Optimal Deterministic Sorting on Parallel Disks [*]

Mark H. Nodine[†]
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II
Austin, TX 78738
U.S.A.

Jeffrey Scott Vitter[‡]
Department of Computer Science
Purdue University
West Lafayette, IN 47907–2107
U.S.A.

Abstract

We present a load balancing technique that leads to 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 of the D disks 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, and it is possible to implement using only striped writes (but independent reads). In a companion paper, we show how to modify the algorithm to achieve optimal CPU time, even on parallel processors and parallel memory hierarchies.

Keywords: Parallel disks, parallel sorting, load balancing, distribution sort, input/output complexity.

AMS subject classification: 68P10 (Theory of data: searching and sorting).

[*]An earlier and shortened version of this work appeared in [NoVc].

[†]This research was done while the author was at Brown University and Motorola, supported in part by an IBM Graduate Fellowship, by NSF research grants CCR–9007851 and IRI–9116451, and by Army Research Office grant DAAL03–91–G–0035. E-mail address: nodine@intrinsity.com.

[‡]This research was done while the author was at Brown and Duke Universities. Support was provided in part by Army Research Office grant DAAL03–91–G–0035 and by National Science Foundation grants EIA–9870734 and CCR–0082986. E-mail address: jsv@purdue.edu.

1 Introduction

Input/Output communication (I/O) between primary and secondary memory is a major bottleneck in many important computations. The I/O bottleneck is especially troublesome when parallel processors are used. Of particular importance is the problem of 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. Block transfer is motivated by the fact that the seek time is usually much longer than the time needed for transmitting a single 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 [CLG, GHK, 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

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

where $M < N$, and $1 \le DB \le M/8$. In each I/O, D blocks of B records each can be transferred simultaneously, as illustrated in Figure 1. The notions of "block" and "track" are used interchangeably in this model for ease of exposition; a track on one of the disks corresponds to one location where a single block of records is stored. This notion differs from actual hardware, in which a track on disk is usually divided into multiple blocks, but such considerations are easy to deal with effectively. For simplicity in this paper, we also adopt the block-track correspondence.

The above 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

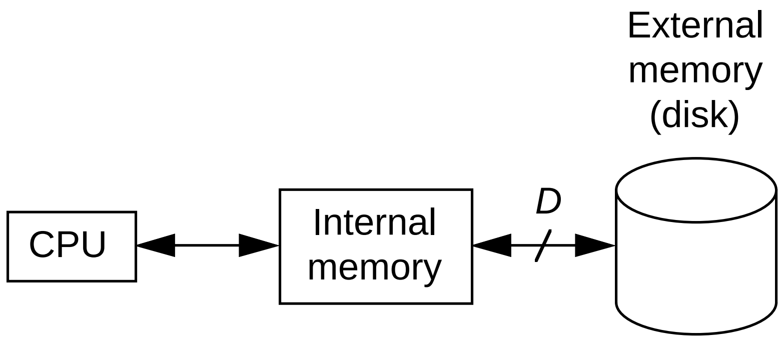

Figure 1: A simple D-parallel two-level memory model.

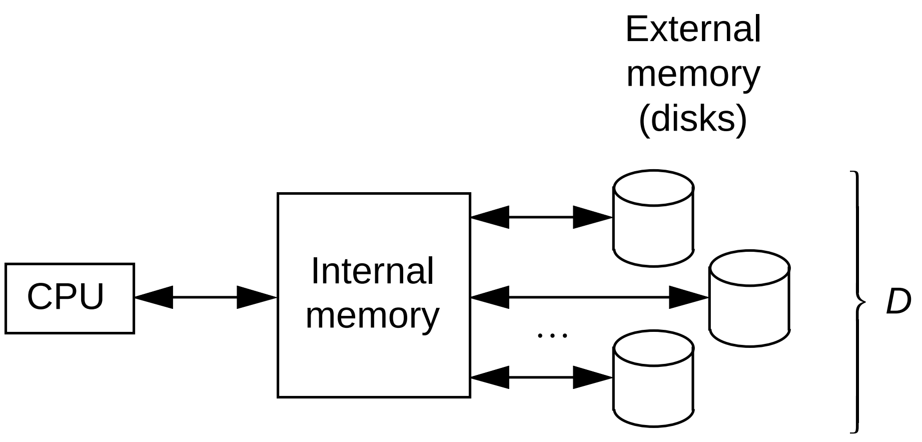

Figure 2: Model of parallel disks

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 minor case in which M and B are extremely small, in which case the comparison model is 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 *parallel disk model*, in which the secondary storage is partitioned into D physically distinct disk drives [ViS], as in Figure 2. Each head of a multi-head drive can 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.

Our measure of performance in this paper is the number of parallel I/Os required. The model also applies to the case in which the D disks are controlled by a P CPUs each with enough internal memory to store M/P records; the P 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 when P is large enough.

