Optimal Deterministic Sorting on Parallel Processors and Parallel Memory Hierarchies *

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 practical deterministic load balancing strategy for distribution sort that is applicable to parallel disks and parallel memory hierarchies with both single and parallel processors. The simplest application of the strategy is an optimal deterministic algorithm called Balance Sort for external sorting on multiple disks with a single CPU, as described in the companion paper. However, the internal processing of Balance Sort does not seem parallelizable. In this paper, we develop an elegant variation that achieves full parallel speedup. The algorithms so derived are optimal for all parallel memory hierarchies with any type of a PRAM base-level interconnection and are either optimal or best-known for a hypercube interconnection. We show how to achieve optimal internal processing time as well as optimal number of I/Os in parallel two-level memories.

Keywords: Memory hierarchies, 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].

†Part of this research was done while the author was at Brown University, 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.

‡Part of this research was done while the author was at Brown University. 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

Large-scale computer systems contain many levels of memory—ranging from very small but very fast registers, to successively larger but slower memories, such as multiple levels of cache, primary memory, magnetic disks, and archival storage. It is often the case that the overall performance of an algorithm depends more on the characteristics of the memory hierarchy than upon those of the CPU, since the limiting factor in CPU utilization may be the ability of the memory hierarchy to supply the processor with the needed data. In this paper, we investigate the capabilities of *parallel* memory hierarchies for the sorting problem.

Conceptually, the simplest large-scale parallel memory is the two-level memory with multiple disks, known as the parallel disk model [ViSa]. Figure **??**a shows the uniprocessor ($P = 1$) parallel disk model with $D > 1$ disks. A more general model we consider in this paper, in which the internal processing is done on $P \geq 1$ interconnected processors, is shown in Figure **??**b for the special case $P = D$. The interconnections we consider are the hypercube and the Exclusive-Read/Exclusive-Write (EREW) PRAM.

In a single I/O, each of the D disks can simultaneously transfer a block of B records. Our two measures of performance are the number of I/Os and the amount of internal processing done. The Balance Sort algorithm of the companion paper [NoVa] is optimal in terms of the number of I/Os, but does not allow fast internal processing in parallel.

We also consider parallel multilevel hierarchical memory models, based on three sequential hierarchical models. The Hierarchical Memory Model (HMM) proposed by Aggarwal *et al.* [AAC] is depicted in Figure **??**a. In the $\mathrm{HMM}_{f(x)}$ model, access to memory location x takes $f(x)$ time. Figure **??**a suggests the $\mathrm{HMM}_{\lceil \log x \rceil}$ model, where each layer in the hierarchy is twice as large as 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 to access. We consider all "well-behaved" cost functions $f(x)$, by which we mean that $f(2x) \leq c\, f(x)$ for all x. Typical examples are $f(x) = \log x$ and $f(x) = x^{\alpha}$, for $\alpha > 0$.[1]

An elaboration of HMM is the Block Transfer (BT) model of Aggarwal *et al.* [ACSa], depicted schematically in Figure **??**b. Like HMM, it has a cost function $f(x)$ to access a single record, but in addition it incorporates the notion of block transfer by allowing access to the previous ℓ records in unit time per record; that is, the $\ell + 1$ locations $x, x-1, \ldots, x-\ell$ can be accessed at cost $f(x) + \ell$. In Figure **??**b, the $f(x)$ cost corresponds to injecting the "needle"; pushing in or pulling out additional items once the first item has been pushed or pulled takes only one time unit per item.

An alternative block-oriented memory hierarchy is the Uniform Memory Hierarchy (UMH) of Alpern *et al.* [ACF], depicted in Figure **??**c. 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

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

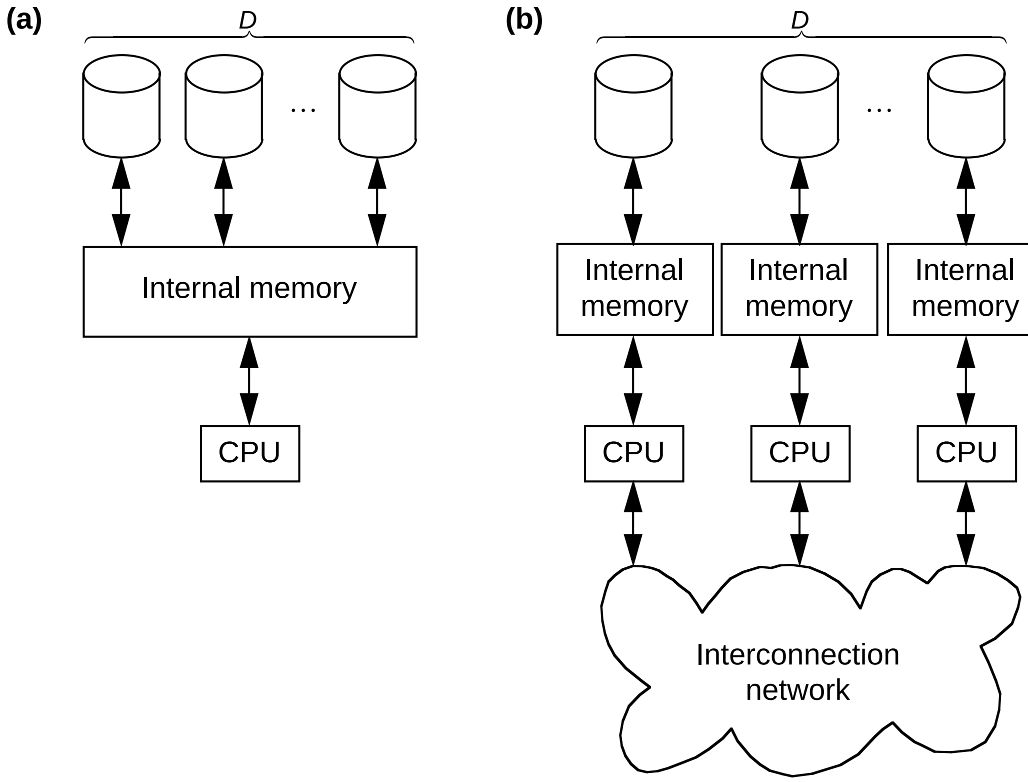

Figure 1: (a) The parallel disk model. Each of the D disks can simultaneously transfer B records to and from internal memory in a single I/O. The internal memory can store $M \geq 4DB$ records. (b) Multiprocessor generalization of the I/O model in (a), in which each of the $P = D$ internal processors controls one disk and has an internal memory of size M/P. The P processors are connected by some interconnection network topology such as a hypercube or an EREW PRAM and their memories collectively have size M.

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 Sequential UMH or SUMH model. A less restrictive model that we proposed earlier [ViN] and which corresponds closely to currently used programming constructs, is the Restricted UMH or 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, as shown in Figure **??**. The base memory levels of the H hierarchies are attached to H interconnected processors. We assume that the hierarchies are all of the same kind and all have the same parameters. The parallel hierarchical models for HMM, BT, UMH, SUMH, and RUMH, are respectively denoted as P-HMM, P-BT, P-UMH, P-SUMH, and P-RUMH.

In this paper, we present practical and optimal deterministic algorithms for sorting on parallel multilevel memory hierarchies. We use an elegant modification of the Balance Sort

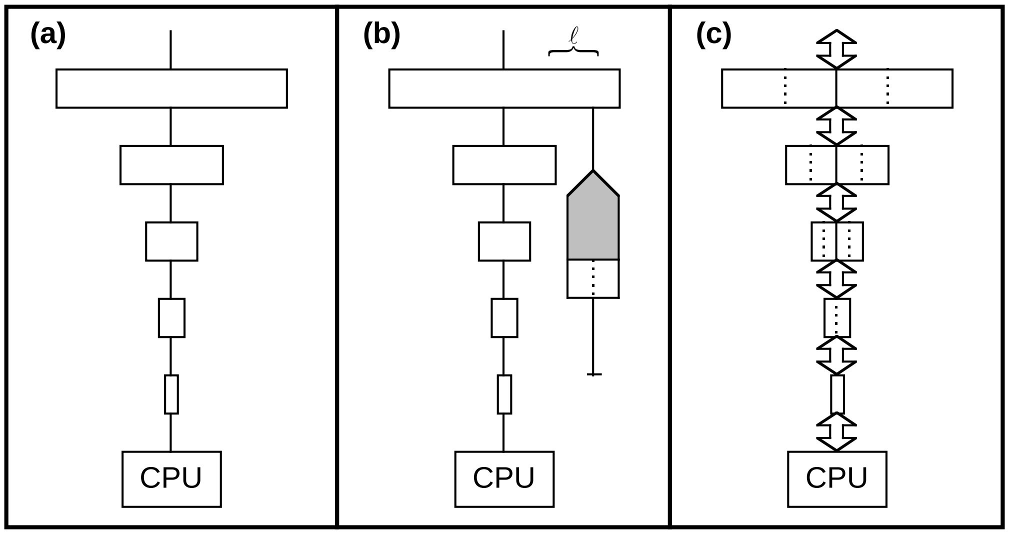

Figure 2: Multilevel hierarchy models. (a) The HMM model. (b) The BT model. (c) The UMH model.

approach in the companion paper [NoVa], which may in fact be preferable to the original Balance Sort on the multiple-disk two-level model. Our main results are in the next section. We give an optimal algorithm for sorting on $D \geq 1$ disks and $P \geq 1$ processors. Optimality is in terms of both the number of I/Os and the internal processing time. We also give optimal sorting algorithms for the P-HMM and P-BT parallel memory hierarchies. The same technique can also be applied to the randomized algorithms for P-UMH, P-SUMH and P-RUMH [ViN] to get corresponding deterministic algorithms. In Section **??**, we give an algorithm that is optimal for all the parallel multi-level hierarchies. In Section **??**, we show how to alter the algorithm to deal with parallelism of CPUs in the parallel disk model. The resultant algorithm is very practical and is recommended as the sorting method of choice for parallel disks. Our conclusions are in Section **??**.

