Optimal Deterministic Sorting in Parallel Memory Hierarchies

Mark H. Nodine and Jeffrey Scott Vitter

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-38
August 1992

Optimal Deterministic Sorting in Parallel Memory Hierarchies

Mark H. Nodine and *Jeffrey Scott Vitter*

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

August 31, 1992

Abstract

We present a general deterministic sorting strategy that is applicable to a wide variety of parallel memory hierarchies with parallel processors. The simplest incarnation of the strategy is an optimal deterministic algorithm called Balance Sort for external sorting on multiple disks with a single CPU. Balance Sort was the topic of a previous report. This report shows how to adapt Balance Sort to sort deterministically in parallel memory hierarchies. The algorithms so derived will be optimal for all parallel memory hierarchies for which an optimal algorithm is known for a single hierarchy. In the case of D disks, P processors, block size B, and internal memory size M, they are optimal in terms of I/Os for any $P \leq M$ and $DB < M$, and in terms of internal processing time except when $P > M \log \min\{M/B, \log M\} / \log M$ and $\log M/B = o(\log M)$.

1 Introduction

Input/Output communication (I/O) between primary and secondary memory is a major bottleneck in many important computations, and it 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 reside 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 in place. An increasingly popular way to get further speedup is to use many disk drives working in parallel [GHK, GiS, Jil, Mag, PGK, Uni].

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

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

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. Their measure of performance is the number of parallel I/Os required; They ignore internal computation time. 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]

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

All logarithms in this paper are base 2 logarithms. Their lower bound is based solely on routing arguments, except for the pathological case in which M and B

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

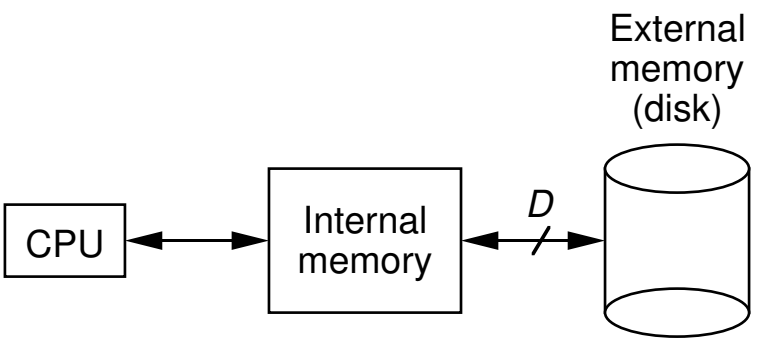

Figure 1: A simple D-parallel two-level memory model.

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.

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 $O(N \log_S N)$.

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 2a. (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.

Vitter and Shriver presented a randomized version of distribution sort in the D-disk model 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 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. 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.

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 $\log(M/B)$.

The question posed by Vitter and Shriver 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 obviously 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.

The companion paper [NoVa] described 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*. In this report, we describe how to adapt Balance Sort

to parallel memory hierarchies, including a generalization of the parallel disk model in which P CPUs are working in parallel.

Section 2 describes the memory models considered in this report. Section 3 lists our main results. In Section 4, we give an algorithm that is optimal for all the parallel multi-level hierarchies. In Section 5, we show how to alter the algorithm to deal with parallelism of CPUs in the parallel disk model. Finally, we summarize our results in Section 6.

2 Memory Models

2.1 Parallel disk models

The first memory hierarchy we consider is the two-level memory hierarchy, known as the disk model. In this case, there is a P-processor CRCW (Concurrent Read, Concurrent Write) PRAM with some limited amount of global internal memory capable of storing M records, and a large external memory, in the form of D disks. The N records of the file reside on the disk and are grouped into blocks of size B.

$$\begin{aligned} N &= \#\text{ records in the file} \\ M &= \#\text{ records that can fit in internal memory} \\ P &= \#\text{ internal processors} \\ D &= \#\text{ disks} \\ B &= \#\text{ records per block} \end{aligned}$$

We require that $1 \leq P \leq M$ and $2BD \leq M$.

In a single I/O, each of the D disks can simultaneously transfer a block of B records. Our main measure of performance is the number of I/Os, but we will also consider the internal processing time. The difficulty in designing optimal algorithms is dealing with the partitioning of secondary storage into separate disks.

Our disk model is a generalization of the Vitter-Shriver model in which we allow more than one processor. Figure 2a at the end of the paper shows the uniprocessor ($P = 1$) multiple disk model, corresponding to the Vitter-Shriver model. The more general model, in which the internal processing is done on a CRCW PRAM with P processors, is shown in Figure 2b for the special case $P = D$, although we allow P to be as large as M.

2.2 Parallel multilevel hierarchies

The simplest multilevel hierarchy model is the Hierarchical Memory Model (HMM) proposed by Aggarwal *et al.* [AAC], depicted in Figure 3a. In the $\text{HMM}_{f(x)}$ model, access to memory location x takes $f(x)$ time. Figure 3a suggests the $\text{HMM}_{\lceil \log x \rceil}$ model, where each layer in the hierarchy has a size that is some constant multiple of

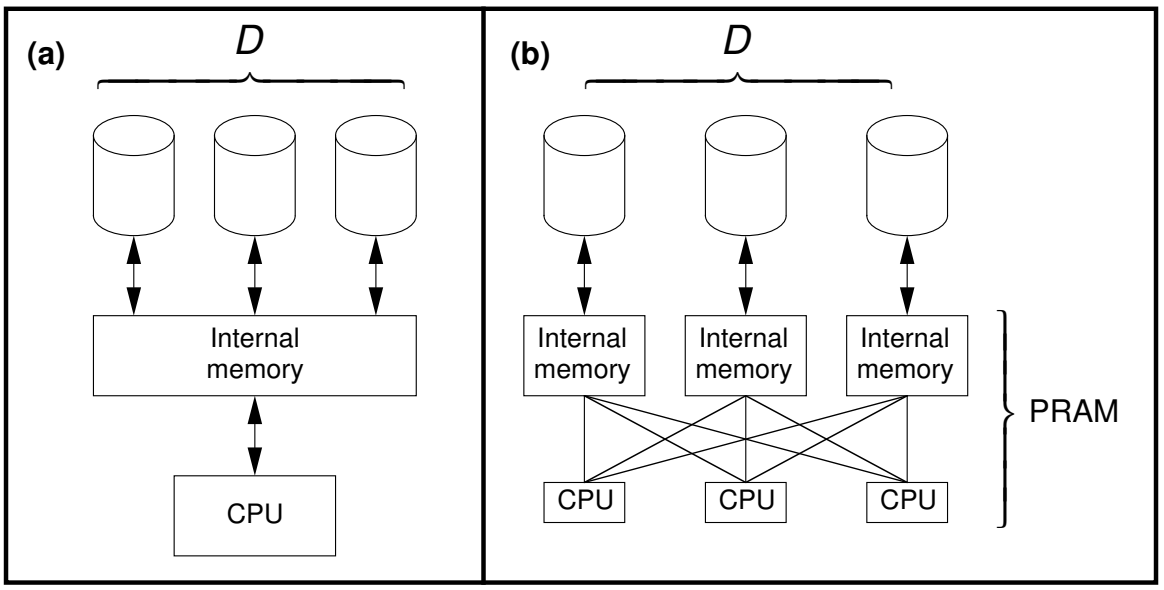

Figure 2: (a) The parallel disk model. Each of the D disks can simultaneously transfer B records to and from internal memory in a single I/O. The internal memory can store $M \geq DB$ records. (b) Multiprocessor generalization of the I/O model in (a), in which each of the $P = D$ internal processors controls one disk and has an internal memory of size M/P. The P processors and their memories function collectively as a CRCW PRAM with global memory size M.

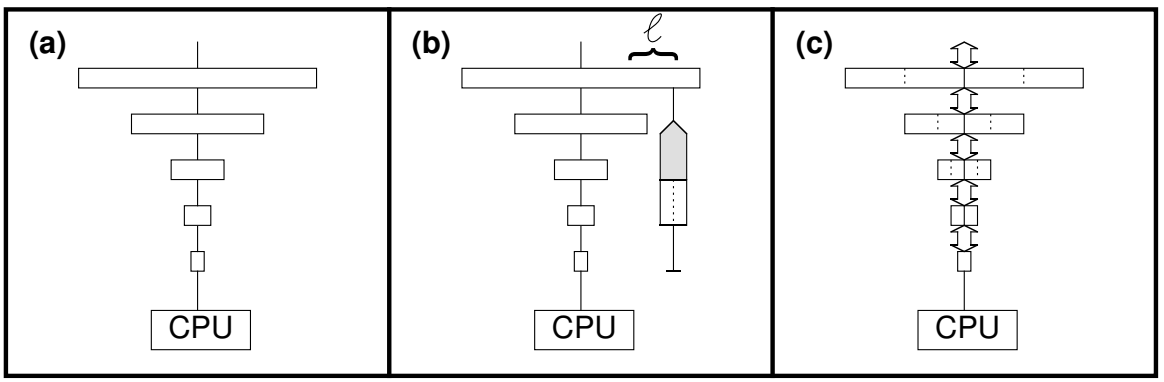

Figure 3: Multilevel hierarchy models. (a) The HMM model. (b) The BT model. (c) The UMH model.