Disk striping is a commonly-used technique in which the D disks are synchronized, so that each of the D blocks accessed during an I/O is located at the same relative position on its disk. This technique effectively transforms the D disks into a single disk with larger block size $B' = DB$. Merge sort combined with disk striping is deterministic, but when D is large 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 consists of consecutive records in the total

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

sorted list. The algorithm makes a pass over the data, separating the records into their respective buckets, and then sorts each bucket recursively. Finally, the buckets are concatenated 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 [ViS]. Their algorithm meets the I/O lower bound (1) for the more lenient model of [AgV], and thus the algorithm is optimal. Randomization is used to distribute each of the buckets evenly over the D disks so that they can 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 did not seem applicable to finding an optimal deterministic algorithm for parallel memory hierarchies.

In this paper we describe Balance Sort, the first known optimal and deterministic sorting algorithm based on distribution sort. Our main result is that Balance Sort is optimal for parallel disk sorting in terms of the number of I/O steps.

Theorem 1 *Balance Sort sorts N records in the parallel disk model in $O(N \log N)$ internal processing time and with*

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

I/Os, which are optimal. The lower bounds apply to both the average and the worst case. The I/O lower bound does not require the use of the comparison model of computation, except for the case when M and B are extremely small with respect to N, namely, when $B\log(M/B) = o(\log(N/B))$. The internal processing lower bound uses the comparison model.

Balance Sort provides a more practical and yet deterministic alternative to the optimal randomized algorithm of Vitter and Shriver [ViS]. 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, which does not lend itself to efficient implementation in memory hierarchies.

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 [CLG, Sch].

Subsequently to our work, Aggarwal and Plaxton [AgP] developed another optimal deterministic external sorting algorithm. Their algorithm is based on the hypercube sorting method of Cypher and Plaxton [CyP] which has higher constant factors. Barve, Grove, and Vitter [BGV] developed a practical sorting method based on merge sort, which does probably better than disk striping even for moderate values of D. The algorithm, however, is not theoretically optimal for some values of N, D, and B. This algorithm was improved by Barve and Vitter [BaV].

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 5 explains the significance of this algorithm.

2 How to Balance

For simplicity, we assume that the N keys are distinct; this assumption is realizable by appending to each key the record's initial location. *Distribution sort* is a sorting paradigm that chooses $S-1$ partitioning elements that partition the file to be sorted into S ranges of values called *buckets*. The file is processed and the buckets are formed, and then the buckets are sorted recursively. The sorted buckets are concatenated in the right order to produce a completely sorted file. For example, if we want to sort checks from a bank statement, and the checks fall in the range 350–399, we might choose for our partitioning elements the values 360, 370, 380, and 390. We then partition the checks into stacks (buckets). Once each stack is sorted, we then put the sorted 350s first, followed by the sorted 360s, etc., and the checks are fully sorted.

In this paper we describe a new algorithm called Balance Sort that is based on the distribution sort paradigm. In Balance Sort, we find $S-1$ partitioning elements that partition the file into S nearly equal-sized blocks. We read D records into memory in parallel, partition them, and write them back to the disks in parallel. We read, partition, and write the next D records, and so on, until the whole file has been partitioned into buckets. The crucial problem we address is how 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 a novel Balance routine that guarantees that each bucket is spread evenly among the disks. In this section, we describe the Balance routine. In the next section, we describe how to use Balance to achieve the optimal number of I/Os. For simplicity of exposition, we assume that each block stores only one record; that is, the block size is $B = 1$. We show in Section 3.2 how to accommodate arbitrary block sizes.

Let x_{bd} be the number of records written to disk d from bucket b during the partition process; we call the $S \times D$ matrix $X = (x_{bd})$ the *histogram matrix*. The number of parallel reads required to read bucket b into internal memory during the next level of recursion is $r_b = \max_{1\le d\le D}\{x_{bd}\}$. Our goal is to minimize the sum

$$\sum_{1\le b\le S} r_b.$$

We process the file one track at a time (all of the disks are synchronized for reading) and