2 Main Results

In this section, we present our main results. The elegant load balancing approach 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 of [ViSa], [ViSb] and [ViN]. We also improve upon the deterministic Greed Sort algorithm in [NoVb], which is known to be optimal only for the parallel disk models and not for hierarchical memories. Greed Sort requires both independent reads and writes, whereas this algorithm works with striped writes, which is advantageous for error correction. The lower bounds are proved in [ViSb] (Theorems **??** and **??**) and [AgV] (Theorem **??**).

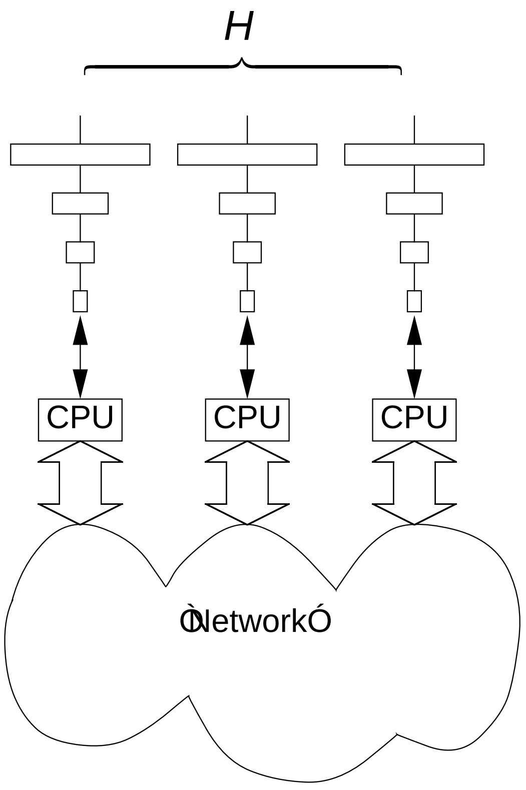

Figure 3: Parallel multilevel memory hierarchies. The H hierarchies (of any of the types listed in Figure **??**) have their base levels connected by H interconnected processors.

Theorem 1 *In the P-HMM model with an EREW PRAM interconnection, the time $T(N)$ for sorting N elements is*

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

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

On a hypercube interconnection, the P-HMM time $T(N)$ for sorting N elements is

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

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

where $T(H)=O(\log H\,(\log\log H)^2)$ is the time needed to sort H items on an H-processor hypercube. The upper bounds are given by a deterministic algorithm based on Balance Sort. The lower bounds for the PRAM interconnection hold for any type of PRAM. The lower bounds for the $f(x)=x^\alpha$ case require the comparison model of computation.

The current best deterministic bound for $T(H)$ on a hypercube comes from [CyP]. In the hypercube case for $f(x)=\log x$, the term in the sorting time involving $T(H)$ is thus possibly nonoptimal by an $O((\log\log H)^2)$ factor; however, the algorithm is optimal when N is large ($N > H^{(\log H)^{\log\log H}}$) or small ($N = O(H)$). Note that if we are allowed to use a randomized sorting algorithm as a subroutine to sort in time $T(H) = \log H$, then our algorithm is optimal for hypercubes.

Theorem 2 *In the P-BT model with an EREW PRAM, the time $T(N)$ for sorting N elements is*

$$\begin{array}{ll} \Theta\left(\frac{N}{H}\log N\right) & \text{if } f(x) = \log x; \\[2ex] \Theta\left(\frac{N}{H}\log N\right) & \text{if } f(x) = x^\alpha,\ \ 0 < \alpha < 1; \\[2ex] \Theta\left(\frac{N}{H}\left(\log^2\frac{N}{H} + \log N\right)\right) & \text{if } f(x) = x^\alpha,\ \ \alpha = 1; \\[2ex] \Theta\left(\left(\frac{N}{H}\right)^\alpha + \frac{N}{H}\log N\right) & \text{if } f(x) = x^\alpha,\ \ \alpha > 1. \end{array}$$

The corresponding bounds $T(N)$ for a hypercube interconnection of the P-BT memory hierarchies are

$$\begin{array}{ll} O\left(\frac{N\log N}{H\log H}T(H)\right) & \text{if } f(x) = \log x; \\[2ex] O\left(\frac{N\log N}{H\log H}T(H)\right) & \text{if } f(x) = x^\alpha,\ \ 0 < \alpha < 1; \\[2ex] \Theta\left(\frac{N}{H}\left(\log^2\frac{N}{H} + \frac{\log N}{\log H}T(H)\right)\right) & \text{if } f(x) = x^\alpha,\ \ \alpha = 1; \\[2ex] \Theta\left(\left(\frac{N}{H}\right)^\alpha + \frac{N\log N}{H\log H}T(H)\right) & \text{if } f(x) = x^\alpha,\ \ \alpha > 1, \end{array}$$

where $T(H) = O(\log H\,(\log\log H)^2)$ is the time needed to sort H items on an H-processor hypercube. The upper bounds are given by a deterministic algorithm based on Balance Sort. The lower bounds for the PRAM interconnection hold for any type of PRAM. 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 with our PRAM interconnection. In this paper, however, we concentrate on the P-HMM and P-BT models.

Theorem 3 *The number of I/Os needed 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 with an EREW PRAM interconnection when $\log M = O(\log(M/B))$. The same running time can be achieved

Algorithm 1 [Sort(N, T)]

```
        if $N \leq 3H$
            $n := \lceil N/H \rceil$
            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 $\leq 3$ 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 $b$th row of $L$ {set in Balance}
                $N_b :=$ number of elements in bucket $b$
(8)             Sort($N_b, T$)
(9)             Append sorted bucket to output area
```

on a CRCW PRAM interconnection if $P \leq M \log \min\{M/B, \log M\}/\log M$. When the interconnection is a hypercube, the internal processing time is the number of I/Os times the time to partition DB elements among $\sqrt{M/B}$ sorted partition elements on a P-processor hypercube. 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.

3 Parallel Memory Hierarchies

This section describes the deterministic algorithm that serves as the basis for the upper bounds in Theorems **??** and **??**. Subsection **??** gives the overall sorting algorithm. Our sorting algorithm is in the spirit of the Balance Sort algorithm given in the companion paper, but with several important modifications needed to cope with hierarchies and the inherent parallelism at the base level. We put these changes in perspective in Subsection **??** and discuss the reasons and ramifications of the changes. In Subsection **??**, we analyze the algorithm for the P-HMM model. Subsection **??** covers the P-BT model.

3.1 The sorting algorithm

For simplicity we assume without loss of generality that the N keys are distinct; this assumption is easily realizable by appending to each key the record's initial location. Algorithm **??** is the top-level description of our sorting algorithm for parallel memory hierarchies. We also assume without loss of generality 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.[2] The algorithm is a version of *distribution sort* (sometimes called bucket sort). It sorts a set of elements by choosing $S-1$ partitioning elements of approximately evenly-spaced rank in the set and using them to partition the data into S disjoint ranges, or *buckets*. The individual buckets are then sorted recursively and concatenated to form a completely sorted list.

The difficult part of the algorithm is the load balancing done by the routine Balance, which makes sure that the buckets are (approximately) evenly distributed among the hierarchies during partitioning. In order for the balancing to occur in optimal deterministic time, it is necessary to do partial striping of the hierarchies, so that we will have only H' *logical hierarchies* with logical blocks of size $B = H/H'$. We number the logical hierarchies $0, \dots, H'-1$. The value of H' we use is $H' = \sqrt{H}$. We must not use full striping because it will not give optimal results when N becomes less than about H^2. In this case, the optimal algorithm takes only a constant number of further levels of recursion, whereas full striping will take $\Omega(\log\log(N/H))$ levels of recursion.