the size of the previous layer. Accesses to records in the first layer take one time unit; in general each record in the nth layer takes n time units. Figure 3a can actually be taken as representative of every *well-behaved* cost function $f(x)$. The definition of a well-behaved cost function is that there exists a constant c such that $f(2x) \leq c\, f(x)$ for all x. Therefore, access to layer n will take no more than cn time units, so we have only a constant-factor slowdown. Our model is pessimistic in that $f(x)$ could

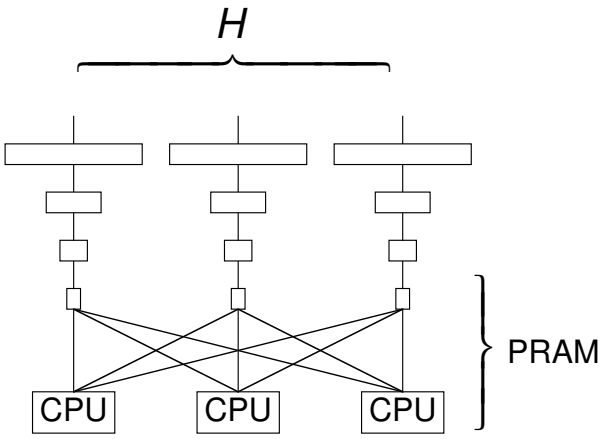

Figure 4: Parallel multilevel memory hierarchies. The H hierarchies (of any of the types listed in Figure 3) are connected at the base level by an H-processor CRCW PRAM with H global memory locations.

grow much slower or not at all.

An elaboration of HMM is the Block Transfer (BT) model of Aggarwal *et al.* [ACSa], depicted schematically in Figure 3b. Like HMM, it has a cost function $f(x)$, but additionally it simulates the effect of block transfer by allowing the $\ell + 1$ locations $x - \ell, \ldots, x$ to be accessed for cost $f(x) + \ell$. So in the figure, the main cost is incurred in injecting the "needle"; pushing additional items in (or pulling them out) once the needle has been placed takes only one time unit per item.

A possibly more realistic memory hierarchy is the Uniform Memory Hierarchy (UMH) of Alpern *et al.* [ACF], depicted in Figure 3c. UMH has several additional parameters:

$$\begin{aligned} \rho &= \text{expansion factor between levels} \\ \alpha &= \#\text{ of base memory locations} \\ b(\ell) &= \text{bandwidth function} \end{aligned}$$

In particular, the ℓth level of the hierarchy contains $\alpha\rho^\ell$ blocks, each holding ρ^ℓ records, for a total of $\alpha\rho^{2\ell}$ records in the ℓth level. Additionally, between level ℓ and level $\ell+1$ is a bus with a bandwidth $b(\ell)$, so that a block on level ℓ can be transferred to or from level $\ell+1$ in time $\rho^\ell/b(\ell)$. The base memory level is level 0. A novel feature of the UMH model is that all the buses can be active simultaneously, allowing for a kind of controlled data parallelism. The restriction that allows only one bus to be active at a time is called the SUMH model. A less restrictive model that we proposed earlier [ViN] and which corresponds closely to currently used programming constructs, is the RUMH model, in which several buses are allowed to be active simultaneously if they are cooperating on a single logical block move operation.

As with two-level hierarchies, multilevel hierarchies can be parallelized. The parallelization is done in the same way as for the two-level memory hierarchies, as shown

in Figure 4. The base memory level of the H hierarchies forms an H-processor CRCW PRAM with H memory locations. We assume that the hierarchies are all of the same kind and all have the same parameters. The parallel hierarchical models for HMM, BT, etc., are denoted as P-HMM, P-BT, and so on.

3 Main Results

In this section, we present our main results. The modified Balance Sort algorithm we describe in the next section gives us optimal deterministic algorithms for all the models we consider. In particular we get deterministic (as well as more practical) versions of the optimal randomized algorithms based on [ViSa]and [ViN]. We also improve upon the deterministic Greed Sort algorithm in [NoVb], which was optimal only for the parallel disk models and could not be used optimally for hierarchical memories. The lower bounds are proved in [ViSa] (Theorems 1 and 2) and [AgV] (Theorem 3).

Theorem 1 *In the P-HMM model, the time for sorting is*

$$\Theta\left(\frac{N}{H}\log N \log\left(\frac{\log N}{\log H}\right)\right) \qquad \text{if } f(x) = \log x;$$

$$\Theta\left(\left(\frac{N}{H}\right)^{\alpha+1} + \frac{N}{H}\log N\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha > 0.$$

The upper bounds are given by a deterministic algorithm based on Balance Sort. In the lower bound for the $f(x) = x^{\alpha}$ case, the $(N/H)\log N$ term requires the comparison model of computation.

Theorem 2 *In the P-BT model, the time for sorting is*

$$\Theta\left(\frac{N}{H}\log N\right) \qquad \text{if } f(x) = \log x;$$

$$\Theta\left(\frac{N}{H}\log N\right) \qquad \text{if } f(x) = x^{\alpha},\ \ 0 < \alpha < 1;$$

$$\Theta\left(\frac{N}{H}\log N \log\frac{N}{H}\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha = 1;$$

$$\Theta\left(\left(\frac{N}{H}\right)^{\alpha} + \frac{N}{H}\log\frac{N}{H}\log H\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha > 1.$$

The upper bounds are given by a deterministic algorithm based on Balance Sort. The $(N/H)\log N$ terms in the lower bounds require the comparison model of computation.

Our techniques can also be used to transform the randomized P-UMH algorithms of [ViN] into deterministic ones in our model, as we discuss in Section 4.5.

Theorem 3 *The number of I/Os required for sorting N records in the parallel disk model is*

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

The upper bound is given by a deterministic algorithm based on Balance Sort, which also achieves simultaneously optimal $\Theta((N/P)\log N)$ internal processing time, unless $P > M \log\min\{M/B, \log M\}/\log M$ and $\log M/B = o(\log M)$. The lower bounds apply to both the average case 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.

4 Parallel Memory Hierarchies

This section describes the deterministic algorithm that serves as the basis for the upper bounds in Theorems 1 through 7. Subsection 4.1 gives the overall sorting algorithm. In Subsection 4.3, we analyze the algorithm for the P-HMM model. Subsection 4.4 analyzes the algorithm for the P-BT model. Subsection 4.5 analyzes the algorithm for the P-UMH and related models.

4.1 The sorting algorithm

Algorithm 1 gives the algorithm used for sorting in parallel memory hierarchies. We assume that the records have been augmented with a pointer so that each record can point to the next record in the same bucket on the same hierarchy; this expansion of the records results in no more than a constant increase in the space needed (hence running time).

The difficult part of the algorithm is in making sure that the buckets are (approximately) evenly distributed among the hierarchies; the Balance routine and its descendants handles this complexity. In order for the balancing to occur in optimal deterministic time, it turns out that it is necessary to do partial striping of the hierarchies, so that we will have only H' *virtual hierarchies* with logical blocks of size $B = H/H'$. We will use $H' = \sqrt{H}/\log H$.

There are a number of parameters in the algorithm that merit explanation, some of which are "global" and accessed by the subroutine Balance. We use "global" in quotes, because there is actually a distinct copy of each variable for each level of recursion. They can be thought of as allocated by the Sort routine in local storage and then having a pointer passed to the routines that need to access them.

N = # of elements to sort

T = array of H' elements pointing to the starting block on each virtual hierarchy

S = # of partition elements

Algorithm 1 [Sort(N, T)]

```
    if N ≤ 12H
        n := ⌈N/H⌉
        for m := 1 to n
(1)         Read H locations to the base level (last read may be partial)
            Sort internally
(2)         Write back out again
(3)     Do binary merge sort of the up to 12 sorted lists
    else
(4)     E := ComputePartitionElements(S)
        { The T array says what location to start with on each hierarchy }
(5)     Initialize X, L
(6)     Balance(T)
        for b := 1 to S
(7)         T := Read bth row of L
            N_b := number of elements in bucket b
(8)         Sort(N_b, T)
(9)         Append sorted bucket to output area
```

X = (global) $S \times H'$ *histogram* matrix (described later)

E = (global) array of $S - 1$ partition elements

L = (global) $S \times H'$ *location* matrix (described later)

We should make a remark about storage of these "global" arrays. We will assume throughout that each array will be stored at the lowest level in which it fits and that accesses to all elements of the array have the same cost. We are justified in doing this because we restrict ourselves to well-behaved cost functions, as described in Section 2, so that the levels give within a constant factor an upper bound of the actual cost. The alert reader will notice that we often will be storing more than one matrix at the same level. Since the level is chosen so that each individual matrix (barely) fits in the level, that level would not have enough space for all of the matrices combined. As long as we limit ourselves to a constant number of matrices inhabiting the same level (which we do), we will only be overlooking a constant factor in the cost. Taking these liberties simplifies the analysis considerably without doing damage to the integrity of our results.

