Algorithmica (1994) 12: 110–147

Algorithmica

Algorithms for Parallel Memory, I: Two-Level Memories[1]

J. S. Vitter[2] and E. A. M. Shriver[3]

Abstract. We provide the first optimal algorithms in terms of the number of input/outputs (I/Os) required between internal memory and multiple secondary storage devices for the problems of sorting, FFT, matrix transposition, standard matrix multiplication, and related problems. Our two-level memory model is new and gives a realistic treatment of *parallel block transfer*, in which during a single I/O each of the P secondary storage devices can simultaneously transfer a contiguous block of B records. The model pertains to a large-scale uniprocessor system or parallel multiprocessor system with P disks. In addition, the sorting, FFT, permutation network, and standard matrix multiplication algorithms are typically optimal in terms of the amount of internal processing time. The difficulty in developing optimal algorithms is to cope with the partitioning of memory into P separate physical devices. Our algorithms' performances can be significantly better than those obtained by the well-known but nonoptimal technique of disk striping. Our optimal sorting algorithm is randomized, but practical; the probability of using more than l times the optimal number of I/Os is exponentially small in $l(\log l)\log(M/B)$, where M is the internal memory size.

Key Words. I/O, Input/output, Disk, Secondary memory, Sorting, Distribution sort, FFT, Matrix multiplication, Transposition, Permutation.

1. Introduction.

Sorting is the canonical information-processing application. It accounts for roughly 20–25% of the computing resources on large-scale computers [8], [10]. In applications where the file of records cannot fit into internal memory, the records must be stored on (external) secondary storage, usually in the form of disks. Sorting in this framework is called *external sorting*. The bottleneck in external sorting and many other applications is the time for the input/output (I/O) between internal memory and the disks [8], [10]. This bottleneck is accentuated as processors get faster and parallel computers are used.

The remedy we explore in this paper is to use secondary storage systems with parallel capabilities [5], [6], [11], [14], [17]. We restrict our attention in this paper

[1] A summarized version of this research was presented at the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, MD, May 1990. This work was done while the first author was at Brown University. Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM, by NSF Research Grants DCR-8403613 and CCR-9007851, by Army Research Office Grant DAAL03-91-G-0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-91-J-4052 ARPA Order 8225. This work was done in part while the second author was at Brown University supported by a Bellcore graduate fellowship and at Bellcore.

[2] Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129, USA.

[3] Courant Institute, New York University, 251 Mercer Street, New York, NY 10012, USA.

Received September 26, 1990; revised January 14, 1993. Communicated by C. K. Wong.

to two-level storage systems with random access secondary storage. Magnetic disks, for example, provide the functionality needed in our model of secondary storage, so for simplicity we refer to secondary storage as disk storage, consisting of one or more disk drives. Efficient algorithms for multilevel hierarchical memory are considered in the companion paper [19].

In a previous work Aggarwal and Vitter [1] presented optimal upper and lower bounds on the I/O needed for sorting-related problems of size N using a two-level memory model where internal memory can store M records and the secondary memory size is limitless. In their model an I/O can simultaneously transfer P physical blocks, each consisting of B contiguous records. Their results generalized the groundbreaking work done by Floyd [4], who gave optimal bounds for sorting, realized by standard two-way merge sort, for the special case $P = 1$ and $M = 2B = \sqrt{N}$. The $P > 1$ model in [1] is somewhat unrealistic, however, because secondary storage is usually partitioned into P separate physical devices, each capable of transferring only one block per I/O.

We are interested in optimal algorithms for realistic two-level storage systems that allow P simultaneous data transfers. By realistic, we mean that each block transfer must be associated with a separate secondary storage device. In the next section we define a realistic two-level memory model with parallel block transfer, which consists of an internal memory (capable of storing M records) and P disks, each disk capable of simultaneously transferring one block of B records, as shown in Figure 1. Another version of our model that turns out to be sufficient for our purposes is to have the P disks controlled by P' processors, each with internal memory capable of storing M/P' records. If $P' \leq P$, each of the P' processors can drive about P/P' disks; if $P < P' \leq M$, each disk is associated with about P'/P processors. The P' processors are connected by a network, such as a hypercube or cube-connected cycles, that allows some basic operations like sorting of the M records in the internal memories to be performed quickly in parallel in $O((M/P') \log M)$ time. The special case in which $P = P'$ is shown in Figure 2.

Our main measure of performance is the number of parallel I/Os required, but we also consider the amount of internal computation. The bottleneck in the problems we consider is generally the I/O, at least in the multiprocessor versions of the two-level memory model described above.