A number of parameters in each level of the algorithm merit explanation:

N	$=$	# of elements to sort
T	$=$	array of H' elements pointing to the starting block on each logical hierarchy
S	$=$	# buckets = (# of partition elements) + 1
E	$=$	"global" array of $S-1$ partition elements
X	$=$	"global" $S \times H'$ *histogram* matrix (described later)
A	$=$	"global" $S \times H'$ *auxiliary* matrix (described later)
L	$=$	"global" $S \times H'$ *location* matrix (described later)

Some of these parameters 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 passed by reference to the routines that need to access them. We should make a remark about storage of these "global" arrays. We 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 this assumption because of the well-behaved cost function $f(x)$, as defined in Section **??**. 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 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. To get optimal performance, we determine the number S of buckets differently depending upon which hierarchical model we are using. Algorithm **??** gives the routine for computing the $S-1$ partition elements, based on [AAC, ViSb]. It works by recursively sorting sets of size $\lceil N/G \rceil$ and choosing every $\lfloor \log N \rfloor$th element. The approximate partition

[2]This modification does not affect the asymptotic overall running time, since it increases the space requirements by only a constant factor and we have a well-behaved cost function.

Algorithm 2 [ComputePartitionElements(S)]

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

elements are selected from this subset. The specific value of G used in the algorithm is dependent upon which hierarchical memory model is being used. 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. We show the following technical lemma, which leads to an analysis of the effectiveness of the approximate partitioning elements in dividing the records into nearly equal-sized buckets.

Lemma 1 *Assume that all keys are distinct. For any p, if we choose every pth element from the subsets in line (2) of Algorithm* **??**, *the amount of "slop" in the size of the buckets produced is less than pG; that is, for every bucket b, the number N_b of elements in bucket b satisfies*

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

Proof: We collect a set C of $\lfloor N/p \rfloor$ elements by choosing every pth element from each of the G sorted subsets. We then sort C into increasing order. Let c_k denote the kth (smallest) element of C. We set the bth partition element to be $c_{\lfloor bq \rfloor}$, where $q = N/((S-1)p)$. The minimum rank of $c_{\lfloor bq \rfloor}$ in the original set is $\lfloor bq \rfloor p$, since each of the $\lfloor bq \rfloor$ elements in C less than or equal to $c_{\lfloor bq \rfloor}$ represents p elements in the original set. The maximum rank of $c_{\lfloor bq \rfloor}$ in the original set occurs if each of the other $G-1$ subsets has $p-1$ elements not in C that are less than $c_{\lfloor bq \rfloor}$ followed by an element in C greater than $c_{\lfloor bq \rfloor}$. Thus, the amount of "slop" does not exceed $(G-1)(p-1)$. □

Corollary 1 *If we use $p = \lfloor \log N \rfloor$ and choose G such that $G\lfloor \log N \rfloor \leq N/S$, then we get $0 < N_b < 2N/S$ for any bucket b, where N_b is the size of bucket b, so that the buckets are roughly equally sized.*

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 **??** gives the Balance routine for balancing the buckets among the H' logical hierarchies. Balance works as follows, successively track by track: A parallel read is done from the current subset of the G sorted subsets in order to get a full track of records. (Some records may have been left from the previous iteration.) These records are partitioned into buckets by merging them with the partition elements, and the contents of the buckets from the track are formed into logical blocks. Each logical block resides on some logical hierarchy. The logical blocks that

Algorithm 3 [Balance(T)]

```
        $v := H'$
        A := **0**
        **while** there are unprocessed elements
(1)         Read next $v$ logical blocks
(2)         Partition the records into buckets (in parallel)
(3)         Collect buckets into logical blocks of size $H/H'$
                (all elements of any block are from the same bucket)
            { Rebalance writes some logical blocks, returns # of blocks written }
(4)         $v :=$ Rebalance()
(5)         Update the internal pointers of the logical blocks and location matrix $L$
(6)         Collect unprocessed logical blocks to allow room for next $v$ logical blocks in
                the next iteration
```

do not overly unbalance their respective buckets, meaning that they do not introduce a 2 into the auxiliary matrix (described below), are written to higher levels of their respective logical hierarchies. Those logical blocks that do unbalance their buckets are sent to the Rebalance subroutine. As the records are partitioned and distributed, the *histogram matrix* $X = \{x_{bh}\}$ records how the buckets are distributed among the hierarchies; in particular, x_{bh} is the number of logical blocks of bucket b on logical hierarchy h. Updating the histogram matrix on line (4) simply means that if logical hierarchy h is assigned a logical block from bucket b, then we increment x_{bh} by 1.

The *location matrix* $L = \{l_{bh}\}$ tells what location was written last on each logical hierarchy for each bucket. We do not assume that the logical blocks are at the same location on each logical 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. Subsequent tracks are read by following pointers present within each hierarchy; these pointers are set during the previous level of recursion. The first level of recursion reads the tracks in order in a round-robin fashion.

The *auxiliary matrix* $A = \{a_{bh}\}$ determines if the placement becomes too badly skewed. Specifically, if m_b is the median number of logical blocks that bucket b has on all the logical hierarchies (i.e., m_b is the median of $x_{b1}, \ldots, x_{bH'}$),[3] we compute in the ComputeAux routine (Algorithm **??**)

$$a_{bh} := \max\{0, x_{bh} - m_b\}.$$

(ComputeAux 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.) This definition forces the important invariant:

Invariant 1 *At least $\lceil H'/2 \rceil$ entries of every row of the auxiliary matrix A are 0s.*

[3]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 if D is even.

The Balance routine (and its subroutine Rebalance) maintains good balance by guaranteeing that there are at most $\lfloor H'/2 \rfloor$ 1s in the auxiliary matrix A, and that the remaining entries must be 0s. If a block is to be written to a virtual hierarchy for which the corresponding entry in A is 1, it is considered to be "unprocessed" and is held in memory to be processed with the next track. The net result is that the auxiliary matrix A contains only 0s and 1s:

Invariant 2 *After each track is processed, the auxiliary matrix A is binary; that is, all of its entries are either 0 or 1. Hence $x_{bh} \leq m_b + 1$ for all $1 \leq h \leq H'$, where m_b is the median entry on row b of A.*

Invariant 2, coupled with the definition of median, proves the following theorem that the buckets are balanced. The proof is the same as that of Theorem 3 in the companion paper.

Theorem 4 *Any bucket b will take no more than a factor of 2 above the optimal number of tracks to read.*

In order to assign the records to buckets in line (2) of Algorithm **??**, we merge the elements in the base memory level with H consecutive partition elements. We pick as the first of the H partition elements the one that separates the last bucket seen during the preceding track from the one that follows it (for the first track, we use the first H partition elements). The final locations of the partition elements serve to demarcate the buckets.

Collecting the buckets into logical blocks in line (3) of Algorithm **??** is surprisingly easy. We choose a value of G such that the initial sorts of size $\lceil N/G \rceil$ put together within each run an average of at least H/H' records for the S buckets. One possible issue is that there may be internal fragmentation due to the fact that the number of elements within a given bucket is likely not an exact multiple of the logical block size. However, we lose no more than a factor of 2 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 of B elements 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 not exceed

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

□

We will apply this lemma using $k = S$ and $B = H/H'$. Because of internal fragmentation, processing v logical input blocks results in processing up to $2v$ logical blocks all containing the same value; lines (4) to (6) are executed once for each set of v logical blocks. We will not consider this issue further for hierarchies as it results in at most a factor of two in the running time.

Algorithm 4 [ComputeAux(X)]

```
for $b := 1$ to $S$
    $m_b :=$ median ($\lfloor H'/2 \rfloor$th smallest) element of $x_{b1}, \ldots, x_{bH'}$
    for $h := 1$ to $H'$ in parallel
        $a_{bh} := \max\{0, x_{bh} - m_b\}$
```

Algorithm 5 [Rebalance returns number of logical blocks written v]

```
        { Create a matching matrix $M$ }
        for $h := 0$ to $H' - 1$ in parallel
            $b :=$ the bucket on logical hierarchy $h$
(1)         $b_h := b$
            for $h' := 0$ to $H' - 1$ in parallel
(2)             $M[h, h'] := a_{bh'}$

        { Find a cyclic shift that minimizes the diagonal sum. }
        for $h := 0$ to $H' - 1$ in parallel
(3)         $s_h = 0$
            for $h' := 0$ to $H' - 1$ in parallel
                $h'' = (h + h') \bmod H'$
(4)             $s_h := s_h + M[h', h'']$
(5)     $h :=$ value of $h'$ for which $s_{h'}$ is minimized
(6)     route logical blocks using cyclic shift of the logical hierarchies
        for $h' := 0$ to $H' - 1$ in parallel
            $h'' = (h + h') \bmod H'$
(7)         if $M[h', h''] = 0$
(8)             write the logical block to logical hierarchy $h''$
(9)             $b := b_{h'}$
(10)            $x_{bh''} := x_{bh''} + 1$
(11)    $A =$ ComputeAux($X$)
        return $H' - s_h$
```

After the rebalancing, Step (6) of Algorithm **??** routes any unprocessed logical blocks into a contiguous region that does not overlap with any of the next v logical hierarchies to be read. This operation takes time $O(\log H)$ by monotone routing [Lei, Section 3.4.3].