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. We will determine the value of S differently depending upon which hierarchical model we are using in order to produce optimal performance, as shown in the subsections that analyze the performance for the various hierarchical memory models.

Algorithm 2 [ComputePartitionElements(S)]

partition into G groups of at most $\lceil N/G \rceil$ elements, $\mathcal{G}_1, \ldots, \mathcal{G}_G$
for $g := 1$ to G
(1) sort $\mathcal{G}_g$ recursively
(2) set aside every $\lfloor \log N \rfloor$th element into C
(3) sort C using binary merge sort with hierarchy striping
(4) $e_j :=$ the $\lfloor jN/((S-1)\log N) \rfloor$th smallest element of C

Algorithm 2 gives the routine for computing the partition elements. The specific value of G used in the algorithm is dependent upon which hierarchical memory model is being considered. Although the array $E = \{e_j\}$ is listed as being returned from the routine, it is too large to fit into the base memory level, and is therefore implemented as a "global" variable. This algorithm is essentially identical to that given in [ViSa]. We show the following lemma, which will lead to an analysis of the effectiveness of the approximate partitioning elements in dividing the records into nearly equal-sized buckets.

Lemma 1 *Assuming that all keys are distinct, if we choose every pth element from the subsets in line (2) of Algorithm 2, the amount of slop in the size of the buckets produced is less than pG, that is, each bucket b will have N_b elements in it satisfying*

$$\frac{N}{S} - pG < N_b < \frac{N}{S} + pG.$$

Proof: We collect a set C of N/p elements by choosing every pth element from each of the G subsets, where $p = \log N$. We then sort C and choose the ith partition element to be the iqth element of C, where $q = N/((S-1)\log N)$. Let c_k denote the kth smallest element of C. The minimum rank in the original set of the c_{iq} is iqp, since each of the iq elements in C less than or equal to c_{iq} represents p elements in the original set. The maximum rank in the original set of c_{iq} occurs if each of the other $G-1$ subsets has $p-1$ elements greater than c_{iq}. Thus, the amount of slop does not exceed $(G-1)(p-1)$. □

Corollary 1 *If we choose every $\log N$th element and we choose G such that $G \log N \leq N/S$ then we get $0 < N_b < 2N/S$ for any bucket b.*

Notice that following the call to ComputePartitionElements, the original set of N records is left as G sorted subsets of approximately N/G records each. This fact is crucially important in allowing for partial hierarchy striping, as described in the next subsections.

Algorithm 3 gives the routine for balancing the buckets. A *track* is a set of H records, one from each hierarchy. We do not assume that the records are at the same

Algorithm 3 [Balance(T)]

for each track
(1) read track
(2) assign records to buckets (in parallel)
collect buckets into blocks of size H/H' (all elements of block from same bucket)
(3) update the placement matrix X
(4) $A :=$ ComputeAux(X)
if there is a 2 in A
(5) Rebalance(A, X)
(6) update the internal pointers of the blocks and the location matrix L
(7) write track

location on each hierarchy; however we assume that the records all come from the same level. The locations of the first track on each hierarchy are gotten from the array T. Since T has only H' elements and we have H hierarchies, the first value in T refers to the first H/H' consecutive hierarchies, the second value in T to the second H/H' hierarchies, and so on. The subsequent tracks are read by following pointers present within each hierarchy; these pointers were placed there by the previous level of recursion. The first level of recursion simply reads the tracks in order.

Collecting the buckets into virtual blocks is surprisingly easy. The value of G is chosen in such a way that the initial sorts of size N/G will put together an average of at least H/H' records from each bucket in each of the S buckets. Thus, we lose no more than a factor of two due to internal fragmentation of the blocks, as shown in the next lemma.

Lemma 2 *If a set containing k distinct values has $N \geq kB$ elements, then it can be broken into no more than $2\lceil N/B \rceil$ blocks such that each block contains all the same value.*

Proof: Write out as many complete blocks all containing the same value as possible. Then write out partially filled blocks containing all the same value. The number of partially filled blocks is at most k. By supposition, $k \leq \lfloor N/B \rfloor$. There are no more than $\lceil N/B \rceil$ completely filled blocks. So the total number of blocks will be no more than

$$\left\lceil \frac{N}{B} \right\rceil + \left\lfloor \frac{N}{B} \right\rfloor \leq 2 \left\lceil \frac{N}{B} \right\rceil .$$

□

We will apply this lemma using $k = S$ and $B = H/H'$. Processing an input track results in processing up to two output tracks; lines (3) to (7) will be executed once for each of the output tracks. We will not consider this issue further as it results in at most a factor of two in the running time.

Algorithm 4 [ComputeAux(X)]

for $b := 1$ to S
  $m :=$ $\lceil 2H'/3 \rceil$th smallest element of $x_{b1}, \ldots, x_{bH'}$
  for $h := 1$ to H' **in parallel**
    $a_{bh} := \max(0, x_{bh} - m)$

Algorithm 5 [Rebalance]

$\mathcal{H} :=$ {virtual hierarchies with the first $\lfloor H'/3 \rfloor$ 2s}
(1) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {virtual hierarchies with the second $\lfloor H'/3 \rfloor$ 2s}
(2) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {virtual hierarchies with the third $\lfloor H'/3 \rfloor$ 2s}
(3) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {any remaining virtual hierarchies with 2s}
(4) Shuffle($\mathcal{H}$)

The *histogram matrix* $X = \{x_{bh}\}$ specifies how the buckets are distributed among the hierarchies; $x_{bh} =$ # of records of bucket b on hierarchy h. Updating the histogram matrix on line (3) simply means that if hierarchy h has read a record from bucket b, that it increments the value of x_{bh}. *Auxiliary matrix* $A = \{a_{bh}\}$ is designed to let us know if the placement becomes too badly skewed. The *location matrix* $L = \{l_{bh}\}$ tells what location was written last on each hierarchy for each bucket. We update the internal pointer in the record to hold the old value of L for the hierarchy and bucket and store the location to which the bucket is written (which is the same location from which the original record on that hierarchy was read) into L. Thus, by starting at l_{bh} and following the pointers, we can read all the records from bucket b on hierarchy h in the inverse of the order in which they were processed at the previous level of recursion.

Algorithm 4 gives the ComputeAux routine. It is listed as returning a matrix A, but since this matrix is larger than will fit into the base memory level, it is another "global" variable. ComputeAux is guaranteed to set at least $\lceil 2H'/3 \rceil$ elements of every row of A to 0. We wish to maintain the buckets so that no element of A is larger than 1. If any element of A becomes larger than 1, then we call the routine Rebalance to restore the balance. In fact, Rebalance will take care of as many buckets as have become unbalanced during the last track.

Algorithm 5 gives the rebalancing routine. At most H' 2s could be introduced

Algorithm 6 [Shuffle($\mathcal{H}$)]

```
    { Create a bipartite matching problem }
    U := H
    V := {1, ..., H'}
    E := ∅
    for i := 1 to |U|
        h := U_i
        b := (unique) bucket such that a_bh = 2
        for j := 1 to |V| in parallel
            h' := V_j
(1)         if a_bh' = 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 hierarchy }
    { The array W will tell us on what hierarchy to write the block }
    for h := 1 to H'
        r_h := 0
        w_h := 0
    for each match (U_i, V_j) in parallel
        h := U_i
        r_h := b
        w_h := V_j
(3)     Update X to reflect the swap
(4) Read the last block of bucket r_h on virtual hierarchy h if r_h ≠ 0
(5) Write the block read from virtual hierarchy h onto virtual hierarchy w_h if r_h ≠ 0
```

by the track being processed into the auxiliary matrix A. The reason is that only H' values in the histogram matrix X are incremented by the track, one for each virtual hierarchy, and only values of the histogram matrix that are incremented can become 2. We call the routine Shuffle to remove any 2s that are introduced $\lfloor H'/3 \rfloor$ at a time; it follows that at most 4 calls to Shuffle will remove all the 2s that may have been introduced.

Algorithm 6 shows the routine Shuffle, which is able to remove up to $\lfloor H'/3 \rfloor$ 2s in a single parallel memory reference. Lines (4) and (5) of Algorithm 6 are done in parallel. The routine is based on the simple observation that if we read a block from a virtual hierarchy h corresponding to a bucket b that has $a_{bh} = 2$ and we write it to another virtual hierarchy h' for which $a_{bh'} = 0$, then we will have removed the 2. We set up the matching so that we can accommodate up to $\lfloor H'/3 \rfloor$ 2s simultaneously in

a single parallel memory reference. We prove these two facts in the next two lemmas.