In this paper we consider large-scale instances of the following important problems, which are defined in Section 3: sorting, permuting, matrix transposition, FFT, permutation networks, and standard matrix multiplication. In Section 4 we state our main results, namely, tight upper and lower bounds on the number of I/Os and amount of internal processing needed to solve these problems, for each of the uniprocessor and multiprocessor models mentioned above. The standard matrix multiplication algorithm is simultaneously optimal in terms of internal processing time. Our sorting, FFT, and permutation network algorithms are also simultaneously optimal in terms of the internal processing time when $P' = O(1)$ or $\log M = O(\log(M/B))$. For large-scale computer configurations, in which P and PB are large, our algorithms for sorting, permuting, FFT, and permutation networks can be significantly faster than the algorithms obtained by applying the well-known technique of disk striping to good single-disk algorithms.

Sections 5–7 are devoted to the algorithms and their analyses. In Section 5 we develop optimal algorithms for matrix transposition, FFT, and permutation networks, by making use of the shuffle-merge primitive. Even though these problems are sorting-related, it is much easier to develop optimal algorithms for them than it is for sorting, since their I/O schedules are fixed and nonadaptive.

The main contribution of this paper is the optimal randomized algorithm for sorting (and permuting) and its probabilistic analysis in Section 6. The probability that it uses more than l times the optimal number of I/Os is exponentially small in $l(\log l)\log(M/B)$.[4] The sorting algorithm is a variant of a distribution sort; a combination of two randomized techniques is used to do the partitioning so as to take full advantage of parallel block transfer. The algorithm requires that the disks operate independently in read mode, but in each disk write, the track written to is the same among all the disks, which facilitates writing error correction information (see [5], for example); in fact, some of the initial disk arrays being developed require that parallel writes have this same-track property. In Section 7 we cover standard matrix multiplication. Conclusions and open problems are discussed in Section 8.

2. The Two-Level Memory Model.

First we define the parameters for our *two-level memory model with parallel block transfer*, as shown in Figures 1 and 2:

Definition 1. The parameters are defined by

$$N = \text{number of records in the file},$$

$$M = \text{number of records that can fit in the internal memory},$$

$$B = \text{number of records per block},$$

$$P = \text{number of disk drives},$$

$$P' = \text{number of internal processors},$$

where $1 \leq B \leq M/2$, $M < N$, $1 \leq P \leq \lfloor M/B \rfloor$, and $1 \leq P' \leq M$. The parameters N, M, B, P, and P' are referred to as the *file size*, *memory size*, *block size*, *number of disks*, and *number of processors*, respectively. We denote the P disks by $\mathscr{D}_1, \mathscr{D}_2, \ldots, \mathscr{D}_P$. Each disk is partitioned into consecutive *tracks*, each capable of storing one block of B records, as shown in Figure 3. (For simplicity in our model, we assume that a block, our unit of transfer, is the same size as a track.) If no disk is specified when we refer to the "kth track," we mean the kth track of all P disks collectively.

For purposes of making the problem definitions we give in the next section more concrete, the locations on disk are numbered track-by-track in the following

[4] For simplicity of notation, we use $\log x$, where $x \geq 1$, to denote the quantity $\max\{1, \log_2 x\}$.

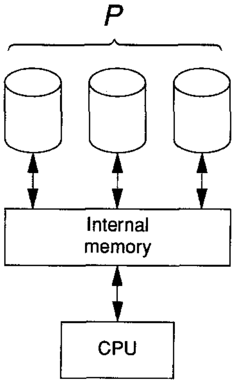

Fig. 1. The uniprocessor two-level storage model with parallel block transfer. The measure of performance is the number of I/Os. During an I/O, each of the P disks can simultaneously transfer a block of B contiguous records to or from the internal memory of size M.

cyclical fashion. Track 1 contains the first PB memory locations:

track 1 of $\mathscr{D}_1$ contains memory locations $1, 2, \ldots, B$,
track 1 of $\mathscr{D}_2$ contains memory locations $B + 1, B + 2, \ldots, 2B$,
$$\vdots$$
track 1 of $\mathscr{D}_P$ contains memory locations $(P - 1)B + 1, (P - 1)B + 2, \ldots, PB$.