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 $\lceil N/D \rceil$
    read $D$ records in parallel into array $T$ in internal memory from track $t$
    { Compute the matching matrix $Z$ }
    for $i := 1$ to $D$
        $\tau[i] :=$ Bucket$(T[i], P)$
        for $d := 1$ to $D$
            $z_{i,d} := y_{\tau[i],d}$
    find a permutation $\pi$ of $T$ that is a min-cost matching of matching matrix $Z$
    for $d := 1$ to $D$
        increment $x_{\tau[\pi(d)],d}$
        increment $y_{\tau[\pi(d)],d}$
    decrement any row of $Y$ that has no zeros
    write out $T$ in the permuted order
write out the location matrix $L$
```

use a greedy strategy that attempts to minimize the increase of this sum. We compute an $S \times D$ matrix Y, called the *skew matrix*, that measures how unbalanced each bucket is. We define

$$y_{bd} = x_{bd} - x_b^{\min}, \qquad \text{where } x_b^{\min} = \min_{1 \le d \le D} \{x_{bd}\}.$$

Intuitively, the skew matrix element y_{bd} indicates how many more records of bucket b will be on disk d after all the records in the bucket that reside on some other disk have been read. We call y_{bd} the *skew* of bucket b on disk d.

In Algorithm 1, we try 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. We assume that the procedure Bucket(a, P) returns the number of the bucket to which a belongs, based on the set of partitioning elements P. Algorithm 1 is an on-line algorithm in that it makes its decisions about where to place elements based only on the records it has already seen (as codified in the arrays X and Y). The CPU time complexity of finding a permutation for the min-cost matching is $O(D^3)$, since Z is a $D \times D$ matrix.

It should be noted that after the first level of recursion, there is no longer any guarantee that all buckets will occupy successive tracks on a disk. This difficulty is easily overcome 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. We augment each block with a pointer to the next block to read on that disk for the same bucket. (This pointer actually points backwards to the most recently written track for the bucket on that disk.) At the end, we are left with an $S \times D$ matrix giving the locations where the next level of recursion must begin processing. We refer to this matrix as the *location matrix*, L. Thus, starting from the final configuration of L, we can read the bucket backwards, using only a constant amount of overhead in each block.

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

Lemma 1 *The column sums $x_d = \sum_{1 \le b \le S} x_{bd}$ in the histogram matrix X 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. $\square$

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. 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, if we assign a bucket b to a disk d for which y_{bd} is larger than $y_{bd'}$ for all other disks d', we increase the total number of reads needed to process the bucket beyond the number of reads that are strictly needed. By doing a min-cost matching, we attempt to spread the records of each bucket evenly over the D disks simultaneously. The skew matrix has some useful properties, as evidenced by the following simple lemmas:

Lemma 2 *Every bucket b contains at least one 0 in the skew matrix Y.*

Proof: By definition of the skew matrix, those disks d for which $x_{bd} = \min_{1 \le d \le D} \{x_{bd}\}$ will have $y_{bd} = 0$. $\square$

Lemma 3 *In the skew matrix Y, we have $y_{bd} \ge 0$ for all $1 \le b \le S$ and $1 \le d \le D$.*

Proof: By definition of the skew matrix, we subtract from each row of the histogram matrix X the minimum value for each row. $\square$

Lemma 4 *The column sums $y_d = \sum_{1 \le b \le S} y_{bd}$ in the skew matrix Y are all equal.*

Proof: This lemma 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. $\square$

Before we proceed, 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 the next track has been processed using the min-cost matching to assign buckets to disks, and after any rows without zeroes have been decremented by 1. If $y'_{bd} > y_{bd}$ then we say that bucket b has been *promoted* on disk d.

Lemma 5 *At most one bucket is promoted on any given disk on any track. Furthermore, that bucket can increase its skew by at most* 1 *on the disk.*

Proof: Exactly one bucket has its value in the histogram matrix incremented by 1 for the given disk, and hence it is the only one whose value can increase in the skew matrix column for that disk. Since the value in the histogram matrix is incremented by 1, the maximum possible increase in its skew is also 1. □

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 attain the larger skew in y' must have a skew of max in Y on every disk for which the skew in y' is $max + 1$.*

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 $i + j \geq 2max$. Since no element of Y is greater than max, it follows that $i = j = max$. Since this fact is true for every pair of buckets that achieve a new maximum skew, the result follows. □

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 number of I/Os for reading all the buckets back in, it has been difficult to prove this conjecture in the general case. Intuitively, the algorithm tries to spread the buckets among the disks that in a way that conflicts least with what has been previously placed. If it can 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 that we propose instead 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 the 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, given in Algorithm 2, uses $\alpha = \frac{1}{2}$. 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 (i.e., 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 parts of Algorithm 2 that differ from Algorithm 1 are underlined. Algorithm 2 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:

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

Invariant 2 *Every row of A contains at least $\lceil D/2 \rceil$ 0s (or equivalently, every row contains no more than $\lfloor D/2 \rfloor$ 1s).*

It is easy to see that Invariant 2 is maintained, since the smallest $\lceil D/2 \rceil$ elements all become 0, 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 invariant is maintained. Auxiliary matrix A is initially binary (all 0s).

We will prove some 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 2 establishes that Invariant 1 holds.

Lemma 7 *The only entries of the auxiliary matrix 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* 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 2 [Balance(P, T)]

```
    { $P$ contains the partition elements; $T$ says where the first block is on each disk. }
    { 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 $Mem$ in internal memory from track $t$
        { Compute the matching matrix $Z$ }
        for $i := 1$ to $D$
            $\tau[i] := \text{bucket}(Mem[i], P)$
            for $d := 1$ to $D$
(1)             $\underline{z_{i,d} := a_{\tau[i],d}}$
        find a permutation $\pi$ of $Mem$ that is a min-cost matching with matching matrix $Z$
        write out $Mem$ in the permuted order
        for $d := 1$ to $D$
            increment $x_{\tau[\pi(d)],d}$
(2)     $\underline{A := \text{ComputeAux}(X)}$ { $X$ is the histogram matrix. }
(3)     $\underline{\textbf{if}\text{ there is a 2 in }A}$
(4)         $\underline{\text{Rebalance}(A, X)}$
(5)         $\underline{A := \text{ComputeAux}(X)}$
(6) write out the location matrix $L$
```

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 $D$
            $a_{bd} := \max(0, x_{bd} - m)$
```