Algorithm **??** assumes that it always has v logical blocks available (until the end), 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 logical 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.

Algorithm **??** gives a Rebalance subroutine significantly different from the sequential-based balance technique in the companion paper. The Rebalance subroutine is based on the simple observation that if we only write a logical block of bucket b to a logical hierarchy h for which $a_{bh} = 0$, then the auxiliary matrix A remains a binary matrix, i.e., a matrix in which all elements are 0 or 1. We set up a mapping that writes at least half the logical blocks simultaneously in a single parallel memory reference to logical hierarchies for which A has a 0 for that block.

The mapping used by subroutine Rebalance is computed by considering $H'-1$ very simple potential mappings (specifically, cyclic shifts) and choosing on Line (5) the one that works the best. The matrix M allows for computing the potential mappings and is constructed as

$$m_{hh'} = a_{bh'},$$

where b is the bucket represented by the block read from logical hierarchy h. Figure **??** gives an example of the balancing process. In this figure, $H' = 4$ and we are processing Track t on the "Before" side of the diagram. We read in blocks that must be assigned to Buckets 1, 2, 3, and 2, respectively. The figure shows the relevant portion of the histogram matrix X and the auxiliary matrix A. We form matching matrix M by taking rows from the auxiliary matrix for the buckets read from each virtual hierarchy (depicted as a "disk" in the figure) in track t. We look for a diagonal in M with a minimal sum; one such diagonal is marked. We rearrange the blocks according to the selected matching of M and move to the "After" side of the diagram. We write each block whose entry in the matching matrix M was 0, updating X and A to reflect the new counts. The block containing Bucket 1 that was read from Disk 1 has a non-zero entry in the matching matrix M; accordingly, we consider that block to be unprocessed this round and do not write it to the output when we write the other blocks. The unprocessed block will be considered to be part of the next track (in other words, we will only read the next three blocks for Track $t+1$).

Line (4) of Algorithm **??**, in which the value of s_h is computed as a linear sum, can be computed using a parallel prefix operation in $O(\log H')$ time using H' processors. Since $H = O\left((H')^2\right)$, we can compute all the s_h values simultaneously in $O(\log H)$ time using $H' \times H' = H$ processors. Similarly, the value of h on Line (5) of Algorithm **??** can be computed in $O(\log H)$ time using parallel prefix. Line (6) of Algorithm **??** is a cyclic shift and is accomplished in $O(\log H)$ time by two monotone routing operations, the first of which handles those records that do not "wrap around" and the second those that do "wrap around". Line (8) is a parallel memory reference.

In the following lemmas, we prove that Algorithm **??** leaves A as a binary matrix and that it writes at least half of the H' virtual blocks.

Lemma 3 *Assume the auxiliary matrix A is binary. Suppose we write a logical block corresponding to a bucket b that has $a_{bh} = 0$ to logical hierarchy h. If we update X to reflect the change and call ComputeAux to compute the new auxiliary matrix A' after the operation, then the resulting A' is binary.*

Proof: Since a_{bh} was 0 and x_{bh} is incremented by only 1, a'_{bh} can be at most 1. No other element of A' can be larger than its corresponding element of A, although some elements in row b may be smaller if incrementing x_{bh} causes it to become the new median element of

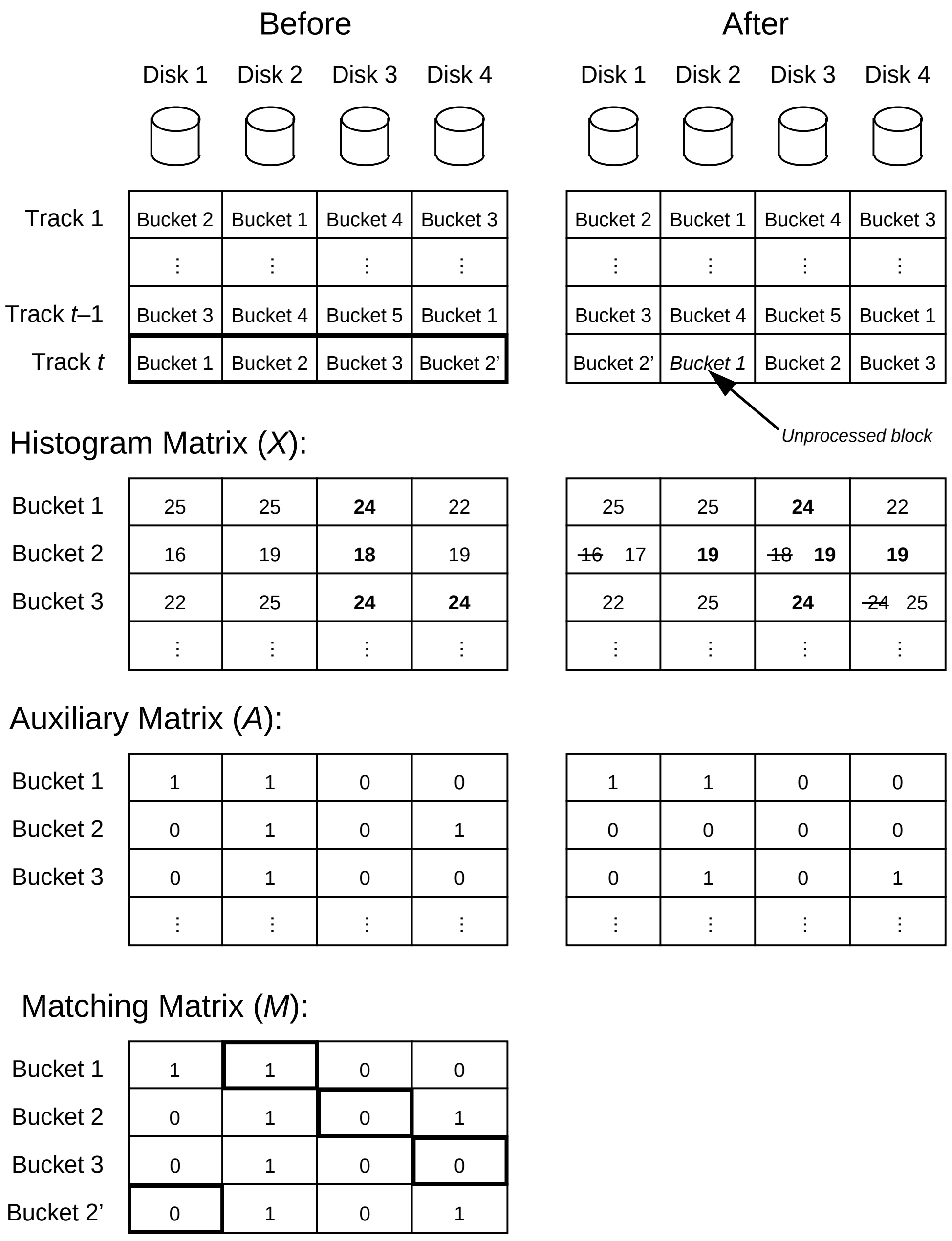

Figure 4: An example of balancing blocks for Track t.

row b. Since all elements of A were assumed to be binary and since ComputeAux can never produce negative values, A' must also be binary. □

Lemma 4 *Assume that Rebalance is called while auxiliary matrix A is binary. Then Rebalance writes at least $\lceil H'/2 \rceil$ blocks.*

Proof: This lemma is equivalent to saying that for the value of h chosen on Line (5) of Algorithm **??**, at least $\lceil H'/2 \rceil$ values of the set $\{M[h', (h + h') \bmod H'] \mid 0 \leq h' \leq H' - 1\}$ are 0. By Invariant 1, at least $\lceil H'/2 \rceil$ of the entries in each row of M are 0. We consider H' possible mappings. The average number $\overline{z}$ of 0s among the mappings is at least $\lceil H'/2 \rceil$ since the mappings use each element of M once. The best mapping must have a number z of 0s that is at least the average $\overline{z}$; since z must be an integer, it follows that $z \geq \lceil H'/2 \rceil$. Since one logical block is written for each 0 in the matching, Rebalance writes at least $\lceil H'/2 \rceil$ blocks. □

3.2 Differences from Balance Sort

Before we plunge into the analysis of the modified Balance Sort algorithm on the various hierarchies, we want to point out the important ways in which the Balance Sort algorithm of the companion paper needed to be changed:

1. The Balance routine no longer does a min-cost matching to specify an initial placement of the blocks. No initial matching is done at all. All of the work in making sure that the records are balanced evenly among the hierarchies is now done by the subroutine Rebalance. The min-cost matching was removed since it was simply too costly to use it in parallel.