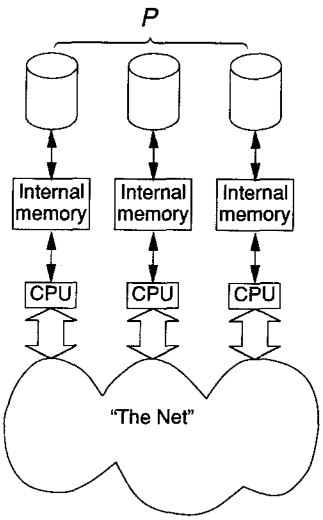

Fig. 2. An alternate multiprocessor version of the model, for $P' = P$. Each of the P disks is controlled by a separate processor with its own internal memory of size M/P. The interprocessor communication is assumed to be sufficiently fast so that internal sorting can be done rapidly in parallel.

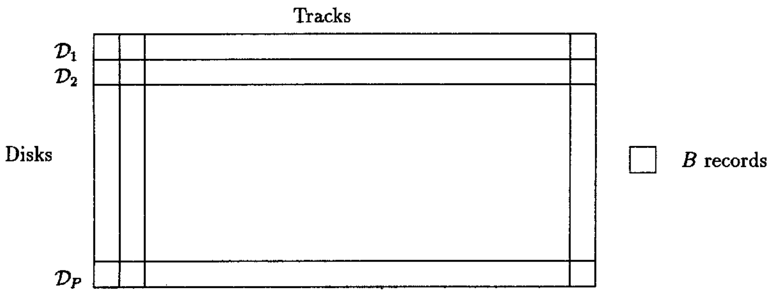

Fig. 3. The disks are represented by horizontal lines and the tracks by vertical lines.

Track 2 contains the next PB memory locations:

track 2 of $\mathscr{D}_1$ contains memory locations $PB + 1, PB + 2, \ldots, (P + 1)B$,
track 2 of $\mathscr{D}_2$ contains memory locations $(P + 1)B + 1, (P + 1)B + 2, \ldots, (P + 2)B$,

$$\vdots$$

track 2 of $\mathscr{D}_P$ contains memory locations $(2P - 1)B + 1, (2P - 1)B + 2, \ldots, 2PB$.

The numbering continues in this fashion for tracks 3, 4, ...

Parallelism appears in our model in two basic ways. First, records are transferred concurrently in blocks of B contiguous records. It takes roughly the same amount of time to access and transfer one block as it does one record. This reflects the fact that the seek time for a record greatly dominates the time to transmit a record. The second type of parallelism arises because P blocks can be transferred in a single I/O. We make the realistic restriction that the P blocks must be associated with tracks from P *different* disks. That is, only one track per disk can be accessed, but there is no constraint on which track is accessed on each disk.

The restriction that only one block can be accessed per disk during an I/O is what distinguishes our model from the less realistic model of [1]. This distinction is akin to the difference in parallel computation between the MPC (module parallel computer) model and the less realistic PRAM model. However, general PRAM simulation techniques use logarithmic time per step; if they were applied to the algorithms in [1], the resulting algorithms would not be optimal in terms of I/O. The algorithms we develop on our model use the same number of I/Os as those in [1] for the less realistic model.

3. Problem Definitions.

The problems we consider in this paper have been well described in the literature. Most of the following definitions are those from [1], with suitable modifications.

SORTING.

Problem Instance: The internal memory is empty, and the N records are stored in the first N locations of secondary storage.

Goal: The internal memory is empty, and the N records are stored in sorted nondecreasing order in the first N locations of secondary storage.

Permuting.
The problem instance and goal are the same as for the sorting problem, except that the key values of the N records are required to form a permutation of $\{1, 2, \ldots, N\}$.

Matrix Transposition.
Problem Instance: The internal memory is empty, and a $p \times q$ matrix $A = (A_{i,j})$ of $N = pq$ records is stored row by row in the first N locations of secondary storage.
Goal: The internal memory is empty, and the transposed matrix A^{T} is stored row by row in the first N locations of secondary storage. (The $q \times p$ matrix A^{T} is called the transpose of A if $A^{\mathrm{T}}_{i,j} = A_{j,i}$, for all $1 \le i \le q$ and $1 \le j \le p$.)

The matrix transposition problem above is a special case of permuting in which the permutation to be realized corresponds to converting a matrix from row-major order to column-major order.

Fast Fourier Transform (FFT).
Problem Instance: Let N be a power of 2. The internal memory is empty, and the N records are stored in the first N locations of secondary storage.
Goal: The internal memory is empty, the N output nodes of the FFT digraph are "pebbled" (as explained below), and the N records are stored in the first N locations of secondary storage.