Proof: The 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. Since the median for a bucket either stays the same or increases as a result of the matching, the difference between any element not matched and the median either stays the same or decreases. The elements of A involved in the match increase by only 1, and since the median does not decrease, the difference between the matched elements and the median can increase by at most 1. □

Algorithm 4 [Rebalance(A, X)]

```
    { The array $R$ tells us which bucket to read from each disk }
    { The array $W$ tells us on which disk to write the block }
    for $d := 1$ to $D$
        $r_d := 0$
        $w_d := 0$
    { Create a bipartite matching problem }
    $U := \{d \mid a_{bd} = 2$ for some $b \in 1, \ldots, S\}$
    $V := \{1, \ldots, D\} - U$
    $E := \emptyset$
    for $i := 1$ to $|U|$
        $d := U_i$
(1)     $b :=$ (unique) bucket such that $a_{bd} = 2$
        $r_d := b$
        for $j := 1$ to $|V|$
            $d' := V_j$
            if $a_{bd'} = 0$
                $E := E \cup (d, d')$
    { We will swap a pair of blocks for every edge in the following match }
(2) find a maximum bipartite matching on the graph $G = (U \cup V, E)$
    for each match $(d, d')$
        $w_d := d'$
        { 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 $r_d \neq 0$
    write the block read from disk $d$ onto disk $w_d$ if $r_d \neq 0$
```

We now show that lemmas similar to Lemmas 5 and 6 for skew matrices also hold for the auxiliary matrix.

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

Proof: By Lemma 7, only those elements involved in a match can increase and even then, only by 1. Since A is binary prior to the match, the largest element will be 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 can become a 2 for that disk. □

Lemma 9 *Assume that the auxiliary matrix A is binary at line (1) of Algorithm 2 and let $\mathcal{D}$ be the set of disks having a 2 in A at line (3) during the same iteration. Then every bucket that has a 2 in A at line (3) must have had a 1 in A for every disk in $\mathcal{D}$ back in line (1) during the same 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 revert to 1s during the call to ComputeAux at line (2) of the algorithm, if the median for that bucket happens to increase. $\square$

The call to ComputeAux at line (2) guarantees that every bucket in A at line (3) of Algorithm 2 has at least $\lceil D/2 \rceil$ 0s. We need one more fact about A before we can complete our analysis of the Rebalance routine.

Lemma 10 *Assuming that the auxiliary matrix A is binary at line* (1) *of Algorithm* 2*, at most $\lfloor D/2 \rfloor$ disks will have* 2*s 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 1s" in the matching matrix Z. But each bucket has at least $\lceil D/2 \rceil$ zeroes, which means that the maximum width of the block of 1s is $\lfloor D/2 \rfloor$. $\square$

Now that we know the structure of the auxiliary matrix A at the call to Rebalance at line (4) of Algorithm 2, we need to show some facts about the Rebalance subroutine. Lemma 8 implies that the bucket at line (1) of Algorithm 4 is unique.

Lemma 11 *Assuming that the auxiliary matrix A is binary at line* (1) *of Algorithm* 2*, the maximum matching at line* (2) *of Algorithm* 4 *will include all vertices in U.*

Proof: By Lemma 10, we have $|U| \leq \lfloor D/2 \rfloor$. Each vertex in U has at least $\lceil D/2 \rceil$ outgoing edges. Each edge that is chosen for the matching 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 (namely, choosing the first edge for the first vertex of U, the first remaining edge for the second vertex of U, and so on) produces a matching that includes all the vertices of U. $\square$

At long last, we are ready to prove that Invariant 1 holds.

Theorem 2 *The auxiliary matrix A is a binary matrix 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 0.

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, and thus at 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 guarantees that the call to ComputeAux in line (5) of Algorithm 2 will result in $a_{bd} \leq 1$ and $a_{bd'} \leq 1$, since the median element cannot be on disk d, and so there is no chance of lowering the median. By Lemma 11, in a single parallel I/O operation we can simultaneously rebalance all of the 2s that appeared in A . 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). $\square$

The following theorem tells how well each bucket is balanced.

Theorem 3 *The number of I/Os needed to read any bucket b during the next level of recursion of Balance Sort will be no more than a factor of about 2 above the optimal.*

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

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

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

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

from the optimal. □

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 has the characteristic that $x_{0d} = 0$ for all $1 \leq d \leq D$, so in particular, its row of the matching matrix contains all 0s. The part of the algorithm that increments the values of X should ignore bucket 0.

Step (6) requires writing a set of cardinality DS, which can be done in S/B parallel writes.