Lemma 3 *Reading a block from a virtual hierarchy h corresponding to a bucket b that has $a_{bh} = 2$ and writing it to a virtual hierarchy h' for which $a_{bh'} = 0$ and updating X to reflect the change will result in $a_{bh} \leq 1$ and $a_{bh'} = 0$ if ComputeAux is called to recompute A after the operation.*

Proof: There are two cases to consider: (1) incrementing $x_{bh'}$ does not change the $\lceil 2H'/3 \rceil$th smallest element of row b and (2) incrementing $x_{bh'}$ does change the $\lceil 2H'/3 \rceil$th smallest element. In either case, x_{bh} decreases by at least 1 with respect to the $\lceil 2H'/3 \rceil$th smallest element and so a_{bh} will no longer be 2. Also in either case, $x_{bh'} \leq$ the $\lceil 2H'/3 \rceil$th smallest element, so $a_{bh'}$ will be 0. □

Lemma 4 *The maximum matching at line (2) of routine Shuffle will always match all $\lfloor H'/3 \rfloor$ elements of U.*

Proof: We can give an elementary sequential algorithm to accomplish this task. First, notice that G is a rather special bipartite graph: it has no more than $\lfloor H'/3 \rfloor$ vertices on the left, H' vertices on the right, and every vertex on the left has edges to at least $\lceil 2H'/3 \rceil$ vertices on the right. So if we simply take the first vertex on the left and match it to any vertex on the right, it can remove at most one possible matching vertex on the right for each of the remaining vertices on the left. We can now match the second vertex on the left, and so on. We see that even when we get around to matching the $\lfloor H'/3 \rfloor$th vertex on the left, there are at least $\lceil 2H'/3 \rceil - \lfloor H'/3 \rfloor \geq \lfloor H'/3 \rfloor$ possibilities left for the match. Hence, the match will always complete. □

As can be seen by the proof to Lemma 4, it is not necessary for us to use a maximum matching routine in Algorithm 6; any maximal matching will be a maximum matching. In order to achieve optimal time for memory hierarchies with a logarithmic access cost, we need to be able to do the matching in time $O(\log H)$. Unfortunately, even the fastest known deterministic parallel algorithm for maximal matching with n items is $O(\log^2 n)$ with a quartic number of processors for dense graphs (which we have) [Luba], or $O(\log^3 n)$ with a quadratic number of processors [IsS] or $n^2/\log n$ processors [GoS]. Since we have $n = H'$, this means that the fastest known algorithm is $\Theta(\log^2 H)$, and still doesn't work since we have only $H = \Theta((H')^2 \log^2 H')$ processors. Subsection 4.2 addresses the subject of how to do the matching efficiently.

For the current purposes it is enough to recognize that the Shuffle routine does successfully remove $\lfloor H'/3 \rfloor$ 2s from the auxiliary matrix A. By "removing a 2" from the auxiliary matrix, we mean that if the auxiliary matrix is immediately recomputed after the operation, an element that was a 2 will now be no more than 1, and no 2s will have been introduced by the operation. Hence it follows that if the auxiliary matrix were computed immediately after the call to Rebalance, there will be no elements in it larger than 1.

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

Algorithm 7 [Fast_Match]

(1) **while** there are unmatched vertices on the left $\Lambda = \{\lambda_i\}$
(2) **while** λ_i has not picked adjacent previously unmatched right vert.
λ_i picks a random vertex on right $\{1, \ldots, H'\}$
(3) **if** vertex on right is picked by more than one on left, smallest wins
add left-right pair to matching

Theorem 4 *Any bucket b will take no more than a factor of around 3 above the optimal number of tracks to read.*

Proof: Let m_b denote the $\lceil 2H'/3 \rceil$th element in bucket b. When the Balance routine terminates, we have $\max_h a_{bh} \leq 1$ for any bucket b. Thus, 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 $\lceil H'/3 \rceil m_b$ blocks in it, since the $\lceil H'/3 \rceil$ virtual hierarchies with the most blocks from bucket b each have at least m_b such blocks, so it requires at least

$$\left\lceil \frac{\lceil H'/3 \rceil m_b}{H'} \right\rceil \geq \left\lceil \frac{m_b}{3} \right\rceil$$

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

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

from the optimal. (The expression could be as large as 4 if H' is divisible by 3 and $m_b = 3$.) □

Algorithm 3 assumes that it reads H' blocks on every track, whereas the previous phase does not guarantee that every track up to the last is fully used. Adjusting the algorithm for partial tracks is simple: if no block is available to be read on some hierarchy h, we pretend that we read from a dummy bucket 0. Bucket 0 will be have the characteristic that $x_{0h} = 0$ for all $1 \leq h \leq H'$, 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.

4.2 How to do deterministic log-time matching

In this subsection, we discuss how to do deterministic log-time matching. This is not the ground-breaking result it would seem to be, because we are able to take advantage of very special properties of our particular matching problem. Recall that we have $k \leq \lfloor H'/3 \rfloor$ vertices on the left, each of which has edges to at least $\lceil 2H'/3 \rceil$ of the H' vertices on the right. We start by giving a randomized algorithm called

Fast_Match, shown in Algorithm 7, for doing our maximal bipartite matching in logarithmic time with a linear (in H') number of processors. Step (3) of this routine is the only place in the entire algorithm where we require the power of the CRCW-PRAM. To prove that this matching can be done in logarithmic time with H' processors, we will show in the following lemmas that Loop (1) of Algorithm 7 will be executed only expected $O(\log H')$ times, and Loop (2) only expected $O(1)$ times. Notice that we can implement the algorithm assigning one processor to each vertex on the right and left, using only $O(H')$ processors.

Lemma 5 *The expected number of iterations of Loop (2) of Algorithm 7 is* $O(1)$.

Proof: Since $k \leq \lfloor H'/3 \rfloor$, at most $\lfloor H'/3 \rfloor - 1$ vertices on the right could already have been matched when we get to Loop (2), or Loop (1) would already have terminated. Since each vertex has at least $\lceil 2H'/3 \rceil$ edges, it follows that at least

$$\left\lceil \frac{2H'}{3} \right\rceil - \left\lfloor \frac{H'}{3} \right\rfloor + 1 \geq \left\lfloor \frac{H'}{3} \right\rfloor + 1$$

of its edges are unmatched. Generating a number from 1 to H' will result in one of these edges being chosen in expected time

$$\frac{H'}{\lfloor H'/3 \rfloor + 1} \leq 3.$$

□

Loop (2) has each unmatched vertex on the left pick a previously unmatched vertex on the right to which it has an edge; this operation is equivalent to having every unmatched vertex on the left pick an edge at random in a graph where edges to any matched vertices on the right have been deleted. There is not enough time to do any actual deletions, but we have enough of a surfeit of edges that we can get the same effect in constant time by just ignoring picks to already-matched vertices on the right.

Lemma 6 *The expected number of iterations of Loop (1) of Algorithm 7 is* $O(\log H')$.

Proof: The question to settle is what fraction of the unmatched vertices on the left will be matched after each has randomly picked an edge to a previously unmatched vertex on the right. If we have k unmatched vertices on the left, then each vertex has at least $\lceil 2H'/3 \rceil - k$ edges to unmatched vertices on the right. If we view the selection process sequentially, the probability that vertex 1 hits a previously selected vertex is 0, the probability that vertex 2 will choose the same one that vertex 1 chose is no more than $1/(\lceil 2H'/3 \rceil - k)$, and in general,

$$\Pr\{\text{vertex } i \text{ hits a previously selected vertex}\} \leq \frac{i-1}{\lceil 2H'/3 \rceil - k}.$$

If we let X_i be a 0-1 random variable specifying whether vertex i hits a new vertex (with respect to vertices 1 to $i-1$), then

$$E(X_i) \geq \frac{\lceil 2H'/3 \rceil - k - i + 1}{\lceil 2H'/3 \rceil - k} = 1 - \frac{i-1}{\lceil 2H'/3 \rceil - k}.$$

If we let n be the expected number of vertices that do not collide, we get

$$\begin{aligned} n = \sum_{1 \leq i \leq k} E(X_i) &\geq \sum_{1 \leq i \leq k} \left(1 - \frac{i-1}{\lceil 2H'/3 \rceil - k}\right) \\ &= k\left(1 - \frac{k-1}{2\left(\lceil 2H'/3 \rceil - k\right)}\right). \end{aligned}$$

So a "constant" fraction of the vertices on the left is matched, where the "constant" is

$$1 - \frac{k-1}{2\left(\lceil 2H'/3 \rceil - k\right)}.$$

This "constant" is minimized as k gets large. But we know that $k \leq \lfloor H'/3 \rfloor$, so that the constant is at least $1/2$. Hence, in expected $O(\log H')$ time, we can match all the vertices on the left. □

This algorithm can be derandomized in an efficient way using the techniques of Luby [Luba, Lubb]. First, notice that we have H processors available instead of only H'. When $H' = \sqrt{H}/\log H$, that means that $H = \Theta((H')^2 \log^2 H')$. The above randomized matching algorithm uses only pairwise independence in the analysis of the running time, so we construct a special probability space to take advantage of the pairwise independence. We let q be a prime number in the range $H' \log H' \leq q \leq 2H' \log H'$ and generate a probability space with $H' \log H'$ random variables and q possibilities for each random variable. Those q possibilities will consist of the numbers $1, \ldots, H'$ repeated $\Theta(\log H')$ times. Since q is not divisible by H', we add enough 0s at the end to pad out to exactly q possibilities for each random variable. Selecting a 0 for the random variable will mean that the pick was missed; this modification has no effect on the expected running time and leaves the probability distribution unchanged. We will use q^2 processors (which we have available to us) to search the probability space exhaustively in parallel. Since the algorithm terminates in expected $O(\log H')$ steps using expected $O(H')$ random variables on the average, there must be some point in the probability space that achieves at least the average, and hence that can complete the matching with the given number of random variables in $O(\log H')$ steps. We had to choose q at least as large as the number of random variables due to a technicality in the way that Luby's method achieves pairwise independence.