The FFT digraph consists of $\log N + 1$ levels each containing N nodes; level 0 contains the N input nodes, and level $\log N$ contains the N output nodes. Each noninput node has indegree 2, and each nonoutput node has outdegree 2. We denote the ith node ($0 \le i \le N - 1$) in level j ($0 \le j \le \log N$) in the FFT digraph by $n_{i,j}$. For $j \ge 1$ the two predecessors to node $n_{i,j}$ are nodes $n_{i,j-1}$ and $n_{i \oplus 2^{j-1}, j-1}$, where $\oplus$ denotes the exclusive-or operation on the binary representations. (Note that nodes $n_{i,j}$ and $n_{i \oplus 2^{j-1}, j}$ each have the same two predecessors.)

The ith node in each level corresponds to record R_i. We are allowed to pebble node $n_{i,j}$ if its two predecessors $n_{i,j-1}$ and $n_{i \oplus 2^{j-1}, j-1}$ have already been pebbled and if the records R_i and $R_{i \oplus 2^{j-1}}$ corresponding to the two predecessors both reside in internal memory. Intuitively, the FFT problem can be phrased as the problem of pumping the records into and out of internal memory in a way that permits the computation implied by the FFT digraph.

Permutation Networks. The problem instance and goal are the same as for the FFT problem, except that the permutation network digraph (see below) is pebbled instead of the FFT digraph.

A permutation network is a sorting network [8] consisting of comparator modules or switches that can be set by external controls so that any desired permutation of the inputs can be realized at the output level of the network. It consists of $J + 1$ levels, for some $J \ge \log N$, each containing N nodes. Level 0 contains the N input nodes, and level J contains the N output nodes. All edges are directed between adjacent levels, in the direction of increasing index. For $0 \le i \le N-1$ and $0 \le j \le J$ we denote the ith node in level j as $n_{i,j}$. For each $j \ge 1$ there is an edge from $n_{i,j-1}$ to $n_{i,j}$. In addition, $n_{i,j}$ can have one other predecessor, call it $n_{i',j-1}$, but in that case there is also an edge from $n_{i,j-1}$ to $n_{i',j}$; that is, nodes $n_{i,j}$ and $n_{i',j}$ have the same two predecessors. We can think of there being a "switch" between nodes $n_{i,j}$ and $n_{i',j}$ that can be set either to allow the data from the previous level to pass through unaltered (that is, the data in node $n_{i,j-1}$ goes to $n_{i,j}$ and the data in $n_{i',j-1}$ goes to $n_{i',j}$) or else to swap the data (so that the data in $n_{i,j-1}$ goes to $n_{i',j}$ and the data in $n_{i',j-1}$ goes to $n_{i,j}$).

A digraph like this is called a permutation network if, for each of the $N!$ permutations $\langle p_1, p_2, \ldots, p_N \rangle$, we can set the switches in such a way to realize the permutation; that is, data at each input node $n_{i,0}$ is routed to output node $n_{p_i,J}$. The ith node in each level corresponds to the current contents of record R_i, and we can pebble node $n_{i,j}$ if its predecessors have already been pebbled and if the records corresponding to those predecessors reside in internal memory.

There is an important difference between permutation networks and general permuting. In the latter case the I/Os may depend upon the desired permutation, whereas with permutation networks all $N!$ permutations can be generated by the same sequence of I/Os or memory accesses.

Standard Matrix Multiplication.
Problem Instance: The internal memory is empty. The elements of two $k \times k$ matrices, A and B, where $2k^2 = N$, are each stored in the first N locations of secondary storage.
Goal: The internal memory is empty, and the product $C = A \times B$, formed by the standard matrix multiplication algorithm that uses $O(k^3)$ arithmetic operations, is stored in the first N locations of secondary storage.

4. Main Results.

In this section we state our main results, Theorems 1–5, which report optimal algorithms in terms of the number of I/Os and internal processing time, in our two-level model with parallel block transfer, to solve the problems defined in the previous section.

Theorem 1. *The number of I/Os required for sorting N records is*

$$\Theta\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right);$$

the internal processing time is $O(N \log(N/B)(\log M)/P' \log(M/B))$, which is the optimal time $O((N \log N)/P')$, for example, when $\log M = O(\log(M/B))$. The upper bounds are given by a randomized algorithm; the probability of using more than the average number of I/Os or internal processing time falls off exponentially. The lower bounds apply to both the average case and the worst case. The lower bound on *internal processing time is the well-known* $O((N \log N)/P')$ *bound from the comparison model of computation. 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* $B \log(M/B) = o(\log(N/B))$.