We can now bound the total number of parallel I/Os needed for balancing N records into S buckets:

Theorem 4 *A phase of Balance requires no more than about $2N/DB$ parallel reads and about $4N/DB + S$ parallel writes.*

Proof: We know from Theorem 3 that the N records we are processing occupy about $2N/DB$ tracks. Each of these tracks can give rise to at most one parallel read and two parallel writes: one read and write for the original assignment to disks according to the min-cost matching, and at most one parallel write in rebalancing. Those blocks that require rebalancing do not need to be written in the first parallel write. The extra S parallel writes are for writing the location matrix L. □

Subsequently to our work, an alternative definition of the auxiliary matrix was proposed, although without the development of a full sorting algorithm. It has a similar effect of making each bucket balanced within a factor of 2; the term a_{bh} is defined to be 1 when the number of blocks per bucket is more than twice the desired evenly-balanced number [Gro].

Algorithm 5 [Balance_Sort(N, T)]

```
       if $N \leq M$
(1)        Read the whole bucket into memory
           Sort internally
(2)        Write the bucket out again
       else
           $S := \min\{\lfloor 2\sqrt{M/B} \rfloor, \lfloor 2N/M \rfloor\}$
(3)        $P :=$ ComputePartitionElements($S$)
           { The $T$ array gives the starting tracks on each disk. }
(4)        Balance($P, T$)
           for $b := 1$ to $S$
(5)            $T :=$ Read $b$th row of $L$ { Written in Balance }
               $N_b :=$ number of elements in bucket $b$
(6)            Balance_Sort($N_b, T$)
(7)            Append sorted bucket to output area
```

3 The Full Sorting Algorithm

In this section we describe how to fit the balancing algorithm into an optimal algorithm for sorting, which we call Balance Sort. We assume 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. We assume for simplicity without loss of generality that N, M and B are powers of 2 and that $\sqrt{M/B}$ is an integer. The Balance routine has up until now assumed that $B = 1$. In Subsection 3.2, we describe how to handle the general case $B \geq 1$.

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 $S = \min\{\lfloor 2\sqrt{M/B} \rfloor, \lfloor 2N/M \rfloor\}$ in order to produce optimal performance, as shown in Section 4.

The next few subsections establish the proof of Theorem 1, that Balance Sort uses an optimal number of I/Os. The lower bounds on the number of I/Os needed to sort follows from [AgV], where the same lower bound was established for a more powerful (but less realistic) model. The lower bound is based only on routing concerns and does not assume a comparison model of computation, except in a pathological case. The well-known $\Omega(N \log N)$ lower bound for internal processing does assume the comparison model of computation.

3.1 Finding the partition elements

The following procedure for finding the partition elements is based on that of [ViS], and is presented in Algorithm 6. The algorithm is deterministic.

Lemma 12 *Algorithm 6 finds $S - 1$ partition elements such that, for any bucket b, the* *number of elements in that bucket N_b obeys the constraint*

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

Proof: We constructed $\mathcal{M}'_i$ to consist of every $\lfloor S/8 \rfloor$th record. Thus, each element of $\mathcal{M}'_i$ represents exactly $\lfloor S/8 \rfloor$ elements that are less than or equal to it but larger than the previous element of $\mathcal{M}'_i$. So the element b_j defined as the $jN/(\lfloor S/8 \rfloor S$th element of $\mathcal{M}'$ has at least

$$\frac{jN}{\lfloor S/8 \rfloor S} \lfloor S/8 \rfloor = j\frac{N}{S}$$

elements less than or equal to it. But it could have more, since each of the other $\lceil N/M \rceil - 1$ memoryloads could have up to $\lfloor S/8 \rfloor - 1$ records less than or equal to b_j. So b_j has at most $jN/S + z$ records less than or equal to it, where

$$z = (\lceil N/M \rceil - 1)(\lfloor S/8 \rfloor - 1).$$

The largest a bucket could be is if the previous partition element has rank jN/S and the next has rank $(j+1)N/S + z$, giving it a size of $N/S + z$. Similarly, the smallest a bucket could be is $N/S - z$. So to prove the lemma, we need to demonstrate that

$$z = (\lceil N/M \rceil - 1)(\lfloor S/8 \rfloor - 1) \le \frac{N}{2S}.$$

Since $N > M$, we can divide by $\lceil N/M \rceil - 1$. We thus need to show that

$$S^2 - S((S \bmod 8) - 1) \le \frac{4N}{\lceil N/M \rceil - 1},$$

where we have expanded $\lfloor S/8 \rfloor$ in terms of the modulo function. This inequality is clearly satisfied if we can show that

$$S^2 \le \frac{4N}{\lceil N/M \rceil - 1}.$$

But we know that $S \le 2\sqrt{M/B}$ so $S^2 \le 4M/B$. The lemma will be proved if we show that

$$\frac{4M}{B} \le \frac{4N}{\lceil N/M \rceil - 1}.$$

We expand the ceiling

$$\left\lceil \frac{N}{M} \right\rceil = \frac{N}{M} - \frac{M \bmod N}{M} + 1 - [M \mid N],$$

where the notation

$$[M \mid N] = \begin{cases} 1 & \text{if } M \text{ divides } N \\ 0 & \text{otherwise.} \end{cases}$$