Notice that the matrix of random variables from which we are sampling occupies only $O((H')^2 \log^2(H')) = O(H)$ space, so that it can all be fit at the base memory level of the hierarchies, and we therefore need not consider the cost of accessing the random variables.

To summarize, we have shown the following theorem.

Theorem 5 *The maximal matching problem can be solved deterministically in* $O(\log H)$ *time using* H *processors.*

4.3 Analysis of P-HMM

We will do the analysis of the sorting algorithm using general $f(x)$ and then compute the specific bound for specific functions $f(x)$. In the P-HMM model, we choose

$$G = \begin{cases} \sqrt{N} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \leq H^2 \end{cases}$$

$$S = \begin{cases} \min\left\{\dfrac{\sqrt{N}}{\sqrt{H}\log H}, \dfrac{\sqrt{N}}{\log N}\right\} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \leq H^2 \end{cases}$$

We show that the virtual blocking can be done and that the buckets are approximately the same size in the following lemmas.

Lemma 7 *With the above values of* G *and* S*, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least* $H/H' = \sqrt{H}\log H$*.*

Proof: The average number of elements per bucket per run is N/SG. When $N > H^2$, we notice that $S \leq \sqrt{N}/(\sqrt{H}\log H)$, so $N/SG \geq \sqrt{H}\log H$ as promised. When $N \leq H^2$, $N/SG = 2\sqrt{H}\log H > \sqrt{H}\log H$. □

Lemma 8 *The buckets created using the above values of* G *and* S *are approximately the same size.*

Proof: The condition we derived in Corollary 1 is that $G\log N \leq N/S$. When $N > H^2$, $S \leq \sqrt{N}/\log N$, so that $N/S \geq \sqrt{N}\log N$. The condition follows directly. When $N \leq H^2$, we rewrite the condition to be $\log N \leq N/SG$. But when $N \leq H^2$, then

$$\log N \leq 2\log H < 2\sqrt{H}\log H = \frac{N}{SG},$$

so the condition is satisfied. □

The recurrence for P-HMM is derived in Table 1. We assume throughout that access to a data structure of size s has a cost of $f(s/H)$. The extra $\log H$ factor on line (2) of Algorithm 6 is due to the matching time. In Algorithm 3, it is possible to assume that the number of iterations of the loop is $O(N/H)$ because of Theorem 4.

Alg.	Line	Repeats	Time/repeat	Total time
6	1	$\frac{(H')^2}{H}$	$f\left(\frac{SH'}{H}\right)$	$\frac{(H')^2}{H} f\left(\frac{SH'}{H}\right)$
	2	1	$(\log H) f\left(\frac{(H')^2}{H}\right)$	$(\log H) f\left(\frac{(H')^2}{H}\right)$
	3	1	$f\left(\frac{SH'}{H}\right)$	$f\left(\frac{SH'}{H}\right)$
	4,5	2	$f\left(\frac{N}{H}\right)$	$2f\left(\frac{N}{H}\right)$
	Total	$C_6 = O\left(\left(\frac{(H')^2}{H}+1\right) f\left(\frac{SH'}{H}\right) + \log H\, f\left(\frac{(H')^2}{H}\right) + f\left(\frac{N}{H}\right)\right)$		
5	1–4	4	C_6	$4C_6$
	Total	$C_5 = O(C_6)$		
3	1,7	$O\left(\frac{N}{H}\right)$	$f\left(\frac{N}{H}\right)$	$\frac{N}{H} f\left(\frac{N}{H}\right)$
	2	1	see text	$\frac{GS}{H} f\left(\frac{S}{H}\right)$
	3,6	$O\left(\frac{N}{H}\right)$	$f\left(\frac{SH'}{H}\right)$	$\frac{N}{H} f\left(\frac{SH'}{H}\right)$
	4	$O\left(\frac{N}{H}\right)$	$\frac{(H')^2}{H} f\left(\frac{SH'}{H}\right)$	$\frac{N(H')^2}{H^2} f\left(\frac{SH'}{H}\right)$
	5	$O\left(\frac{N}{H}\right)$	C_5	$\frac{N}{H} C_5$
	Total	$C_3 = O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + f\left(\frac{SH'}{H}\right) + C_5\right) + \frac{GS}{H} f\left(\frac{S}{H}\right) + \frac{N(H')^2}{H^2} f\left(\frac{SH'}{H}\right)\right)$		
2	1	G	$T\left(\frac{N}{G}\right)$	$G\,T\left(\frac{N}{G}\right)$
	2	G	$\frac{N}{GH\log N} f\left(\frac{\log N}{H}\right)$	$\frac{N}{H\log N} f\left(\frac{\log N}{H}\right)$
	3	1	$\frac{\log N \log\log N}{H} f\left(\frac{\log N}{H}\right)$	$\frac{\log N \log\log N}{H} f\left(\frac{\log N}{H}\right)$
	4	1	$\frac{S}{H}\left(f\left(\frac{S}{H}\right) + f\left(\frac{\log N}{H}\right)\right)$	$\frac{S}{H}\left(f\left(\frac{S}{H}\right) + f\left(\frac{\log N}{H}\right)\right)$
	Total	$C_2 = G\,T\left(\frac{N}{G}\right) + \left(\frac{N}{H\log N} + \frac{\log N \log\log N}{H} + \frac{S}{H}\right) f\left(\frac{\log N}{H}\right) + \frac{S}{H} f\left(\frac{S}{H}\right)$		
1	4	1	C_2	C_2
	5	2	$\frac{SH'}{H} f\left(\frac{SH'}{H}\right)$	$2\frac{SH'}{H} f\left(\frac{SH'}{H}\right)$
	6	1	C_3	C_3
	7	S	$f\left(\frac{SH'}{H}\right)$	$S f\left(\frac{SH'}{H}\right)$
	8	S	$T(N_b)$	$\sum_b T(N_b)$
	9	S	$\frac{N_b}{H} f\left(\frac{N}{H}\right)$	$\frac{N}{H} f\left(\frac{N}{H}\right)$
	Total	$T(N) = \sum_b T(N_b) + C_2 + O\left(C_3 + \left(\frac{SH'}{H} + S\right) f\left(\frac{SH'}{H}\right) + \frac{N}{H} f\left(\frac{N}{H}\right)\right)$		

Table 1: Derivation of the recurrence for P-HMM$_{f(x)}$.

There is a trick used to obtain the running time of assigning records to buckets at line (2) of Algorithm 3. Notice that the records have already been sorted into G sets of size N/G by ComputePartitionElements. It follows that in assigning the records to buckets, we will be using the partition elements G times in order using full parallelism; hence the overall time needed for reading the partition elements is what is given in the table. The evaluation of ComputeAux at line (4) of Algorithm 3 can be done incrementally in the given time.

A major simplification of the recurrence occurs by noticing that $H'/H = O((H')^2/H) = O(1)$, since $H' = \sqrt{H}/\log H$. It is also true that $S = O(N/H)$. To show the base of the recursion, consider what happens when $N \leq H^2$. The first recursive call will have arguments of N/G in the first term and $T(N_b)$ in the sum. But we have arranged it so that $N_b \leq 2N/S = 2N/G$, so the largest argument to a recursive call will be

$$\frac{2N}{G} = \sqrt{8N\sqrt{H}\log H} \leq \sqrt{8\log H}H^{5/4},$$

since $N \leq H^2$. Now consider $N \leq \sqrt{8\log H}H^{5/4}$, and look at the second level of recursion. Here, again, the maximum argument to any recursive call is $2N/G$, or

$$\begin{aligned}\sqrt{8N\sqrt{H}\log H} &\leq \sqrt{8\sqrt{8\log H}H^{7/4}\log H} \\ &= 2^{9/4}H^{7/8}\log^{3/4} H \\ &< 12H\end{aligned}$$

for all H. Thus, when $N \leq H^2$, two further levels of recursion always suffice to bring the maximum size subfile to be sorted to no more than $12H$.

The following lemma helps establish the bound for the base case.

Lemma 9 *Two sorted lists whose length totals N can be merged in P-HMM in time $O((N/H)(f(N/H) + \log H))$.*

Proof: We will always keep $r_1 = \lfloor H/2 \rfloor$ records from list 1 and $r_2 = \lceil H/2 \rceil$ records from list 2 in the base memory level, until one of the lists runs out of records. We start by reading the first r_1 records of list 1 and r_2 records of list 2. This operation will require two parallel reads, since the elements will reside on hierarchies $1, \ldots, r_1$ and $1, \ldots, r_2$, respectively. Between the two reads, it is necessary to shift the items from list 1 by r_2 processors to make room in the base memory level for the items from list 2; this shift operation takes $O(\log H)$, since one could implement it by sorting.