THEOREM 2. *The number of I/Os required for computing the N-input FFT digraph is*

$$\Theta\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right);$$

the internal processing time is $O(N \log(N/B)(\log M)/P' \log(M/B))$, *which is the optimal time* $O((N \log N)/P')$, *for example, when* $\log M = O(\log(M/B))$. *The lower bound on internal processing time is* $O((N \log N)/P')$. *The average-case and worst-case number of I/Os required for computing any N-input permutation network is*

$$\Omega\left(\frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right);$$

furthermore, there are permutation networks such that the number of I/Os needed to compute them is

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

and the internal processing time is $O(N \log(N/B) \log M)/P' \log(M/B))$, *which is optimal when* $\log M = O(\log(M/B))$. *All lower bounds apply to both the average case and the worst case.*

THEOREM 3. *The number of I/Os required to permute N records is*

$$\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right\}\right),$$

and the internal processing time is

$$O(\min\{N(\log P')/P', N \log(N/B)(\log M)/P' \log(M/B)\}).$$

The I/O lower bound applies to both the average case and the worst case. The second term in the I/O upper bound corresponds to the randomized algorithm of Theorem 1, *and the first term in the I/O upper bound makes use of a modification of Phase* 1 *of the randomized algorithm.*

THEOREM 4. *The number of I/Os required to transpose a $p \times q$ matrix of $N = pq$ elements is*

$$\Theta\left(\frac{N}{PB}\left(1 + \frac{\log \min\{M, p, q, N/B\}}{\log(M/B)}\right)\right),$$

and the internal processing time is greater by a multiplicative factor of $O((PB \log P')/P')$.

THEOREM 5. *The number of I/Os required to multiply two $k \times k$ matrices using the standard matrix multiplication algorithm is*

$$\Theta\left(\frac{k^3}{\min\{k, \sqrt{M}\}PB}\right),$$

and the internal processing time required is $\Theta(k^3/P')$.

The I/O bounds in the theorems can be regarded in terms of the number of "passes" through the file. One "pass" corresponds to the number of I/Os needed to read and write the file once, which is $2N/PB$. A "linear-time" algorithm (defined to be one that requires a constant number of passes through the file) would use $O(N/PB)$ I/Os. The logarithmic factors that multiply the N/PB term in the above expressions indicate the degree of nonlinearity.

The I/O lower bounds for Theorems 1–4 follow from the lower bounds proved in [1] for the less realistic model in which P tracks can be accessed on the same disk in a single I/O. Since any algorithm in our model automatically applies to the model in [1], the same lower bounds apply. The I/O lower bounds proved in [1] are based only on routing concerns and thus hold for an arbitrarily powerful adversary, except in the case of sorting for the extreme case mentioned in Theorem 1 when M and B are extremely small, in which case the comparison model is used. *Thus the hard part of sorting in the nonextreme case is the routing of the records, not the determination of the records' order.* The well-known technique of key sorting [8], which attempts to reduce sorting to permutation routing by using a special-purpose method of determining the order of the records, is therefore not going to use significantly fewer I/Os than will general sorting algorithms.

The lower bound in Theorem 5 for standard matrix multiplication follows by taking the bound for the case $P = 1$ in [15] and dividing by P. The algorithms that meet the lower bounds of Theorems 1–5 are described and analyzed in the following sections.

The previously best way to sort with multiple disks when $P' = 1$ was the following combination of the well-known techniques of disk striping and two-way merge sort: The read/write heads of the P disks are synchronized, so that during each I/O all the disk drives access the same track number on their respective disks. This "striping" of the data on the disks effectively reduces the multiple disks to only one logical disk, but with a larger block size $B' = PB$. The number of I/Os needed for two-way merge sort with the one logical disk is

$$\Theta((N/B')\log(N/B')/\log(M/B')) = \Theta((N/PB)\log(N/PB)/\log(M/PB)).$$

This bound can be larger than the one in Theorem 1 by a multiplicative $\log(M/B)$ factor, which can be significant when P and PB are relatively large. For small values of P, striping is efficient in terms of I/O, within a constant factor of optimal.

5. Shuffle-Merge and Its Applications.

In this section we exploit a simple but useful merging operation called *shuffle-merge* that can be used to achieve the optimal I/O bounds mentioned in Theorems 2 and 4 for the problems of FFT, permutation networks, and matrix transposition. The algorithms, which consist of a series of shuffle-merges, are the ones described in [1], except that the disk placement of the blocks of the merged runs must be done in a staggered way so that the merging in the next pass can be done using full parallelism.

