 *A phase of Balance requires no more than $6N/DB$ parallel reads and a like number of parallel writes.*

Algorithm 5 [Balance_Sort(N, T)]

```
    if N ≤ M
(1)     Read the whole bucket into memory
        Sort internally
(2)     Write the bucket out again
    else
        S := min(2√(M/B), 2N/M)
(3)     P := ComputePartitionElements(S)
        { The T array says what track to start with on each disk }
(4)     L := Balance(T)
(5)     Write L
        for b := 1 to S
(6)         T := Read bth row of L
            N_b := number of elements in bucket b
(7)         Balance_Sort(N_b, T)
(8)         Append sorted bucket to output area
```

Proof: We know from Theorem 2 that the N records we are processing occupy no more than $3N/DB$ tracks. Each of these tracks can give rise to at most 2 parallel reads and writes: one for the original assignment to disks according to the min-cost matching, and at most one parallel read and write in rebalancing. □

The other modification that we need to make to the balancing algorithm before it can be used as part of a sorting routine is that it needs some provision for keeping track of where each bucket is located so that the next pass will know how to read it in. This is easily accomplished by keeping another $D \times S$ matrix L such that l_{bd} is the last track with a block from bucket b on disk d. With a constant amount of overhead in each block, we can store a pointer to the previous block on the same disk (or zero if this is the first block). Thus, starting from the final configuration of L, we can read the bucket backwards.

3 The Sorting Algorithm

This section describes how to fit the balancing algorithm into an optimal algorithm for sorting, which we call Balance Sort. It assumes that the Balance routine has been modified to return the starting (ending) blocks of each bucket on each disk, as described in the previous section. Algorithm 5 gives the actual algorithm, assuming for the time being that $B = 1$. Subsection 3.4 describes how to handle the case where $B > 1$.

The correctness of the algorithm is easy to establish, since the bottom level of recursion by definition produces a sorted list and each level thereafter concatenates sorted lists in the right order. In this algorithm, we will let $S = \min(2\sqrt{M/B}, 2N/M)$ in order to produce optimal performance, as shown in Subsection 3.2.

Theorem 4 *Algorithm 5 correctly sorts N elements using*

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

parallel I/O's, as long as $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$.

The next few subsections establish the proof of this theorem.

3.1 Finding the partition elements

The following procedure for finding the partition elements was presented in [ViSb], and is reproduced here for convenience in Algorithm 7. The algorithm is deterministic and guarantees to find $S-1$ partition elements such that, for any bucket b, the number of elements in that bucket N_b obeys the constraint

$$\frac{N}{2S} \leq N_b \leq \frac{3N}{2S}.$$

The algorithm was analyzed in [ViSb] and shown to take $O(N/DB)$ parallel I/O operations which, as we will see in the analysis, is sufficient for achieving optimal performance.

The procedure for finding the partitioning elements uses as a subroutine Algorithm 6, which computes the kth smallest of n elements in $O(n/DB)$ I/Os. It does this by recursively subdividing about an element that is guaranteed to be close to the median.

3.2 Analysis of sorting algorithm

We have numbered each of the steps of Algorithm 5 that could cause an I/O operation. Let $T(N)$ denote the amount of time needed for sorting N records.

Steps (1) and (2) occur only at the innermost level of recursion. From Theorem 2, the bucket will take no more than $3N/DB$ reads and writes.

Step (3), as mentioned, has been shown by [ViSb] to require only $O(N/DB)$ I/Os, regardless of whether $S = 2\sqrt{M/B}$ or $S = 2N/M$. Step (4), as shown in Theorem 3, requires no more than $6N/DB$ I/Os. Step (5) requires writing a set of cardinality DS, which can be done in S/B parallel writes. Step (6) can be done in a single parallel read, since the cardinality of the set is only D, for a total of S reads. Step (7) takes

$$\sum_{1 \leq b \leq S} T(N_b),$$

Algorithm 6 [Select(S_0, k)]