We need to add a field to each record keeping track of which list it originated from. We also keep track of which is the next hierarchy to read for each of the lists and which hierarchy is the first to write with our output elements. At each step, we sort the H records in the base memory locations, shift so that the record with the smallest key is on the next hierarchy to write, and write out the $\lfloor H/2 \rfloor$ records with the smallest keys. We update which hierarchy will be the next to write and count

how many records from each list were written, say t_1 records from list 1 and t_2 records from list 2. We then read in t_1 records from list 1 and t_2 records from list 2, with appropriate shifts before each read, and update the pointers to the next hierarchy to read for each list. Since at each step, we have at least the next $\lfloor H/2 \rfloor$ records from each list, we can always output the next $\lfloor H/2 \rfloor$ records for the merged list. We require $3N/\lfloor H/2 \rfloor$ I/Os to process the N records, each of which takes time $f(N/H)$. We also need $O(\log H)$ time for sorting and shifting between I/Os, yielding a total time that is $O((N/H)(f(N/H) + \log H))$. □

Corollary 2 *When $N \leq 12H$, the amount of time needed to merge two lists is $O(\log H)$.*

Thus, the overall recurrence reduces to

$$T(N) = \begin{cases} G\,T\left(\frac{N}{G}\right) + \sum_b T(N_b) + O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + \log H\right)\right) & \text{if } N > 12H \\ O(\log H) & \text{if } N \leq 12H \end{cases} \tag{2}$$

Lemma 10 *When $f(x) = \log x$, the algorithm given sorts in deterministic time*

$$\frac{N}{H} \log N \log\left(\frac{\log N}{\log H}\right).$$

Proof: Simply plug the correct $f(x)$ into Recurrence (2). When $N > H^2$, the given value of S satisfies the recurrence. When $N \leq H^2$, there are only two further levels of recursion giving time $O((N/H)(f(N/H) + \log H) = O((N/H)\log N)$. □

Lemma 11 *When $f(x) = x^\alpha$, the algorithm given sorts in deterministic time*

$$\left(\frac{N}{H}\right)^{\alpha+1} + \frac{N}{H} \log N.$$

Proof: Plug the correct $f(x)$ into Recurrence (2). When $N > H^2$, the given value of S satisfies the recurrence. When $N \leq H^2$, there are only two further levels of recursion giving time $O((N/H)(f(N/H) + \log N))$, which is exactly the result. □

Lemmas 10 and 11 complete the proof of Theorem 1. In fact, we can go even farther for the P-HMM model and show that the algorithm is uniformly optimal for any well-behaved cost functions $f(x)$ as defined in Section 2.

Theorem 6 *The algorithm given above is optimal for all well-behaved cost functions $f(x)$.*

Proof: This proof is essentially that of the corresponding theorem in [ViSa] for their randomized algorithm. Let $T_{M,H}(N)$ denote the average number of I/O steps the sorting algorithm does with an internal memory of size M, where we allow each hierarchy to move simultaneously, in a single I/O step, a record between internal memory and external memory. Note that we are only viewing the algorithm from the standpoint of I/O, and therefore we can ignore any internal computation time whenever $M > H$. In particular, the matching time does not get counted. When $N > M^2$, the algorithm gives the recurrence

$$T_{M,H}(N) = \sqrt{N} T_{M,H}(\sqrt{N}) + \sum_{1 \le b \le S} T_{M,H}(N_b) + \frac{N}{H},$$

where $\sum_{1 \le b \le S} N_b = N$ and $N_b < 2N/S$ for all $1 \le b \le S$. We have used the cost function

$$f(x) = \begin{cases} 1 & \text{if } x > M/H \\ 0 & \text{otherwise} \end{cases}.$$

When $N < M^2$, we get $T_{M,H} = O(N/H)$. It can be shown that this recurrence is solved by

$$T_{M,H} = O\left(\frac{N}{H}\frac{\log N}{\log M}\right) \tag{3}$$

Aggarwal and Vitter showed a lower bound for the problem of sorting with internal memory size M and no blocking or parallelism of

$$T_M = \Omega\left(\frac{N \log N}{\log M} - M\right),$$

where the "$-M$" term allows up to M items to be present initially in the internal memory. At best we could achieve a speed-up of a factor of H by using H disks in parallel, so dividing T_M by H gives a lower bound on the I/O complexity with H disks. But Equation (3) is within an additive term of $O(M/H)$ of this optimal I/O bound. At the base memory level, we consider $M = H$, and find that we load in $O(N \log N/(H \log H))$ memory loads, each of which takes $O(\log H)$ communication time for matching or sorting. Thus, the amount of time used in the base memory level by the algorithm is $O((N/H) \log N)$, all of which is above the optimal algorithm. Define

$$\delta f\left(\frac{M}{H}\right) = f\left(\frac{M+1}{H}\right) - f\left(\frac{M}{H}\right).$$

Then the total time above the optimal can be found by adding the amount of time used in the base memory level of the algorithm to the amount used incrementally for every memory size between H and N. Thus, we get that the amount of time spent above the optimal is

$$O\left(\frac{N}{H}\log N + \sum_{H \le M < N} \frac{M}{H} \delta f\left(\frac{M}{H}\right)\right)$$

$$
\begin{aligned}
&= O\left(\frac{N}{H}\log N + \frac{N}{H} f\left(\frac{N}{H}\right) - f(1) - \frac{1}{H}\sum_{H \le M < N} f\left(\frac{M+1}{H}\right)\right) \\
&= O\left(\frac{N}{H}\log N + \frac{N}{H} f\left(\frac{N}{H}\right)\right) \qquad (4)
\end{aligned}
$$

where the sum at the end of the first line has been expanded and rearranged to produce the second line and the sum at the end of the second line contains fewer than N items each of which has value no greater than $f(N/H)$. The first term in (4) is the lower bound resulting from using a comparison-based sorting algorithm. The second term in (4) is the minimum amount of time needed to read all of the elements into the base memory level at least once, assuming that the cost function $f(x)$ is well-behaved and that we are therefore justified in counting all the records in the set to be sorted as taking the same amount of time to access as the last record. It follows that the sorting algorithm is optimal. □

4.4 Analysis of P-BT

Essentially the same algorithm will work for the P-BT model as we used in the P-HMM model. The only difference is that in Algorithm 1, we need to add another line right after step (6) to reposition all the buckets into consecutive locations on each virtual memory hierarchy. This repositioning is done on a virtual-hierarchy-by-virtual-hierarchy basis, using the generalized matrix transposition algorithm given in [ACSa].

We concentrate in this section on the cost function $f(x) = x^\alpha$, where $0 < \alpha < 1$. Deterministic algorithms for $f(x) = x^\alpha$ with $\alpha \ge 1$ have been reported previously [ViSa]. The upper bound for $f(x) = \log x$ can be no larger than that for $f(x) = \sqrt{x}$ since $\log x < \sqrt{x}$ for all $x > 0$; since the lower bounds are identical it follows that the algorithm for $f(x) = \sqrt{x}$ applied to hierarchies with cost function $f(x) = \log x$ is optimal.

In the algorithm given, we will choose

$$
G = \begin{cases} \dfrac{1}{4\log H} N^{1-\alpha} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \le H^2 \end{cases}
$$

$$
S = \begin{cases} \dfrac{N^\alpha}{\sqrt{H}\log N} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \le H^2 \end{cases}
$$

We show that the virtual blocking can be done and that the buckets are approximately the same size in the following lemmas.

Lemma 12 *With the above values of G and S, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least $H/H' = \sqrt{H}\log H$.*

Proof: The average number of elements per bucket per run is N/SG. When $N > H^2$, $N/SG = \sqrt{H}\log N \geq \sqrt{H}\log H$. When $N < H^2$, the values of S and G are the same as in the P-HMM case, so the same proof holds as before. □

Lemma 13 *The buckets created using the above values of G and S are approximately the same size.*

Proof: The condition we derived in Corollary 1 is that $G\log N \leq N/S$. When $N > H^2$, we require $N^{1-\alpha}\log N \leq \sqrt{H}N^{1-\alpha}\log N$. When $N < H^2$, the values of S and G are the same as in the P-HMM case, so the same proof holds as before. □

As with the P-HMM algorithm, there are at most two additional levels of recursion after $N \leq H^2$. [ViSb] showed that two lists can be merged in P-BT using the same amount of time as that needed to touch all the elements.

We need to make one more change to the algorithm for BT hierarchies, but one that is hard to write explicitly. Aggarwal *et al.* gave an algorithm called the "touch" algorithm [ACSa]. This algorithm takes an array of n consecutive records stored at the lowest possible level and passes them through the base memory level in order, using time $O(n\log\log n)$ for $0 < \alpha < 1$. As it turns out, all the data structures are processed in order throughout the algorithm due to the sorted runs of size N/G created by finding the partitioning elements. The effect of this change is that we get the same recurrence as for the P-HMM model, using an effective cost function $f(x) = \log\log x$. It turns out that the limiting step in the algorithm for P-BT is the need for repositioning the buckets, which can be done using the cited algorithm in time $O\left((N/H)(\log\log(N/H))^4\right)$. So the overall recurrence is

$$T(N) = GT\left(\frac{N}{G}\right) + \sum_b T(N_b) + O\left(\frac{N}{H}\left(\left(\log\log\frac{N}{H}\right)^4 + \frac{GS}{H}\log\log\frac{S}{H} + \log H\right)\right). \tag{5}$$

for the case $N > 12H$. The case $N < 12H$ is the same as in (2).