Without loss of generality, we assume for simplicity of exposition that N, M, P, and B are powers of 2. The operation of *shuffle-merge* consists of performing a perfect shuffle [16] on the elements of M/B runs of r records each, and the result is a single shuffled run of rM/B elements. Pictorially, suppose the sorted runs initially look like this:

$$\begin{array}{lcccc}
\text{Run } 1: & a_1^1 & a_2^1 & \cdots & a_r^1 \\
\text{Run } 2: & a_1^2 & a_2^2 & \cdots & a_r^2 \\
\vdots & & \vdots & & \\
\text{Run } \dfrac{M}{B}: & a_1^{M/B} & a_2^{M/B} & \cdots & a_r^{M/B}
\end{array}$$

After the perfect shuffle, a single shuffled run remains:

$$a_1^1 \ a_1^2 \ \cdots \ a_1^{M/B} \ a_2^1 \ a_2^2 \ \cdots \ a_2^{M/B} \ \cdots \ a_r^1 \ a_r^2 \ \cdots \ a_r^{M/B}.$$

It is easy to do shuffle-merges and take full advantage of parallel block transfer, if the input runs are blocked and the blocks are staggered with respect to one another on the disk, so that in a single I/O we can read the next track from each of the next P runs. For example, it suffices if the kth block of records from the ith run (consisting of records $a_{(k-1)B+1}^i, \ldots, a_{kB}^i$) is stored on track $(i-1)\lceil r/PB \rceil + \lceil k/P \rceil$ of disk $\mathscr{D}_{1+((k+i-2) \bmod P)}$. (If $r < PB/2$, then this placement can be modified so that more than one run is packed per track.) The algorithm consists of a series of parallel block transfers. On reads $(k-1)M/PB + 1, \ldots, kM/PB$, we bring into internal memory the kth block from each of the M/B runs. The records are shuffled appropriately in internal memory and then written to the disks. The total number of I/Os for the entire shuffle-merge is $O(rM/PB)$, which is best possible, since each record is read once from disk and written once to disk, making full use of parallelism and blocking.

Permutation Networks and FFT. Every permutation of N elements can be realized by three passes through an FFT network, by an appropriate setting of the switches in the FFT network that depends on the permutation [20]. So we can get optimal I/O strategies for an FFT-based permutation network by getting optimal I/O strategies for FFT digraphs.

The FFT diagraph is defined in Section 3. For simplicity, we assume that $\log M$ divides $\log N$ evenly. We divide the N records into N/M groups of M contiguous records. Each group corresponds to a set of rows of the FFT digraph whose nodes have links to only each other in the next $\log M$ levels of the FFT digraph. For each of N/M I/Os, we input the M records in a group, pebble forward in the FFT digraph $\log M$ levels, and then write the group back to the disks, in a staggered way. Afterwards, a series of shuffle-merges are done to realign the records into new groups of size M so that pebbling of each group can proceed for the next $\log M$ levels, as shown in Figure 4. This continues until the entire FFT digraph is pebbled. The above algorithm stops and performs a series of shuffle-merges $\log N/\log M$ times. Each series consists of $\max\{1, \log_{M/B} \min\{M, N/M\}\}$ shuffle-merges, each requiring $O(N/PB)$ I/Os. Thus the total number of I/Os used is

$$\Theta\left(\frac{N}{PB}\frac{\log N}{\log M}\left(1 + \log_{M/B} \min\left\{M, \frac{N}{M}\right\}\right)\right),$$

which can be shown by some algebraic manipulation to equal the bound given in Theorem 2.

The bound on internal processing time in Theorem 2 follows from the fact that the processing is done one memoryload at a time. Each memoryload accounts for $\Theta(M/PB)$ I/Os, and $O((M \log M)/P')$ time is used to do the internal processing of each memoryload.

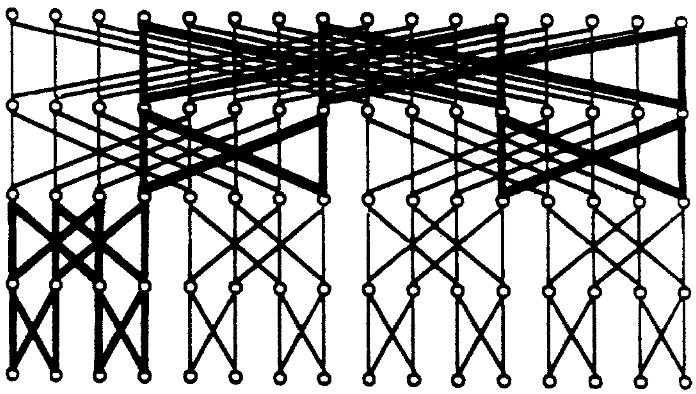

Fig. 4. Decomposition used for optimal pebbling of the FFT digraph, for $N = 16$, $M = 4$. The M pebbles can be slid forward $\log M$ levels before the pebbles have to be regrouped.