> $t := \lceil |S_0|/M \rceil$
> **for** $i := 1$ to t
>   Read M elements in parallel ($\lceil M/DB \rceil$ tracks; the last may be partial)
>   Sort internally
> $R_i :=$ the median element
> **if** $t = 1$
>   return(kth element)
> $s :=$ the median of $R_1, \ldots, R_t$ using algorithm from [BFP]
> Partition into two sets S_1 and S_2 such that $S_1 < s$ and $S_2 \geq s$
> $k_1 := |S_1|$
> $k_2 := |S_2|$
> **if** $k \leq k_1$
>   return(Select(S_1, k))
> **else**
>   return(Select($S_2, k - k_1$))

Algorithm 7 [ComputePartitionElts(S) returns P]

> **for** each memoryload of records $\mathcal{M}_i (1 \leq i \leq N/M)$
>   Read $\mathcal{M}_i$ into memory
>   Sort internally
>   Construct $\mathcal{M}'_i$ to consist of every $(S-1)/4$th record
>   Write $\mathcal{M}'_i$
> $\mathcal{M}' := \mathcal{M}'_1 + \cdots + \mathcal{M}'_{N/M}$
> **for** $j := 1$ to $S - 1$
> $b_j :=$ Select($\mathcal{M}', 4Nj/(S-1)^2$)
> return($\{b_j\}$)

where N_b is the number of elements in bucket b. Let $N_b = \frac{N}{S} + \Delta_b$, where $|\Delta_b| \leq \frac{N}{2S}$ and $\sum_{1 \leq b \leq S} \Delta_b = 0$. Finally, step (8) can read the bucket in no more than $3N_b/DB$ I/Os and write it in no more than N_b/DB I/Os, for a total (over all buckets) of

$4N/DB$ I/Os. Thus, we obtain the recurrence

$$T(N) = \begin{cases} \sum_{1 \le b \le S} T\left(\frac{N}{S} + \Delta_b\right) + O\left(\frac{N}{DB}\right) + O(S) & \text{if } N > M \\ \frac{3N}{DB} & \text{if } N \le M \end{cases}$$

Furthermore, we can show that

$$S = \begin{cases} 2\sqrt{\frac{M}{B}} & \text{if } N \ge \frac{M^{3/2}}{\sqrt{B}} \\ \frac{2N}{M} & \text{if } N \le \frac{M^{3/2}}{\sqrt{B}} \end{cases}$$

First, consider the case $N \le M^{3/2}/\sqrt{B}$. This results in a recurrence of

$$T(N) = \sum_{1 \le b \le S} T\left(\frac{M}{2} + \Delta_b\right) + O\left(\frac{N}{DB}\right) + O\left(\frac{N}{M}\right),$$

where $|\Delta_b| \le M/4$. In particular, each bucket has size no more than $3M/4$, and so we can sort it internally. Thus,

$$T(N) = \frac{1}{DB} \sum_{1 \le b \le S} \left(\frac{M}{2} + \Delta_b\right) + O\left(\frac{N}{DB}\right) + O\left(\frac{N}{M}\right) = O\left(\frac{N}{DB}\right),$$

since $M > DB$.

When $N \ge M^{3/2}/\sqrt{B}$, it is straightforward, though tedious, to demonstrate by substitution and approximating $\log(1+\epsilon)$ by the first few terms of its infinite expansion that a solution to the recurrence is

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

All of this analysis leads to the following theorem.

Theorem 5 *The algorithm given is optimal for sorting in the parallel disk model.*

Proof: The preceding analysis shows that the algorithm achieves the same time bound as was shown in [AgV] to be a lower bound for sorting with parallel block transfer. □

3.3 Memory usage of the sorting algorithm

One aspect of the algorithm that we still need to show is that we do not exceed the memory requirements of the computer. The algorithm maintains three $S \times D$ arrays: L, A, and X.