2. The Rebalance subroutine now overlaps the input and output hierarchies in each I/O. In Balance Sort, the "block of 1s" ensures, in terms of the auxiliary matrix A, that no bucket that has a 2 on some disk can have a 0 on any other disk that has a 2 on it; consequently the disks can be safely partitioned into those that have 2s and those that potentially have 0s. Since there is no "block of 1s" in the modified Balance routine, any hierarchy except the one on which the 2 occurs is a possible candidate for rebalancing.

3. The ComputePartitionElements subroutine is used to accomplish partial striping on the hierarchies. The partial striping makes the number of processors large relative to the size of the matchings so that the Rebalance algorithm can evaluate $O(H')$ potential mappings in $O(\log H')$ time.

3.3 Analysis of P-HMM

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

$$G = \begin{cases} \left\lfloor \dfrac{\sqrt{N}}{\sqrt{2 \log N} H^{1/4}} \right\rfloor & \text{if } 2N \log N > H^{3/2}; \\ \left\lfloor \dfrac{N}{H} \right\rfloor & \text{if } 2N \log N \leq H^{3/2}; \end{cases} \tag{1}$$

$$S = \begin{cases} \left\lfloor \dfrac{\sqrt{2N}}{\sqrt{\log N} H^{1/4}} \right\rfloor & \text{if } 2N \log N > H^{3/2}; \\ \left\lfloor \dfrac{2N}{H} \right\rfloor & \text{if } 2N \log N \leq H^{3/2}. \end{cases} \tag{2}$$

In the following lemmas, we show that the logical blocking can be done and that the buckets are approximately the same size.

Lemma 5 *With the above values of G and S, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least $H/H' = \sqrt{H}$ and at least $\log N$.*

Proof: The average number of elements per bucket per run is N/SG. Since $S \leq \sqrt{2N}/\sqrt{\log N}H^{1/4}$ and $G \leq \sqrt{N}/\sqrt{2\log N}H^{1/4}$, we have $N/SG \geq \sqrt{H}\log N$. □

Corollary 2 *With the above values of G and S, the condition needed for Corollary* **??** *is satisfied and the buckets created are approximately the same size.*

In order to avoid clutter in the recurrences we develop and analyze, we will take some liberties and occasionally neglect the floor or ceiling terms needed to insure that various quantities are integers. The asymptotics of the solutions do not change.

The recurrence for P-HMM is derived in Table **??**. We assume throughout that a data structure of size s has an access cost of $f(s/H)$. The analysis of Algorithm **??** is critically dependent upon the fact that the algorithm works in parallel. In particular, it assumes that the hth columns of A and M and the elements s_h and b_h are all stored on logical hierarchy h, so that their accesses can occur in parallel. In the analysis of Algorithm **??**, $P(H)$ is the amount of time needed to do a parallel prefix operation on H elements and $R_c(H)$ is the time needed for routing a cyclic permutation.

In Algorithm **??**, it is possible to assume that the number of iterations of the loop is $O(N/H)$ because of Theorem **??**. There is a trick used to obtain the running time of assigning records to buckets at line (2) of Algorithm **??**. The records have already been sorted into G sorted runs $\mathcal{G}_i$ of size $\lceil N/G \rceil$ by ComputePartitionElements. To assign the records to buckets, we merge each sorted run $\mathcal{G}_i$ with the S partitioning elements, and hence we use the S partition elements G times. The evaluation of ComputeAux at line (11) of Algorithm **??** can be done incrementally in the given time.

The recurrence from Table **??** reduces to

$$T(N) = G\,T\Big(\frac{N}{G}\Big) + \sum_b T(N_b) + O\Big(\frac{N}{H}\Big(f\Big(\frac{N}{H}\Big) + P(H) + R_c(H)\Big)\Big), \tag{3}$$

for the case $N > 2H$, where $P(H)$ is the time needed to do a parallel prefix operation on H items and $R_c(H)$ is the time needed to route a cyclic permutation. For both the hypercube and any type of PRAM, $P(H) = O(\log H)$. We have $R_c(H) = O(1)$ on a PRAM and $R_c(H) = O(\log H)$ on a hypercube.

Now let us consider the case $N = O(H)$. The following lemma about merging helps establish the sorting bound in this case.

Lemma 6 *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 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 requires two parallel reads, since the elements 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 $O(\log H)$ time by monotone routing.

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 merge the two lists 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 have been 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 $3N/\lfloor H/2 \rfloor$ read/write operations to process the N records, each of which takes time $f(N/H)$. We also need $O(\log H)$ time for merging and shifting between read/write operations, yielding a total time that is $O((N/H)(f(N/H) + \log H))$. □

Using the fact that $T(H) \geq \log H$, we can now establish the base case of the recurrence.

Corollary 3 *When $N = O(H)$, the amount of time needed to sort N records is $O(T(H))$.*

Before we derive the general solution to recurrence (**??**), we need a technical lemma to help us deal with the summation in the recurrence.

Lemma 7 *Let $T(N)$ be a function that grows at least linearly and let $\sum_{1 \leq b \leq S} N_b = N$ and $0 \leq N_b \leq 2N/S$, for $1 \leq b \leq S$. Then*

$$\sum_{1 \leq b \leq S} T(N_b) \leq \frac{S}{2}\left(T\left(\frac{2N}{S}\right) + T(0)\right). \tag{4}$$

Proof: By the convexity of our function, $\sum_b T(N_b)$ is maximized if $N_b = 0$ for exactly half the values of b and $N_b = 2N/S$ for the other half. This observation gives the desired result. □

Plugging Equation (**??**) into recurrence (**??**) with $T(0) = 0$ yields

$$T(N) \leq G\,T\left(\frac{N}{G}\right) + \frac{S}{2}T\left(\frac{2N}{S}\right) + O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + \log H\right)\right). \tag{5}$$

For the case $H < N$ and $2N \log N \leq H^{3/2}$, one iteration of the recurrence suffices to reduce the recursive subproblems to be of size at most H. By (**??**) we get

$$T(N) = O\left(\frac{N}{H}\left(T(H) + f\left(\frac{N}{H}\right) + \log H\right)\right). \tag{6}$$

Now let us consider the case $2N \log N > H^{3/2}$. If we plug into (**??**) the values of G and S from (**??**) and (**??**), we get

$$T(N) \leq \frac{\sqrt{2N}}{\sqrt{\log N} H^{1/4}} T\left(\sqrt{2N \log N} H^{1/4}\right) + O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + \log H\right)\right). \tag{7}$$

Let $N_{(i)}$ be the argument of T after i applications of recurrence (**??**). In other words, we have $N_{(0)} = N,\ \ N_{(1)} = \sqrt{2N \log N}\, H^{1/4}$, and so on. By repeatedly applying the recurrence s times (which we refer to as "unwinding"), we get

$$T(N) = O\left(\frac{N}{H} \sum_{0 \le i < s} 2^i f\left(\frac{N_{(i)}}{H}\right) + \frac{N}{H}\left(2^{s+1} - 1\right) \log H \right) + \frac{2^s N}{N_{(s)}} T(N_{(s)}), \tag{8}$$

where for $1 \le i \le s$ we have $\log N_{(i)} \le \log N$ and

$$N_{(i)} \le 2(\log N)\sqrt{H} \left(\frac{N}{2(\log N)\sqrt{H}} \right)^{1/2^i}. \tag{9}$$

The number of times we unwind recurrence (**??**) is chosen so that the resulting argument $N_{(s)}$ in (**??**) satisfies $2N_{(s)} \log N_{(s)} \le H^{3/2}$. Since $N_{(s)} \log N_{(s)} < {N_{(s)}}^2$, the condition is satisfied if ${N_{(s)}}^2 \le H^{3/2}/2$, which by (**??**) happens when

$$s = \left\lceil \log \left(\frac{\log\left(N/2(\log N)\sqrt{H}\right)}{\log H - 6 - 4 \log\log N} \right) \right\rceil + 2. \tag{10}$$

The following lemma establishes the formulas for the case $f(x) = \log x$ in Theorem **??**:

Lemma 8 *When $f(x) = \log x$, the algorithm sorts in deterministic time*

$$T(N) = O\left(\frac{N}{H} \left(\log \frac{N}{H} \log\left(\frac{\log N}{\log H}\right) + \frac{\log N}{\log H} T(H) \right) \right).$$

Proof: The formula for $2N \log N \le H^{3/2}$ follows directly from (**??**). When $2N \log N > H^{3/2}$, we can plug $f(x) = \log x$ into Equation (**??**) and use (**??**) for $T(N_{(s)})$, where s is given by (**??**). By the fact that $T(H) \ge \log H$, we get the desired bound. □

A similar proof establishes the solutions for the $f(x) = x^\alpha$ case in Theorem **??**:

Lemma 9 *When $f(x) = x^\alpha$, the algorithm sorts in deterministic time*

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

On a hypercube, the best known value of $T(H)$ is $O(\log H\ \log\log H)$ if precomputation is allowed and $O(\log H\,(\log\log H)^2)$ with no precomputation [CyP]. On an EREW-PRAM, we have $T(H) = O(\log H)$ [Col].