Lemma 14 *The given algorithm sorts in the P-BT model using time*

$$T(N) = \frac{N}{H}\log N.$$

Proof: Substituting the solution above into Equation (5) demonstrates that it is a solution. □

We have now completed the proof for Theorem 2.

4.5 Sorting in P-UMH and relatives

Vitter and Nodine gave upper and lower bound results for P-UMH, P-RUMH, and P-SUMH [ViN], some of which are were based on simulating the randomized P-HMM and P-BT algorithms given in [ViSa]. The results of the last two subsections can be used to give deterministic algorithms in our model for all those cases that previously required randomization, leading to the following theorem:

Theorem 7 *Distribution sort algorithms can be scheduled on P-UMH and P-RUMH with the following running times.*

$$\begin{aligned}
&\Theta\left(\frac{N}{P}\log N\right) && \text{if } b(\ell) = 1; \\
&O\left(\frac{N}{P}\log N \log\left(\frac{\log N}{\log P}\right)\right) && \text{if } b(\ell) = \frac{1}{\ell+1}; \\
&\Theta\left(\left(\frac{N}{P}\right)^{1+c/2} + \frac{N}{P}\log N\right) && \text{if } b(\ell) = \rho^{-c\ell},\ c > 0.
\end{aligned}$$

The bound given for $b(\ell) = 1/(\ell+1)$ is also a lower bound in P-RUMH.

The following bounds are matching upper and lower bounds for sorting in P-SUMH.

$$\begin{aligned}
&\Theta\left(\frac{N}{P}\log N \log\left(\frac{\log N}{\log P}\right)\right) && \text{if } b(\ell) = 1; \\
&\Theta\left(\frac{N}{P}\log N \log\frac{N}{P}\right) && \text{if } b(\ell) = \frac{1}{\ell+1}; \\
&\Theta\left(\left(\frac{N}{P}\right)^{1+c/2} + \frac{N}{P}\log N\right) && \text{if } b(\ell) = \rho^{-c\ell},\ c > 0.
\end{aligned}$$

In all these models, the upper bounds for $b(\ell) = 1$ and $b(\ell) = 1/(\ell+1)$ are given by a deterministic algorithm based on Balance Sort.

The disadvantage of the deterministic algorithms with respect to the randomized ones is that the former require a CRCW-PRAM to interconnect the processors, whereas the latter will work with a hypercube or cube-connected-cycles interconnection.

5 Parallel Disks with Parallel Processors

In this section, we present a version of Balance Sort for the parallel disk model. The algorithm is optimal in terms of parallel disk I/Os and will also be optimal in the internal processing time when more than one processor is present, up to $P = M\log(M/B)/\log M$.

Algorithm 8 [Balance_Disks]

for each memoryload
(1) read memoryload
(2) assign all M records to buckets
(3) group buckets together (sort using bucket number as key)
(4) **for** each consecutive D blocks of values **in parallel**
update X
$A :=$ ComputeAux(X)
Rebalance(A, X)
(5) write memoryload

The algorithm for the parallel disk model is the same as that used for the P-HMM model, except for a few small changes. In Algorithm 1, we use $N \leq M$ rather than $N \leq H$ as the termination condition for the recursion. We use the Balance_Disks algorithm given in Algorithm 8 instead of the Balance routine given in Algorithm 3. Note that the parameter we call D (the number of disks) is equivalent to the parameter we called H for parallel memory hierarchies. We will likewise use partial striping to group D/D' disks together to make D' virtual disks, with $D' = \sqrt{D}/\log D$. We also use a different method for computing the partitioning elements, described in [ViSa]. Finally, we will select

$$S = \left(\frac{M}{B}\right)^{\gamma}.$$

The procedure for finding the partitioning elements uses as a subroutine Algorithm 9, 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. Algorithm 10 gives the algorithm for finding the partitioning elements.

We start by showing the correctness of the overall sorting algorithm. The general proof of correctness for distribution sort applies, except that we must show that we can group into virtual blocks on our virtual hierarchies. If this grouping is not possible, the matching will not correctly distribute records into buckets, and the sorting algorithm will fail. We need to achieve a virtual blocking with $D/D' = \sqrt{D}\log D$ blocks per virtual block so that all the records in the virtual block fall in the same bucket. When we read in a memoryload, we have M records in memory, and an average of M/S records in each bucket. Thus, each bucket fills an average of $M/(SB)$ blocks. We need to ensure that $M/(SB) \geq \sqrt{D}\log D$. But

$$\frac{M}{SB} = \left(\frac{M}{B}\right)^{1-\gamma} \geq D^{1-\gamma} > \sqrt{D}\log D.$$

Here we have used the fact that $M \geq DB$. If we choose $\gamma = 1/4$, then we find that for any value of D, $D^{1-\gamma} \geq k\sqrt{D}\log D$, where $k = (e\log 2)/4$, so we have an additional

Algorithm 9 [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 $k \leq k_1$
    Select(S_1, k)
else
    Select$(S_2, k - k_1)$

Algorithm 10 [ComputePartitionElements(S)]

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 $(S-1)/2$th record
    Write $\mathcal{M}'_i$
$\mathcal{M}' := \mathcal{M}'_1 \cup \cdots \cup \mathcal{M}'_{N/M}$
for $j := 1$ to $S - 1$
    $e_j :=$ Select$(\mathcal{M}', 2Nj/(S-1)^2)$

factor of no more than $4/(e \log 2) \approx 2.123$ due to internal fragmentation from having blocks that are not quite full on average.

From the standpoint of showing I/O and internal processing bounds, we need to show that the routine for computing the partition elements produces buckets that have nearly the same size. We re-apply Lemma 1, except this time we have $p = (S-1)/2$. Substituting in G and S, it results in a slop that is less than $(M/B)^\gamma N/2M$. The ratio between the slop and the average number of elements per bucket is

$$\frac{1}{2M}\left(\frac{M}{B}\right)^{2\gamma} < \frac{1}{2}$$

for any $\gamma > 0$. Thus, we have for any bucket b,

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

The I/O bound is easy to show for the new algorithm. A, X, L, and E all reside in the internal memory, so there is no cost to access them. We have shown that these data structures will fit in the internal memory in a previous report.

A "memoryload" is the collection of $O(M)$ records that fit into memory. The only place that I/Os are done in the disk version of the algorithm is in lines (1) and (5) of Algorithm 8 and in computing the partition elements. We can read and write a memoryload using full parallelism using $O(M/DB)$ I/Os. Since there are overall $\lceil N/M \rceil$ memoryloads, we find that the number of I/Os done per call to Balance_Disks is $O(N/DB)$. Vitter and Shriver showed that the partition elements can also be computed using $O(N/DB)$ I/Os [ViSa]. Thus, we get the recurrence

$$T(N) = \begin{cases} ST\left(\frac{N}{S}\right) + O(N/DB) & \text{if } N > M \\ 0 & \text{if } N \leq M \end{cases},$$

which has solution

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

This is the same bound as was shown to be optimal for the parallel disk model [AgV].

We devote the remainder of this section to showing that the algorithm operates with optimal internal processing time as long as the number of processors $P \leq M \log \min\{M/B, \log M\} / \log M$. The optimal internal processing time (assuming comparison-based sorting) is $\Omega((N/P) \log N)$. The crux of the matter is in showing that we can use the given number of processors efficiently in all aspects of the sorting algorithm.

The algorithm for finding the partition elements does a total of $O(N/DB)$ I/Os. Since the data are always processed in terms of memoryloads, this corresponds to $O(N/M)$ memoryloads. Each of the memoryloads can be processed in time $O((M/P) \log M)$ for any number of processors $P \leq M$, giving a total time $O((N/P) \log M) = O((N/P) \log N)$.

The remainder of the internal computation time occurs in the routine Balance_Disks. We assume that the CPUs are inactive during I/O, so there is no CPU time charged during steps (1) and (5) of Algorithm 8.

Steps (2) and (3) of Algorithm 8 are easy to analyze in terms of processing time needed per memoryload. The next two lemmas state these bounds.

Lemma 15 *Step (2) of Algorithm 8 can be done in time $O((M/P) \log(M/B))$ for each memoryload for any $P \leq M$.*

Proof: Assign M/P records to each processor. Each processor can then do a binary search on the $(M/B)^\gamma$ partition elements for each of its records, giving a total time of $O((M/P)\log(M/B))$. □

Lemma 16 *Step (3) of Algorithm 8 can be done in time $O((M/P)\log(M/B))$ for each memoryload if $P \leq M \log\min\{M/B, \log M\}/\log M$ or $\log M = O(\log M/B)$.*

Proof: To group the records within each bucket together, we sort the records in internal memory. If $\log M = O(\log M/B)$ then we can sort the M records using Cole's parallel merge sort algorithm [Col] in time $(M/P)\log M = O((M/P)\log(M/B))$ for any $P \leq M$.