Theorem 6 *For $0 < \beta < 1/2$, the amount of primary memory space needed for the data structures of the sorting algorithm is $O(M^{1/2+\beta})$.*

Proof: In either range of S, it is easy to show that the space required is no more than $2\sqrt{M/B}D$. At first glance, it seems if $D = \Omega(M)$ and $B = 1$ that we will require $\Omega(M^{3/2})$ storage space in primary memory, which is clearly impossible. However, we can use a partial disk striping method throughout the course of the algorithm, as described in [NoVb]. Assume that D grows faster than M^β, where $0 < \beta < 1/2$. We can cluster our D disks into clusters of $D' = M^\beta$ clusters of $B' = BD/D'$ disks synchronized together. Each of the D' clusters acts like a logical disk with block size B'. Thus, the number of primary storage locations we need is at most

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

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

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

□

3.4 Dealing with larger blocks

Although we gave the analysis for general block sizes, there has until now been no description of how the algorithm collects the records into blocks. The modifications to do this take place in the Balance routine (Algorithm 2). Algorithm 8 shows the modified Balance routine. The number of I/O's performed is not affected; the changes only affect the amount of memory needed.

At the point in the loop where D records are read in parallel into array T, we read whole blocks, and repack into blocks containing only records that belong in the same bucket. Whenever we have D full blocks, we go into the normal process of min-cost matching and rebalancing if necessary. At the end, we also use the min-cost and rebalancing method to output the partially full blocks.

There can be no more than S partially filled blocks, and consequently the amount of space needed before we find D completely filled blocks is never more than $(S+D)B$. We thus can show the following theorem.

Theorem 7 *The amount of space needed for buffering and collecting into buckets will never exceed M as long as $M \geq 4DB$.*

Proof: As shown above, the amount of space needed for collecting into blocks is no more than $(S + D)B$. In addition, we need space DB for the input buffer. Thus, the total amount of storage needed for buffering and collecting into blocks will never

Algorithm 8 [Balance_With_Blocks(P)]

```
{ Initialize }
for b := 1 to S
    for d := 1 to D
        x_{bd} := 0
        a_{bd} := 0

{ Process the file }
for t := 1 to ⌈N/DB⌉
    read D blocks in parallel into input array
    Apportion records into blocks containing all the same bucket
    if there are at least D full blocks or t = ⌈N/D⌉
        if there are at least D full blocks
            Define array T to comprise D full blocks
        else
            Define array T to be any D blocks that need writing
        { Compute the matching matrix Z }
        for i := 1 to D
            τ[i] := bucket(T[i])
            for d := 1 to D
                z_{i,d} := a_{τ[i],d}
        find a permutation π of T that is a min-cost matching with matching matrix Z
        write out T in the permuted order
        for d := 1 to D
            increment x_{τ[π(d)],d}
        A := ComputeAux(X)
        if there is a 2 in A
            Rebalance(A, X)
            A := ComputeAux(X)
```

exceed $SB + 2DB$. But $S \leq \sqrt{M/B}$ throughout the algorithm and by hypothesis, $M \geq 4DB$, so the amount of space needed is no more than

$$\begin{aligned} SB + 2DB &\leq \sqrt{MB} + 2DB \\ &\leq \sqrt{MDB} + 2DB \\ &\leq \sqrt{M\frac{M}{4}} + 2\frac{M}{4} \\ &= M \end{aligned}$$

□

4 Conclusions

In this report, we have described a new algorithm for sorting on parallel disk subsystems called Balance Sort. This algorithm has the following advantages over previously known deterministic algorithms:

1. The new algorithm is also applicable to parallel memory hierarchies, as described in the companion paper [NoVa]. The difficulty with Greed Sort is that it was based on merge sort, and in many cases of hierarchical memories, there is no known way to make merge sort work optimally on even a single memory hierarchy. This problem would have had to have been solved before Greed Sort could have had any applicability to these parallel memory hierarchies.

2. The hidden constants in the big-oh notation are small. The problem with the Greed Sort algorithm of [NoVb] is that it uses Columnsort [Lei] as a subroutine, which introduces at least an additional factor of 4 into the constant of proportionality.

3. The algorithm can operate using only striped write operations. Some parallel disk subsystems, such as Thinking Machine's Data Vault, impose this requirement so that they can maintain error checking and redundancy information to give acceptable levels of reliability when dealing with many disks, each of which has an independent probability of failure. It was unclear how to adapt the Greed Sort algorithm to use only striped writes, again because of the embedded call to Columnsort.