Lemmas **??** and **??** complete the proof for the upper bounds of Theorem **??**. The lower bounds for all the terms not involving $T(H)$ were shown in [ViSb]. 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 $f(x)$ on any interconnection that has $T(H) = O(\log H)$.

Theorem 5 *The algorithm given above is optimal for all well-behaved cost functions $f(x)$ for any interconnection that has $T(H) = O(\log H)$.*

Proof: This lower bound proof is essentially that of Theorem 6 in [ViSb] for their randomized algorithm. Let $T_{M,H}(N)$ denote the average number of I/O steps the sorting algorithm does with respect to an internal memory of size $M \geq H$, where in a single I/O step we allow each of the H hierarchies to simultaneously move a record between internal memory and external memory. This model is equivalent to using the cost function

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

and not counting the time for parallel prefix and routing on the network. We are viewing the algorithm only from the standpoint of I/O for the lower bound, and thus the matching time does not get counted. When $N > M$ and $2N \log N > H^{3/2}$, we get from (**??**) the recurrence

$$T_{M,H}(N) \leq \frac{\sqrt{N}}{\sqrt{\log N} H^{1/4}} T_{M,H}\left(\sqrt{2N \log N} H^{1/4}\right) + \sum_{1 \leq b \leq S} T_{M,H}(N_b) + O\left(\frac{N}{H}\right),$$

where $\sum_{1 \leq b \leq S} N_b = N$ and $N_b < 2N/S$ for all $1 \leq b \leq S$. When $M < N$ and $2N \log N \leq H^{3/2}$, we get $T_{M,H} = O(N/H)$. By the same kind of derivation above, we can show that this recurrence satisfies

$$T_{M,H} = O\left(\frac{N}{H}\frac{\log N}{\log M}\right). \tag{11}$$

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 can 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. Equation (**??**) is thus within an additive term of $O(M/H)$ of this optimal I/O bound. At the base memory level, in which $M = H$, 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. Hence, the amount of time used in the base memory level by the algorithm is $O((N/H) \log N)$.

Let us define

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

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. More specifically, the amount of time spent above the optimal is

$$\begin{aligned} &O\left(\frac{N}{H}\log N + \sum_{H \leq M < N} \frac{M}{H}\delta f\left(\frac{M}{H}\right)\right) \\ &= O\left(\frac{N}{H}\log N + \frac{N}{H}f\left(\frac{N}{H}\right) - f(1) - \frac{1}{H}\sum_{H \leq M < N} f\left(\frac{M+1}{H}\right)\right) \\ &= O\left(\frac{N}{H}\log N + \frac{N}{H}f\left(\frac{N}{H}\right)\right). \end{aligned} \tag{12}$$

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 (**??**) is the lower bound resulting from using a comparison-based sorting algorithm. The second term in (**??**) 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. It follows that the sorting algorithm is optimal. $\square$

3.4 Analysis of P-BT

Our algorithm for the P-BT model is similar to that for the P-HMM model. The only difference is that in Algorithm **??** we need to add another step right after Step (6) to reposition all the buckets into consecutive locations on each logical memory hierarchy. This repositioning is done on a logical-hierarchy-by-logical-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 $0 < \alpha < 1$. We choose

$$G = \begin{cases} \min\left\{ \dfrac{N^{1-\alpha}}{H^{1/4}}, \dfrac{N}{H} \right\} & \text{if } N > H^2; \\ \min\left\{ \dfrac{\sqrt{N}}{\sqrt{2}H^{1/4}}, \dfrac{H}{H} \right\} & \text{if } N \leq H^2; \end{cases}$$

$$S = \begin{cases} \min\left\{ \dfrac{N^{\alpha}}{H^{1/4}\log N}, \dfrac{2N}{H} \right\} & \text{if } N > H^2; \\ \min\left\{ \dfrac{\sqrt{N}}{\sqrt{2}H^{1/4}}, \dfrac{2N}{H} \right\} & \text{if } N \leq H^2. \end{cases}$$

(Note that the quantities defined above may be non-integral. The actual values of G and S are the truncated values of these quantities. But as noted in the last section, we ignore this distinction in order to avoid cluttering the asymptotic analysis.)

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

Lemma 10 *With the above values of G and S, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least $H/H' = \sqrt{H}$.*

Lemma 11 *With the above values of G and S, the condition for Corollary* **??** *is satisfied, and the buckets are approximately the same size.*

We need to make one more change to the algorithm for BT hierarchies, but one that is hard to write explicitly. Aggarwal *et al.* [ACSa] gave an algorithm for the "touch" problem, in which n consecutive records stored at the lowest possible level in the hierarchies must pass through the base memory level in linear order on each hierarchy. Their algorithm takes time $O(n \log\log n)$ for $0 < \alpha < 1$. In our sorting algorithm, all the data structures are processed in linear order throughout our algorithm, due to the sorted runs of size N/G

created while finding the partitioning elements. This gives us the same recurrence as for the P-HMM model, but with an effective cost function $f(x) = \log\log x^\alpha = O(\log\log x)$. The limiting step in the algorithm for P-BT is the need for repositioning the buckets, which can be done using the cited algorithm [ACSa] in time $O\left((N/H)(\log\log(N/H))^4\right)$. So the overall recurrence is

$$T(N) = G\,T\left(\frac{N}{G}\right) + \sum_b T(N_b) + O\left(\frac{N}{H}\left(\left(\log\log\frac{N}{H}\right)^4 + T(H)\right) + \frac{GS}{H}\log\log\frac{S}{H}\right), \quad (13)$$

for the case $N > 2H$, where $T(H)$ is the time needed to sort H items on H processors.

The term $(GS/H)\log\log(S/H)$ comes from the repeated mergings of the partition elements with the sorted runs $\mathcal{G}_i$ during the process of partitioning the file into buckets. The case $N < 2H$ is the same as for P-HMM.

Solving the recurrence as in Lemma **??** gives the following time bound, which completes the proof of Theorem **??**.

Lemma 12 *The given algorithm sorts in the P-BT model, with $f(x) = x^\alpha$, for $0 < \alpha < 1$, using time*

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

4 Parallel Disks with Parallel Processors

In this section, we present a version of Balance Sort for the parallel disk model. In the parallel disk model, we have D disks, each capable of simultaneously transferring in a single I/O a block of B records to or from the internal memory, which can hold up to $M \geq 4DB$ records. The internal memory is partitioned into $P \leq M$ local memories belonging to P processors connected by an EREW-PRAM interconnection network. Our algorithm is optimal in terms of the number of parallel disk I/Os to within a constant factor that we evaluate and also in terms of the internal processing time, assuming that $\log M = O(\log(M/B))$. If $\log(M/B) = o(\log M)$, we require a concurrent read/concurrent write (CRCW) PRAM interconnection to attain the internal processing bounds.

The algorithm for the parallel disk model is shown in Algorithm **??**. It is similar to that used for the P-HMM model, with the following changes:

1. In Algorithm **??**, we use $N \leq M$ rather than $N \leq 3H$ as the termination condition for the recursion in Algorithm **??**.

2. We call the Balance_Disks algorithm given in Algorithm **??** instead of the Balance routine given in Algorithm **??**. This algorithm has a different mechanism for handling internal fragmentation from that of Algorithm **??**.

3. The parameter we call D (the number of disks) is similar to the parameter we called H for parallel memory hierarchies; however, the number P of CPUs may be different from the number D of disks. We likewise use a partial striping of $D' = \sqrt{D}$ tracks.

4. We use a different method for computing the partitioning elements, given in Algorithm **??** (this method was described in [ViSa]).

Algorithm 6 [Sort_Disks(N, T)]

```
    if $N \le M$
(1)     Read $N$ locations
        Sort internally
(2)     Write $N$ locations
    else
(3)     $E :=$ ComputePartitionElements($S$)
        { The $T$ array says what location to start with on each hierarchy }
        Initialize $X, L, A$
(4)     Balance_Disks($T$)
        for $b := 1$ to $S$
(5)         $T :=$ Read $b$th row of $L$ {set in Balance}
            $N_b :=$ number of elements in bucket $b$
(6)         Sort_Disks($N_b, T$)
(7)         Append sorted bucket to output area
```

Algorithm 7 [Balance_Disks(T)]

```
    while there remain elements to process
(1)     read data to fill memoryload (some data may still be in memory)
(2)     assign all $M$ records to buckets
(3)     group each bucket together (sort using bucket number as key)
(4)     for each consecutive $D'$ logical blocks of values in parallel
(5)         Call Rebalance on full logical blocks (or all logical blocks if the last time
                through)
```

Algorithm 8 [ComputePartitionElements(S)]

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

5. We let $S = \sqrt{M/B}$.

6. The Rebalance routine operates only on the first D' full logical blocks.

The procedure for finding the partitioning elements uses as a subroutine Algorithm **??**,

Algorithm 9 [Select(S_0, k)]

```
        $t := \lceil |S_0|/M \rceil$
        for $i := 1$ to $t$
(1)         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($k$th element)
        $s :=$ the median of $R_1, \ldots, R_t$ using algorithm from [BFP]
        Partition $S_0$ 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 $k \leq k_1$
(2)         Select($S_1, k$)
        else
(3)         Select($S_2, k - k_1$)
```

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.