So

$$\frac{4N}{\lceil N/M \rceil - 1} = \frac{4M}{1 - \frac{1}{N}((M \bmod N) + M[M \mid N])}.$$

Algorithm 6 [ComputePartitionElts(S) returns P]

```
for each memoryload of records $\mathcal{M}_i$ ($1 \le i \le \lceil N/M \rceil$)
    Read $\mathcal{M}_i$ into memory
    Sort internally
    Construct $\mathcal{M}'_i$ to consist of every $\lfloor S/8 \rfloor$th record
    Write $\mathcal{M}'_i$
$\mathcal{M}' := \mathcal{M}'_1 \cup \cdots \cup \mathcal{M}'_{\lceil N/M \rceil}$
for $j := 1$ to $S - 1$
    $b_j := \text{Select}(\mathcal{M}', 4Nj/S^2)$
return($\{b_j\}$)
```

Now $(M \bmod N) + M[M \mid N] = M$ if $M \mid N$ and is at most $M - 1$ if $M \bmod N \neq 0$. Thus,

$$\frac{1}{N}((M \bmod N) + M[M \mid N]) < 1,$$

since $M < N$. Therefore, the fraction is defined and exceeds $4M$. Putting the two halves together, we have that

$$\frac{4M}{B} \le 4M < \frac{4M}{\lceil N/M \rceil - 1},$$

which completes the proof of the lemma. □

Algorithm 6 was analyzed in [ViS] 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 7, which computes the kth smallest of n elements in $O(n/DB)$ I/Os. It does this task by recursively subdividing about an element that is guaranteed to be close to the median.

3.2 Dealing with larger blocks

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/Os performed is not affected; the changes only affect the amount of memory needed.

At the point in the loop where D records are read from disk in parallel into array Mem, 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 5 *The amount of space needed for buffering and collecting into buckets will never exceed M as long as $M \ge 8DB$.*

Algorithm 7 [Select($\mathcal{S}, k$)]

```
$t := \lceil |\mathcal{S}|/M \rceil$
**for** $i := 1$ to $t$
    Read $M$ elements in parallel (from $\lceil M/DB \rceil$ tracks, the last possibly partial)
    Sort internally
    $R_i :=$ the median element
**if** $t = 1$
    return($k$th element)
$s :=$ the median of $R_1, \ldots, R_t$ using algorithm from [BFP]
Partition into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 < s$ and $\mathcal{S}_2 \geq s$
$k_1 := |\mathcal{S}_1|$
$k_2 := |\mathcal{S}_2|$
**if** $k \leq k_1$
    return(Select($\mathcal{S}_1, k$))
**else**
    return(Select($\mathcal{S}_2, k - k_1$))
```

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 exceed $SB + 2DB$. But $S \leq \sqrt{M/B}$ throughout the algorithm and by hypothesis, $M \geq 8DB$, so the amount of space needed is no more than

$$\begin{aligned} SB + 2DB &\leq \sqrt{MB} + 2DB \\ &\leq \sqrt{MDB} + 2DB \\ &\leq \sqrt{M\frac{M}{8}} + 2\frac{M}{8} \\ &< M \end{aligned}$$

□

4 Analysis

In this section, we prove Theorem 1 that Balance Sort is optimal with respect to the number of parallel I/O steps and number of disk blocks in the parallel disk model.

4.1 I/O performance of the sorting algorithm

We have numbered each of the steps of Algorithm 5 that can cause an I/O operation. Let $T(N)$ denote the number of I/O steps needed for sorting N records. Steps (1) and (2) occur only at the innermost level of recursion. From Theorem 3, the bucket will take no more than $3N/DB$ reads and writes.