Otherwise, instead of sorting using the keys contained in the records, we sort using the bucket number as the key. We will start by considering only $P \geq M/\log M$. We will make use of a deterministic algorithm by Rajasekaran and Reif that appears as Lemma 3.1 in [RaR]. Their algorithm sorts n elements in the range $1, \ldots, \log n$ in time $O(\log n)$ using $n/\log n$ processors. We need to consider two cases:

> *Case 1:* $M/B \leq \log M$. In this case, we let $n = M$. The key values go from 1 to $(M/B)^\gamma < M/B \leq \log M$ so we can apply the Rajasekaran-Reif algorithm directly to sort in time $\log M$ with $M/\log M$ processors, which we have. But $\log M \leq (M/P)\log(M/B)$ as long as $P \leq M\log(M/B)/\log M \leq M \log\log M/\log M$.
>
> *Case 2:* $M/B > \log M$. We will still let $n = M$, but this time $\sqrt{M/B}$ may be larger than $\log M$. What we do is a radix sort using the Rajasekaran-Reif algorithm as a subroutine, sorting $\log M$ bits at a time. Thus, we need $O(\log(M/B)/\log\log M)$ applications of the algorithm, for time
>
> $$O\left(\log M \frac{\log(M/B)}{\log\log M}\right) = O\left(\frac{M}{P}\log\frac{M}{B}\right)$$
>
> whenever
>
> $$P < \frac{M\log\log M}{\log M} < M\frac{\log(M/B)}{\log M}$$
>
> since $\log M < M/B$.

If $P < M/\log M$, then we can attain the same bound by multitasking $P' = M/\log M$ virtual processors onto the P real processors. □

In order to show that loop (4) of Algorithm 8 can be parallelized efficiently, we need to understand more precisely what we mean by "**in parallel**" on line (4). We are dividing the memoryload into tracks, where each track consists of D consecutive blocks of values. We make the observation that if any track consists entirely of elements from the same bucket, it is not necessary for that track to update X or recompute the portion of A that will need to be updated by changing X for that

bucket. We assume that A is updated incrementally, only modifying those rows that were actually affected by the given track. We thus arrive at the following crucial lemma.

Lemma 17 *At most two tracks need to access values of any specific row of A or X during a memoryload.*

Proof: We use here the fact that the records have been sorted according to their bucket numbers. So any pair of consecutive tracks can have only one bucket in common. Thus, if we ignore any track that consists entirely of elements from the same bucket, the lemma is established. □

So we can parallelize this operation by casting out any tracks that consist of all the same bucket, renumbering the tracks, and then doing all the odd-numbered tracks followed by all the even-numbered tracks. The odd/even tracks are guaranteed not to interfere with one another since they access disjoint rows of A and X. This crucial fact leads to the following lemma.

Lemma 18 *Loop (4) of Algorithm 8 can be done in time $O((M/P)\log(M/B))$ for any $P \leq M$.*

Proof: As shown in Lemma 17, at most two tracks need to access any row of X or A. We will have two repetitions of at most $M/2DB$ tracks. We need to update an element of X for each of the M/BD' virtual blocks, which can be done completely in parallel for any number of processors up to M/BD'. The total amount of time updating X is

$$T_X = \begin{cases} \dfrac{M \log D}{PB\sqrt{D}} & \text{if } P \leq M \log DB\sqrt{D} \\ 1 & \text{if } P > M \log DB\sqrt{D} \end{cases}$$

In both cases, the time is less than $(M/P)\log(M/B)$ for any $P \leq M$.

We can do the ComputeAux routine by these two steps.

1. Sort S sets of D' numbers in parallel to pick the $\lceil 2D'/3 \rceil$th smallest element.
2. Update all the SD' elements of A.

The sorts take time

$$T_{A1} = \begin{cases} \dfrac{1}{P}\left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \log\left(\dfrac{\sqrt{D}}{\log D}\right) & \text{if } P \leq \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \\ \log(\sqrt{D}/\log D) & \text{if } P > \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \end{cases}$$

In both cases, the time is no more than $(M/P)\log(M/B)$ for any $P \leq M$. The update time is

$$T_{A2} = \begin{cases} \frac{1}{P}\left(\frac{M}{B}\right)^{\gamma}\frac{\sqrt{D}}{\log D} & \text{if } P \leq \left(\frac{M}{B}\right)^{\gamma}\frac{\sqrt{D}}{\log D} \\ 1 & \text{if } P > \left(\frac{M}{B}\right)^{\gamma}\frac{\sqrt{D}}{\log D} \end{cases}$$

Again, the time is no more than $(M/P)\log(M/B)$ for any $P \leq M$ if $M \geq D^{2/3}/(B^{1/3}\log^{4/3} D)$, which is true since $D^{2/3}/\log^{4/3} D < D$ for all $D \geq 2$.

Finally, we can accomplish the rebalancing by the following steps:

1. Set up M/BD matching problems, each of which is $D' \times D'$.
2. Solve the M/BD matching problems.
3. Do M/BD sets of swaps.

The set-up time can be done in parallel time

$$T_{R1} = \begin{cases} \frac{M}{PB\log^2 D} & \text{if } P \leq \frac{M}{B\log^2 D} \\ 1 & \text{if } P > \frac{M}{B\log^2 D} \end{cases}$$

Each matching problem can use up to D processors to achieve time $\log D$. The matching takes time

$$T_{R2} = \begin{cases} \frac{M\log D}{PBD} & \text{if } P \leq \frac{M}{B} \\ \log D & \text{if } P > \frac{M}{B} \end{cases}$$

Both of these steps take time no more than $(M/P)\log(M/B)$ for any $P \leq M$. Finally, the swaps involve moving up to M items, but no internal processing time other than touching them, for a total time of $O(M/P) \leq (M/P)\log(M/B)$ for any $P \leq M$.

We have shown that each individual step meets the time bound proposed by the lemma, establishing its validity. □

Finally, we are ready to prove our main result for disk sorting, Theorem 3.

Proof: We established the I/O bound above. To show that the CPU time is optimal, we need to show that it takes a total of $O((N/P)\log N)$ time. We have shown that each memoryload can be processed using time $O((M/P)\log(M/B))$ whenever either $P \leq M\log\min\{M/B, \log M\}/\log M$ or $\log M = O(\log M/B)$. The number of memoryloads we need to process the file is $O((N/M)\log(N/B)/\log(M/B))$. Therefore, the total time is

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

as desired. □

6 Conclusions

In this report, we have described the first known deterministic algorithm for sorting optimally using parallel hierarchical memories. This algorithm improves upon the randomized algorithms of Vitter and Shriver [ViSa]. The algorithm applies to P-HMM, P-BT, P-UMH, P-RUMH, and P-SUMH, as well as the parallel disk model. The parallel disk version of the algorithm also improves upon our previously published algorithms [NoVa, ViN] in the following ways:

1. The hidden constants in the big-O 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.

2. 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 error checking and redundancy information can be maintained 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 the Greed Sort algorithm to use only striped writes, again because of the embedded call to Columnsort.

3. The new algorithm can be made to work optimally in terms of internal processing time even in the presence of large degrees of processor parallelism.

The primary disadvantage of the algorithm presented in this report is that it requires that the CPUs be connected to the memories by a CRCW-PRAM. The CRCW-PRAM is required only in the log-time matching algorithm, so any improvement there will automatically improve the overall algorithm. Ideally, one would like to be able to do the matching in $O(\log H)$ time on a hypercube or other actual parallel processor topology.

Although we have presented a deterministic algorithm, in practice the randomized algorithm resulting from doing randomized matching and not requiring any partial hierarchy striping would be simpler to implement.

7 References

[AAC] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, "A Model for Hierarchical Memory," *Proceedings of 19th Annual ACM Symposium on Theory of Computing* (May 1987), 305–314.

[ACSa] A. Aggarwal, A. Chandra, and M. Snir, "Hierarchical Memory with Block Transfer," *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science* (October 1987), 204–216.

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

[ACF] B. Alpern, L. Carter, and E. Feig, "Uniform Memory Hierarchies," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (October 1990), 600–608.

[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.

[Col] Richard Cole, "Parallel Merge Sort," *SIAM J. Computing* 17 (August 1988), 770–785.

[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.

[GoS] Mark Goldberg and Thomas Spencer, "Constructing a Maximal Independent Set in Parallel," *SIAM J. Discrete Math* 2, 322–328.

[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.

[Luba] Michael Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem," *SIAM J. Computing* 15 (1986), 1036–1053.

[Lubb] Michael G. Luby, "Removing Randomness in a Parallel Computation Without a Processor Penalty," International Computer Science Institute, TR-89-044, July 1989.

[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 Disks," Department of Computer Science, Brown University, CS-92-08, 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.

[RaR] S. Rajasekaran and J. Reif, "Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms," *SIAM J. Computing* 18 (1989), 594–607.

[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 II: Hierarchical Multilevel Memories," Brown University, CS-90-22, September 1990.

[ViN] J.S. Vitter and Mark H. Nodine, "Large-Scale Sorting in Uniform Memory Hierarchies," *Journal of Parallel and Distributed Computing* (January 1993), 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.