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 logical blocks on our logical disks. If this grouping is not possible, the matching will not correctly distribute records into buckets, and the sorting algorithm will fail. In particular, line (5) of Algorithm **??** may not have any full logical blocks to write. We show in Lemma **??** that at least half the logical blocks are full for any memoryload.

Lemma 13 *At least half the logical blocks at line (5) of Algorithm* **??** *are full for any memoryload.*

Proof: There are a total of $M/(B\sqrt{D})$ logical blocks per memoryload, of which at most S are only partially full. Thus, the fraction of partially full blocks is at most

$$\frac{S}{M/(B\sqrt{D})} = \frac{\sqrt{\frac{M}{B}}}{\frac{M}{B\sqrt{D}}} = \sqrt{\frac{DB}{M}} \leq \sqrt{\frac{DB}{4DB}} = \frac{1}{2}$$

since $M \geq 4DB$. The remaining half of the logical blocks must be full. □

In order to show bounds on I/O and internal processing time, we need to show that the routine for computing the partition elements produces buckets that have nearly the same size. We reapply Lemma **??**, except this time we have $p = (S-1)/2$. Substituting in G and S, we get a slop that is less than $\sqrt{M/B}N/2M$. The ratio between the slop and the average number of elements per bucket is $1/(2B) \leq 1/2$. Thus, we have for any bucket b,

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

4.1 I/O Bounds

The "big oh" I/O bound is easy to show for the new algorithm. The A, X, L, and E matrices all reside in the internal memory, so there is no I/O cost to access them. A "memoryload" is the collection of $O(M)$ records that fit into memory. 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

$$T(N) = \begin{cases} S\,T\Big(\dfrac{N}{S}\Big) + O\Big(\dfrac{N}{DB}\Big) & \text{if } N > M; \\ O\left(\dfrac{N}{DB}\right) & \text{if } N \le M, \end{cases}$$

which has solution

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

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

In fact, we can go beyond the standard analysis to determine the constant of proportionality hidden by the "big oh". Let α be the factor that represents the ratio between the worst-case time for reading N records and the best-case time, so that reading N records takes at most $\alpha\lceil N/DB \rceil$ reads. (We showed in Lemma **??** that $\alpha = 2$.) Then we derive our recurrences in Table **??**. In this table, we let $R_i(N)$ and $W_i(N)$ be the total number of reads and writes done by Algorithm i when processing N records, respectively.

Line (5) of Algorithm **??** has no I/Os since the L array fits entirely within internal memory. Line (7) of Algorithm **??** also requires no I/Os since we can arrange things so that Line (2) of Algorithm **??** always writes to the next locations in the output area.

We can solve the recurrences in Table **??** in reverse order.

$$\begin{aligned} W_{\mathbf{??}}(N) &= 0; \\ R_{\mathbf{??}}(N) &= \begin{cases} \left\lceil \dfrac{N}{DB} \right\rceil + R_{\mathbf{??}}(N)\left(\left\lceil \dfrac{N}{2} \right\rceil\right) & N > M; \\ \left\lceil \dfrac{N}{DB} \right\rceil & N \le M; \end{cases} \\ &= 2\left\lceil \frac{N}{DB} \right\rceil + O(\log N). \end{aligned}$$

where the $O(\log N)$ term adds up all the round-ups in the recursive arguments. Algorithms **??** and **??** are not recursive, so they can be computed directly:

$$\begin{aligned} W_{\mathbf{??}}(N) &\le \frac{N}{\sqrt{BMD}} \\ &\le \frac{N}{2BD^{3/2}}; \\ R_{\mathbf{??}}(N) &< (\alpha+1)\left\lceil \frac{N}{DB} \right\rceil + O\left(\log \frac{N}{\sqrt{M/B}}\right); \end{aligned}$$

$$\begin{aligned}
W_{??}(N) &= \left\lceil \frac{N}{DB} \right\rceil; \\
R_{??}(N) &= \alpha \left\lceil \frac{N}{DB} \right\rceil.
\end{aligned}$$

So the final recurrence is

$$\begin{aligned}
W_{??}(N) &= \begin{cases} \left\lceil \frac{N}{DB} \right\rceil + \frac{N}{2BD^{3/2}} + S \cdot W_{??}(N)\left(\left\lceil \frac{N}{S} \right\rceil\right) & N > M; \\ \left\lceil \frac{N}{DB} \right\rceil & N \le M; \end{cases} \\
R_{??}(N) &= \begin{cases} (2\alpha+1)\left\lceil \frac{N}{DB} \right\rceil + S \cdot R_{??}(N)\left(\left\lceil \frac{N}{S} \right\rceil\right) & N > M; \\ \alpha \left\lceil \frac{N}{DB} \right\rceil & N \le M. \end{cases}
\end{aligned}$$

We can get an exact solution to this recurrence by noting that

$$F(N) = \begin{cases} kN + S \cdot F\left(\left\lceil \frac{N}{S} \right\rceil\right) & N > M; \\ F(M) & N \le M; \end{cases}$$

has solution

$$F(N) = kN\left(\frac{\log N/M}{\log S}\right) + F(M) + o\left(N\frac{\log N/M}{\log S}\right).$$

So

$$\begin{aligned}
W_{??}(N) &\le \left\lceil \frac{N}{DB} \right\rceil \left(1 + \frac{1}{2\sqrt{D}}\right)\left(\frac{\log N/M}{\log S}\right) + \left\lceil \frac{N}{DB} \right\rceil + o\left(\frac{N}{DB}\frac{\log N/M}{\log S}\right) \\
&= 2\left(1 + \frac{1}{2\sqrt{D}}\right)\left\lceil \frac{N}{DB} \right\rceil \left(\frac{\log N/B}{\log M/B}\right) + o\left(\frac{N}{DB}\frac{\log N/B}{\log M/B}\right); \\
R_{??}(N) &= 2(2\alpha+1)\left\lceil \frac{N}{DB} \right\rceil \left(\frac{\log N/B}{\log M/B}\right) + o\left(\frac{N}{DB}\frac{\log N/B}{\log M/B}\right).
\end{aligned}$$

Since $\alpha = 2$, we are within the following factors of optimal in the leading terms:

Read	10
Write	$2 + \frac{2}{\sqrt{D}}$
Total	$6 + \frac{1}{\sqrt{D}}$

This constant factor can be further reduced by choosing different values for S. If we use

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

for any $0 < \beta < 1$, the same big-oh complexity is achieved. As $\beta \to 1$, the factors approach half of those listed above, so that in the limit that $\beta \to 1$ and $D \to \infty$, the overall factor is within 3 of optimal.

4.2 Processing Time

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 $P \leq M \log \min\{M/B, \log M\}/\log M$. If $\log M$ is not $O(\log(M/B))$, we assume a CRCW interconnection. 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.

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 non-recursive 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 step (1) of Algorithm **??**.

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

Lemma 14 *Step (2) of Algorithm* **??** *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 $\sqrt{M/B}$ partition elements for each of its records, giving a total time of $O((M/P)\log(M/B))$. □

Lemma 15 *Step (3) of Algorithm* **??** *can be done in time $O((M/P)\log(M/B))$ for each memoryload if $\log M = O(\log(M/B))$ or if $P \leq M \log \min\{M/B, \log M\}/\log M$.*

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 EREW PRAM 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 start by considering only $P \geq M/\log M$. We 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, but requires a CRCW PRAM. We need to consider two cases:

> *Case 1:* $M/B \leq \log M$. In this case, we let $n = M$. The key values range from 1 to $\sqrt{M/B} < M/B \leq \log M$, so we can apply the Rajasekaran-Reif algorithm directly to sort in time $O(\log M) = O((M/P)\log(M/B))$ with $M/\log M$ processors; we have that many processors available.
>
> *Case 2:* $M/B > \log M$. We 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 $\log\log M$ 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

$$P < \frac{M\log\log M}{\log M} < M\frac{\log(M/B)}{\log M}.$$

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

In order to show that loop (4) of Algorithm **??** 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 the histogram matrix X or recompute the portion of the auxiliary matrix A that needs 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 16 *At most two tracks need to access values of any specific row of matrices 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. $\square$

We can thus 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 17 *Loop (4) of Algorithm* **??** *can be done in time $O((M/P)\log(M/B))$ for any $P \le M$.*

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

$$T_X = \begin{cases} O\left(\dfrac{M\log D}{PB\sqrt{D}}\right) & \text{if } P \le M\log DB\sqrt{D}; \\ O(1) & \text{if } P > M\log DB\sqrt{D}. \end{cases}$$

In both cases, the time is $O((M/P)\log(M/B))$ for any $P \le M$.