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 $\lceil N/DB \rceil$
    read $D$ blocks in parallel into input array
    Gather records into blocks that each represent only one bucket
    if there are at least $D$ full blocks or $t = \lceil N/D \rceil$
        if there are at least $D$ full blocks
            Define array $Mem$ to comprise $D$ full blocks
        else
            Define array $Mem$ to be any $D$ blocks that need writing
        { Compute the matching matrix $Z$ }
        for $i := 1$ to $D$
            $\tau[i] :=$ bucket($Mem[i]$)
            for $d := 1$ to $D$
                $z_{i,d} := a_{\tau[i],d}$
        find a permutation $\pi$ of $Mem$ that is a min-cost matching with matching matrix $Z$
        write out $Mem$ in the permuted order
        for $d := 1$ to $D$
            increment $x_{\tau[\pi(d)],d}$
        $A :=$ ComputeAux($X$)
        if there is a 2 in $A$
            Rebalance($A, X$)
            $A :=$ ComputeAux($X$)
write out the location matrix $L$
```

Step (3), as mentioned, has been shown by [ViS] 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 4, requires no more than $6N/DB$ I/Os. Step (5) can be done in a single parallel read, since the cardinality of the set is only D, for a total of S reads. The number of I/Os for step (6) is

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

where N_b is the number of elements in bucket b. Let $N_b = N/S + \Delta_b$, where $|\Delta_b| \le N/(2S)$ and $\sum_{1 \le b \le S} \Delta_b = 0$. Finally, step (7) 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. Recalling that $S \le 2N/M$ and that $M \ge 2DB$, we see that $S \le N/DB$, so we can fold any terms with S into an overall $O(N/(DB))$ term. This analysis gives us the recurrence

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

By the definition of S, we have

$$S = \begin{cases} \left\lfloor \dfrac{2N}{M} \right\rfloor & \text{if } N \le \dfrac{M^{3/2}}{\sqrt{B}} \\ \left\lfloor 2\sqrt{\dfrac{M}{B}} \right\rfloor & \text{if } N \ge \dfrac{M^{3/2}}{\sqrt{B}} \end{cases}$$

Let us consider the case $N \le M^{3/2}/\sqrt{B}$. We get the recurrence

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

where $|\Delta_b| \le M/4$. In particular, each bucket has size at most $3M/4$, and so we can sort it internally. Thus,

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

since $M > DB$.

When $N \ge M^{3/2}/\sqrt{B}$, we demonstrate by induction below that

$$T(N) \le \alpha \frac{N}{DB}\frac{\log(N/B)}{\log(M/B)},$$

where α is some positive constant of proportionality. The base case clearly satisfies the constraint. Thus we need to show that

$$\sum_{1\le b\le S} T\left(\frac{N}{S} + \Delta_b\right) + \gamma\frac{N}{DB} \le \alpha \frac{N}{DB}\frac{\log(N/B)}{\log(M/B)},$$

where γ is an upper bound of the constant of proportionality assumed by the "big Oh" in the recurrence. By the induction hypothesis, the left-hand side is bounded by

$$\sum_{1\le b\le S} \alpha\left(\frac{N}{SDB} + \frac{\Delta_b}{DB}\right)\frac{\log(N/SB) + \log(1 + \Delta_b/(N/S))}{\log(M/B)} + \gamma\frac{N}{DB}.$$

Since $|\Delta_b|/(N/S) \le 1/2$, and $\log(1+x) < x$ whenever $x < 1$, this quantity is no more than

$$\sum_{1\le b\le S} \alpha\left(\frac{N}{SDB} + \frac{\Delta_b}{DB}\right)\frac{\log(N/SB) + \Delta_b/(N/S)}{\log(M/B)} + \gamma\frac{N}{DB}.$$

By splitting the sum into individual terms and using the fact that $\sum_{0\leq b\leq S}\Delta_b = 0$, we get

$$\alpha\frac{N}{DB}\frac{\log(N/SB)}{\log(M/B)} + \frac{\alpha S}{NDB\log(M/B)}\sum_{1\leq b\leq S}{\Delta_b}^2 + \gamma\frac{N}{DB}.$$

Since $|\Delta_b| \leq N/2S$, we can bound the above by

$$\begin{aligned}&\alpha\frac{N}{DB}\frac{\log(N/SB)}{\log(M/B)} + \frac{\alpha S}{NDB\log(M/B)}S\left(\frac{N}{2S}\right)^2 + \gamma\frac{N}{DB}\\ &= \alpha\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)} - \alpha\frac{N}{DB}\frac{\log S - 1/4}{\log(M/B)} + \gamma\frac{N}{DB},\end{aligned}$$

where we have removed the S from the $\log(N/SB)$ and rearranged. Since $S \geq 2\sqrt{M/B}-1$, the above does not exceed

$$\alpha\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)} - \alpha\frac{N}{DB}\frac{\log(2\sqrt{M/B}-1) - 1/4}{\log(M/B)} + \gamma\frac{N}{DB}.$$

Now $M/B > 1$, so if we replace the "-1" above with "$-\sqrt{M/B}$", the value will increase to

$$\begin{aligned}&\alpha\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)} - \alpha\frac{N}{DB}\frac{\log(\sqrt{M/B}) - 1/4}{\log(M/B)} + \gamma\frac{N}{DB}\\ &= \alpha\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)} - \alpha\frac{N}{DB}\left(\frac{1}{2} - \frac{1}{4\log(M/B)}\right) + \gamma\frac{N}{DB}.\end{aligned}$$

The bound is thus proved as long as

$$\gamma\frac{N}{BD} - \alpha\frac{N}{BD}\left(\frac{1}{2} - \frac{1}{4\log(M/B)}\right) \leq 0$$

or

$$\gamma \leq \left(\frac{1}{2} - \frac{1}{4\log(M/B)}\right)\alpha.$$

Since the coefficient of α is always between 1/2 and 1/4, if we choose $\alpha > 4\gamma$, the induction condition is held independently of M and B. Thus,

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

which completes the proof of Theorem 1.

4.2 Memory usage of the sorting algorithm

We still need to show 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 any $0 < \beta < 1/2$, the amount of primary memory space needed for the data structures of the sorting algorithm is $O(M^{1/2+\beta}) = o(M)$.*

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]. If D does not grow as fast as $M^{1/2}$, then the memory cannot be exceeded. 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