Matrix Transposition. Let us denote the B records that end up in the same block in the transposed matrix as a "target group." Initially, in the original untransposed matrix, several members of a given target group may be in the same block. We call these members a "target subgroup." Each target subgroup initially has size

$$x = \begin{cases} 1 & \text{if } B < \min\{p, q\}, \\ \dfrac{B}{\min\{p, q\}} & \text{if } \min\{p, q\} \le B \le \max\{p, q\}, \\ \dfrac{B^2}{N} & \text{if } \max\{p, q\} < B. \end{cases} \tag{1}$$

The transposition algorithm consists of a series of shuffle-merges. Records in the same target subgroup remain together throughout the course of the algorithm. In each pass we merge together sets of M/B target subgroups, thus increasing the size of the resulting target subgroups by a factor of M/B. The number of passes is

$$\left\lceil \log_{M/B} \frac{B}{x} \right\rceil, \tag{2}$$

each requiring $O(N/PB)$ I/Os. The upper bound for I/O in Theorem 4 follows by substituting the three cases of (1) into (2).

Each memoryload accounts for $\Theta(M/PB)$ I/Os, and $O((M \log P')/P')$ time is used to do the internal processing of each memoryload, thus yielding the bound on internal processing time in Theorem 4.

6. External Sorting and Permuting.

In this section we present and analyze the optimal algorithms for sorting and permuting on the two-level memory model, which achieve the bound listed in Theorems 1 and 3. We assume for simplicity of exposition that N, M, P, and B are powers of 2; we show below that this assumption does not result in any loss of generality. For sorting we also make the simplifying assumption that all key values are distinct. This assumption is satisfied, for example, if we append to the key field of each record the original memory location of the record.

Permuting records is a special case of sorting. The bounds for sorting and permuting given in Theorems 1 and 3 are the same, except when the internal memory size M and block size B are extremely small with respect to the file size N. In the latter case, permuting can be done using $O(N/P)$ I/Os by a modification of the sorting technique we discuss below, so we restrict our attention to the sorting problem. The application of the sorting method to the special case of permuting is discussed in Section 6.5.

The FFT and matrix transposition algorithms described in the previous section were easy to implement using an optimal number of I/Os, because the merging pattern in each pass was predetermined; it consisted of a series of shuffle-merges.

This made it easy to distribute the records onto the disks so that the merges in the next pass accessed the records in a balanced fashion among all the disks. However, this does not seem applicable to sorting. When merge sort is used for external sorting, the merges in each pass are not in general perfect shuffles. When the merges are not perfect shuffles, it is difficult to know how to distribute the records onto the disks so as to guarantee balanced access to the disks in the next merge pass. An interesting question of whether merge sort can be implemented using the optimal number of I/Os is discussed in Section 8.

In order to get an optimal sorting algorithm we use the following practical randomized approach, which is a recursive distribution sort:

1. If $N \leq M$, we sort the file internally. Otherwise we do the following steps:
2. [Find partitioning elements.] We deterministically find $S - 1$ partitioning elements $b_1, b_2, \ldots, b_{S-1}$ that break the file into S roughly equal-sized buckets. The parameter S will be defined shortly; it is always small enough so that the partitioning elements can be stored easily in internal memory. For convenience, we define the dummy partitioning elements $b_0 = -\infty$ and $b_S = +\infty$. The jth bucket consists of all the records R in the file whose key values are in the range
$$b_{j-1} \leq key(R) < b_j.$$
3. [Partition into buckets.] We partition the file into buckets based upon the partitioning elements and distribute the records in each bucket evenly among the P disks.
4. [Recurse.] We sort each bucket by applying the sorting algorithm recursively. (With high probability, the records in each bucket are distributed evenly among the P disks, and thus they can be read into internal memory with $O(N/SPB)$ I/Os.) The output of the sorting algorithm is the concatenation of the sorted buckets.