We can do the ComputeAux routine by the following two steps:

1. Sort S sets of D' numbers in parallel to pick the median element.
2. Update all the SD' elements of the auxiliary matrix A.

The sorts take time

$$T_{A1} = \begin{cases} O\left(\frac{1}{P}\sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}\log\frac{\sqrt{D}}{\log D}\right) & \text{if } P \le \sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}; \\ O\left(\log\frac{\sqrt{D}}{\log D}\right) & \text{if } P > \sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}. \end{cases}$$

In both cases, the time is $O((M/P)\log(M/B))$ for any $P \le M$. The update time is

$$T_{A2} = \begin{cases} O\left(\frac{1}{P}\sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}\right) & \text{if } P \le \sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}; \\ O(1) & \text{if } P > \sqrt{\frac{M}{B}\frac{\sqrt{D}}{\log D}}. \end{cases}$$

Again, the time is $O((M/P)\log(M/B))$ for any $P \le M$ if $M \ge D/(B\log^2 D)$, which is true since $D/\log^2 D < 2D$ for all $D \ge 2$.

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

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

The set-up time can be done in parallel time

$$T_{R1} = \begin{cases} O\left(\frac{M}{PB\log^2 D}\right) & \text{if } P \le \frac{M}{B\log^2 D}; \\ O(1) & \text{if } P > \frac{M}{B\log^2 D}. \end{cases}$$

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

$$T_{R2} = \begin{cases} O\left(\frac{M\log D}{PBD}\right) & \text{if } P \le \frac{M}{B}; \\ O(\log D) & \text{if } P > \frac{M}{B}. \end{cases}$$

Both of these steps take $O((M/P)\log(M/B))$ time, for any $P \le M$. The swaps involve moving up to M items, but no internal processing time other than touching them, for a time of $O(M/P)$. Each individual step thus meets the time bound proposed by the lemma, establishing its validity. $\square$

Finally, we are ready to complete the proof of our main result for disk sorting, Theorem **??**. 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 $P \le M \log\min\{M/B, \log M\}/\log M$ 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

$$\begin{aligned} O\left(\left(\frac{N}{M}\frac{\log(N/B)}{\log(M/B)}\right)\left(\frac{M}{P}\log(M/B)\right)\right) &= O\left(\frac{N}{P}(\log N - \log B)\right) \\ &= O\left(\frac{N}{P}(\log N)\right) \end{aligned}$$

as desired.

5 Conclusions

In this paper, 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, ViSb] and the deterministic disk algorithm of Nodine and Vitter [NoVb]. The algorithm applies to P-HMM, P-BT, and the parallel variants of the UMH models. In the parallel disk model with parallel CPUs, our algorithm is simultaneously optimal in terms of both the number of I/Os and the internal processing time. The algorithms do not require non-striped writes, which is a useful feature when the disk subsystem requires only striped writes so that it can do its error checking and correcting protocols.

Our algorithms are both theoretically efficient and practical in terms of constant factors, and we expect our balance technique to be quite useful as large-scale parallel memories are built, not only for sorting but also for other load-balancing applications on parallel disks and parallel memory hierarchies. Although we have presented a deterministic algorithm, the randomized algorithm resulting from the randomized matching is even simpler to implement in practice.

As pointed out in the conclusions of the companion paper [NoVa], Balance Sort performs write operations in complete stripes, which makes it easy to write parity information for use in error correction and recovery. However, 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

[AAC] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra & Marc Snir, "A Model for

Hierarchical Memory," *Proceedings of 19th Annual ACM Symposium on Theory of Computing*, New York, NY (May 1987).

[ACSa] Alok Aggarwal, Ashok K. Chandra & Marc Snir, "Hierarchical Memory with Block Transfer," *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science*, Los Angeles, CA (October 1987).

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

[ACF] Bowen Alpern, Larry Carter, Ephraim Feig & T. Selker, "The Uniform Memory Hierarchy Model of Computation," *Algorithmica* 12 (1994), 72–109.

[BFP] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest & Robert 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.

[CyP] Robert E. Cypher & 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.

[Lei] F. Thomson Leighton, in *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[NoVa] Mark H. Nodine & Jeffrey Scott Vitter, "Optimal Deterministic Sorting on Parallel Disks," *submitted for publication.*

[NoVb] Mark H. Nodine & 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 & 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).

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

[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.

[ViN] Jeffrey Scott Vitter & 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.

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

[ViSb] Jeffrey Scott Vitter & Elizabeth A. M. Shriver, "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories," *Algorithmica* 12 (1994), 148–169.

Alg.	Line	Repeats	Time/repeat	Total time
??	1,9	2	$f\left(\frac{H'}{H}\right)$	$2f\left(\frac{H'}{H}\right)$
	2	1	$f\left(\frac{SH'}{H}\right)+f\left(\frac{(H')^2}{H}\right)$	$f\left(\frac{SH'}{H}\right)+f\left(\frac{(H')^2}{H}\right)$
	3,4,5	2	$2f\left(\frac{H'-1}{H}\right)+P(H')$	$4f\left(\frac{H'-1}{H}\right)+2P(H')$
	6	1	$R_c(H)$	$R_c(H)$
	7	1	$2f\left(\frac{(H')^2}{H}\right)$	$2f\left(\frac{(H')^2}{H}\right)$
	8	1	$f\left(\frac{N}{H}\right)$	$f\left(\frac{N}{H}\right)$
	10	1	$2f\left(\frac{SH'}{H}\right)$	$2f\left(\frac{SH'}{H}\right)$
	11	1	$\frac{(H')^2}{H}f\left(\frac{SH'}{H}\right)$	$\frac{(H')^2}{H}f\left(\frac{SH'}{H}\right)$
	Total	$C_{??}=O\left(\frac{(H')^2}{H}f\left(\frac{SH'}{H}\right)+P(H')+R_c(H)+f\left(\frac{N}{H}\right)\right)$		
??	1	$O\left(\frac{N}{H}\right)$	$f\left(\frac{N}{H}\right)$	$\frac{N}{H}f\left(\frac{N}{H}\right)$
	2	1	see text	$\frac{GS}{H}f\left(\frac{S}{H}\right)$
	4	$O\left(\frac{N}{H}\right)$	$C_{??}$	$\frac{N}{H}C_{??}$
	5	$O\left(\frac{N}{H}\right)$	$f\left(\frac{SH'}{H}\right)$	$\frac{N}{H}f\left(\frac{SH'}{H}\right)$
	Total	$C_{??}=O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right)+f\left(\frac{SH'}{H}\right)+C_{??}\right)+\frac{GS}{H}f\left(\frac{S}{H}\right)\right)$		
??	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	$C_{??}=G\,T\left(\frac{N}{G}\right)+\left(\frac{N}{H\log N}+\frac{\log N\log\log N}{H}+\frac{S}{H}\right)f\left(\frac{\log N}{H}\right)+\frac{S}{H}f\left(\frac{S}{H}\right)$		
??	4	1	$C_{??}$	$C_{??}$
	5	2	$\frac{SH'}{H}f\left(\frac{SH'}{H}\right)$	$2\frac{SH'}{H}f\left(\frac{SH'}{H}\right)$
	6	1	$C_{??}$	$C_{??}$
	7	S	$f\left(\frac{SH'}{H}\right)$	$Sf\left(\frac{SH'}{H}\right)$
	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	$T(N)=\sum_b T(N_b)+C_{??}+O\left(C_{??}+\left(\frac{SH'}{H}+S\right)f\left(\frac{SH'}{H}\right)+\frac{N}{H}f\left(\frac{N}{H}\right)\right)$		

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

Alg.	Line	Reads	Writes
??	1	$\alpha\lceil N/DB\rceil$	0
	2	0	$\lceil N/DB\rceil$
	3	$R_{\textbf{??}}(N)$	$W_{\textbf{??}}(N)$
	4	$R_{\textbf{??}}(N)$	$W_{\textbf{??}}(N)$
	5	0	0
	6	$SR_{\textbf{??}}(\lceil N/S\rceil)$	$SW_{\textbf{??}}(\lceil N/S\rceil)$
	7	0	0
??	1	$\alpha\lceil N/DB\rceil$	0
	5	0	$\lceil N/DB\rceil$
??	1	$\alpha\lceil N/DB\rceil$	0
	2	0	$< N/(\sqrt{BM}D)$
	3	$< \left(\sqrt{M/B}/2\right) R_{\textbf{??}}\left(N/\sqrt{M/B}\right)$	0
??	1	$\lceil N/DB\rceil$	0
	2,3	$R_{\textbf{??}}(N/2)$	0

Table 2: Derivation of the recurrence for disk I/Os.