While Balance Sort is applicable to parallel memory hierarchies, there is still a significant hurdle to be overcome when the hierarchies have an effectively logarithmic cost function. In the models that we consider, there is a requirement that the internal processing be able to be done in $\log P$ parallel time in order to achieve optimal performance, where P CPUs are connected together, each with its own memory hierarchy. It should be noted that the min-cost matching in the Balance routine is operating on a 0-1 matrix, and that therefore the full power of min-cost matching is not required; indeed, maximum matching on the complement of the matching matrix will do the job. Furthermore, the proof that the "block of 1s" is present does not even depend upon maximum matching, but only that the matching is a local maximum. Likewise, it is not hard to show that any maximal matching in the Rebalance routine is also a maximum matching. Thus, we can use maximal matching in both the Balance and Rebalance routines. Unfortunately, the best known deterministic parallel time for maximal matching is $O(\log^3 P)$ [IsS], which is not good enough to get optimal performance on the parallel memory hierarchies with effective logarithmic cost functions. The adjustments necessary to obtain optimal performance are the subject of the companion paper [NoVa].

An interesting sorting algorithm in the context of fixed interconnection networks was proposed recently by Cypher and Plaxton [CyP]. The algorithm runs in $O(\log n (\log \log n)^2)$ time on an n-processor hypercube, shuffle-exchange, or cube-connected cycles network. We are currently exploring whether their techniques would be applicable to our I/O model, and *vice versa* whether our balancing techniques would yield good sorting algorithms for networks.

5 References

[AgV] A. Aggarwal and J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* (September 1988), 1116–1127.

[BFP] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *J. Computer and System Sciences* 7 (1973), 448–461.

[CGK] P. Chen, G. Gibson, R. H. Katz, D. A. Patterson, and M. Schulze, "Two Papers on RAIDs," U. C. Berkeley, UCB/CSD 88/479, December 1988.

[CyP] R. Cypher and C. G. Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers," *Journal of Computer and System Sciences* (to appear).

[Flo] R. W. Floyd, "Permuting Information in Idealized Two-Level Storage," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, ed., Plenum, 1972, 105–109.

[GHK] G. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson, "Coding Techniques for Handling Failures in Large Disk Arrays," U. C. Berkeley, UCB/CSD 88/477, December 1988.

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

[HoK] J. W. Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. of the 13th Annual ACM Symposium on the Theory of Computing* (May 1981), 326–333.

[IsS] A. Israeli and Y. Shiloach, "An Improved Parallel Algorithm for Maximal Matching," *Information Processing Letters* 22 (1986), 57–60.

[Jil] W. Jilke, "Disk Array Mass Storage Systems: The New Opportunity," Amperif Corporation, September 1986.

[Lei] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers* C-34 (April 1985), 344–354, also appears in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, (April 1983), 71-80.

[Mag] N. B. Maginnis, "Store More, Spend Less: Mid-Range Options Around," *Computerworld* (November 16, 1986), 71.

[NoVa] M. H. Nodine and J. S. Vitter, "Optimal Deterministic Sorting on Parallel Memory Hierarchies," Department of Computer Science, Brown University, CS-92-38, August 1992.

[NoVb] M. H. Nodine and J. S. Vitter, "Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks," Brown University, CS-90-04, February 1990, also appears in shortened form in "Large-Scale Sorting in Parallel Memories," *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC (July 1991), 29–39.

[PGK] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings ACM SIGMOD Conference* (June 1988), 109–116.

[Sch] M. E. Schulze, "Considerations in the Design of a RAID Prototype," U. C. Berkeley, UCB/CSD 88/448, August 1988.

[Uni] University of California at Berkeley, "Massive Information Storage, Management, and Use (NSF Institutional Infrastructure Proposal)," Technical Report No. UCB/CSD 89/493, January 1989.

[ViSa] J. S. Vitter and E. A. M. Shriver, "Optimal Disk I/O with Parallel Block Transfer," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (May 1990), 159–169.

[ViSb] J. S. Vitter and E. A. M. Shriver, "Algorithms for Parallel Memory I: Two-Level Memories," Brown University, CS-90-21, September 1990.