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

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

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

□

5 Conclusions

In this paper, we have described a new algorithm called Balance Sort for sorting on parallel disk subsystems. This algorithm has the following advantages over the previously known optimal deterministic algorithm, Greed Sort [NoVb]:

1. The new algorithm is also applicable to parallel processors and to parallel memory hierarchies, as described in the companion paper [NoVa].

2. The hidden constants in the big-oh notation are small. The problem with the Greed Sort algorithm 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 is unclear how to adapt Greed Sort to use only striped writes.

A promising approach to balancing that the authors first considered is to do a greedy balance via min-cost matching on the placement matrix. We conjecture that such an approach results in globally balanced buckets.

There is still a significant hurdle to overcome before Balance Sort can be used to do sorting optimally in terms of internal processing time on parallel processors and in terms of time on parallel memory hierarchies. In the models that we consider in the companion paper [NoVa], there is a requirement that the internal processing must be doable in $O((N/P)\log P)$ parallel time in order to achieve optimal performance, when P CPUs are used. The min-cost matching in the Balance routine is operating on a 0-1 matrix, and 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. The proof that the "block of 1s" is present does not depend upon maximum matching, but only that the matching is maximal. 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 $O(\log^3 n)$ for a problem of size n and with n processors [IsS], which is not good enough to get optimal performance on the parallel memory hierarchies with effective logarithmic cost functions. The interesting and elegant adjustments necessary to obtain optimal performance on P processors are the subject of the companion paper [NoVa]. The sorting algorithm in the companion paper [NoVa] for the $D \geq 1, P \geq 1$ disk model gives an alternate optimal algorithm for the $P = 1$ model we consider in this paper.

The two distribution sort methods Balance Sort and Greed Sort perform write operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach investigated more recently (since the appearance of this work in conferences) is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. A summary of some approaches along those lines and the issues that arise are described in the survey article [Vit].

6 References

[AgP] Alok Aggarwal and C. Greg Plaxton, "Optimal Parallel Sorting in Multi-Level Storage," *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, Washington, D.C. (January 1994).

[AgV] Alok Aggarwal and Jeffrey Scott Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* 31 (September 1988), 1116–1127.

[BGV] Rakosh D. Barve, Edward F. Grove and Jeffrey Scott Vitter, "Simple Randomized Mergesort on Parallel Disks," *Parallel Computing* 23 (1997), 601–631.

[BaV] Rakosh D. Barve and Jeffrey Scott Vitter, "A Simple and Efficient Parallel Disk Mergesort," *Theory of Computing Systems* 35 (March/April 2002), 189–215.

[BFP] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest and Robert E. Tarjan, "Time Bounds for Selection," *J. Computer and System Sciences* 7 (1973), 448–461.

[CLG] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz and D.A. Patterson, "RAID: High-performance, Reliable Secondary Storage," *ACM Computing Surveys* 26 (June 1994), 145–185.

[CyP] Robert E. Cypher and C. Greg Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers," *Journal of Computer and System Sciences* 47 (1993), 501–548.

[Flo] Robert 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] Garth Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz and David A. Patterson, “Coding Techniques for Handling Failures in Large Disk Arrays,” *Algorithmica* 12 (1994), 182–208.

[GiS] David Gifford and Alfred Spector, “The TWA Reservation System,” *Communications of the ACM* 27 (July 1984), 650–665.

[Gro] Edward Grove, 1994, Private communication.

[HoK] Jia-Wei 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*, Milwaukee, WI (May 1981).

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

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

[NoVa] Mark H. Nodine and Jeffrey Scott Vitter, “Optimal Deterministic Sorting on Parallel Processors and Parallel Memory Hierarchies,” *submitted for publication.*

[NoVb] Mark H. Nodine and Jeffrey Scott Vitter, “Greed Sort: Optimal Deterministic Sorting on Parallel Disks,” *Journal of the Association for Computing Machinery* 42 (1995), 919–933.

[NoVc] Mark H. Nodine and Jeffrey Scott Vitter, “Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors (extended abstract),” *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany (June 1993).

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

[Vit] Jeffrey Scott Vitter, “External Memory Algorithms and Data Structures: Dealing with Massive Data,” *ACM Computing Surveys* (June 2001), also appears in updated form at http://www.cs.purdue.edu/homes/jsv/Papers/Vit.IO_survey.pdf, 2007.

[ViS] Jeffrey Scott Vitter and Elizabeth A. M. Shriver, “Algorithms for Parallel Memory I: Two-Level Memories,” *Algorithmica* 12 (1994), 110–147.