The partitioning in Step 3 is done in one of two ways, which we call Phase 1 and Phase 2. Phase 1, which is described in detail in Section 6.1, is used for the partitioning when $N \geq \sqrt{MBP}/\ln(M/B)$. It can be thought of intuitively as a hashing approach to distribute the blocks of each bucket among the disks. It works effectively when the "hash function" distributes the records evenly, and by analogy to the maximum bucket occupancy problem in hashing [18], the distribution is even when the expected number of blocks per disk for each bucket is at least a logarithmic amount. However, if N is not much larger than M, the distribution using the hashing approach can be quite uneven, resulting in non-optimal performance. In the latter case, when $M < N < \sqrt{MBP}/\ln(M/B)$, the partitioning is done by Phase 2, which is described in Section 6.2. Phase 2 uses a partitioning technique motivated by a different instance of the hashing problem and works with overwhelming probability. After the Phase 2 partitioning, each bucket will have at most M records and can be sorted internally, as described in Section 6.3.

An alternative to the Phase 2 technique for small N is the deterministic approach based on Leighton's Columnsort algorithm [9], as mentioned in [1] and described in detail in Theorem 6 of [12]. However, we use Phase 2 here because it complements Phase 1 nicely in approach and has fairly small constant factors.

As noted above, we assume for simplicity that N, M, P, and B are powers of 2. We also choose $S - 1$ to be a power of 2. The assumption on M, P, and B, whose values do not change during the course of the algorithm, clearly does not affect the generality of the algorithm. We can make N a power of 2 at each level of recursion by appending to the file hypothetical dummy records with key value $+\infty$. These dummy records do not need to be written to the buckets, so the cumulative sizes of the buckets is not affected. The running time increases as a result by at most a small constant factor.

DEFINITION 2. We denote the S buckets by $\mathscr{S}_1, \mathscr{S}_2, \ldots, \mathscr{S}_S$. The number of records in the file that belong to bucket $\mathscr{S}_j$ is denoted N_j. In our two-level model with parallel block transfer, we can look at only M records at a time, so partitioning will be done one memoryload at a time. We denote the ith memoryload by $\mathscr{M}_i$. A *write cycle* is defined as the collection of P blocks that we write to the disks concurrently in a single I/O. We denote write cycles by $\mathscr{W}_1, \mathscr{W}_2, \ldots$. Read cycles are defined analogously.

For the time being, let us assume that we can deterministically compute in Step 2 the approximate partitioning elements $b_1, b_2, \ldots, b_{S-1}$, using $O(N/PB)$ I/Os. (The algorithm and the analysis for computing the partitioning elements are given in Section 6.4.) For Phase 1 we set $S \approx \sqrt{M/B}/\ln^2(M/B)$; in particular, we set $S - 1$ to be a maximum of 1 and the largest power of 2 that is $\leq \sqrt{M/B}/\ln^2(M/B)$. We show later in Lemma 3 that

$$\frac{N}{2(S-1)} < N_j < \frac{3N}{2(S-1)}. \tag{3}$$

For Phase 2 we set $S = 2N/M + 1$; we show later in Lemma 4 that

$$\tfrac{3}{8}M < N_j < \tfrac{5}{8}M. \tag{4}$$

The upper bound for sorting in Theorem 1 follows from the following bound, which is the main result of this paper:

THEOREM 6. *The number of I/Os used by the above distribution sort algorithm to sort N records is*

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

and the internal processing time is $O(N\log(N/B)(\log M)/P'\log(M/B))$, *with over-*

whelming probability. In particular, the probability that the number of I/Os or the internal processing time is more than l times the average is exponentially small in $l(\log l)\log(M/B)$.

PROOF. We define $T(N)$ to be the number of I/Os used to sort N records and $T_1(N)$ to be the number of I/Os used for all the calls to Phase 1. We see from Theorem 7 that with high probability Phase 1 uses $O(N/PB)$ I/Os to partition N records and to store each bucket evenly across the disks, so that the buckets can be retrived one-by-one in the next level of recursion with a total of $O(N/PB)$ I/Os. The above construction gives us

$$T_1(N) = \sum_{1 \le j \le S} T_1(N_j) + O\left(\frac{N}{PB}\right) \tag{5}$$

with high probability. In particular, from Theorem 7, the probability that the quantity represented by the big-oh term in (5) is more than lN/PB is exponentially small in $l(\log l)\log(M/B)$. In Phase 1 we set $S \approx \sqrt{M/B}/\ln^2(M/B)$; by (3) we have $N_j \le 3N/2(S-1)$. Substituting this bound into (5) and iterating the recurrence until $N \le M$, by which time Phase 1 certainly ends, we get

$$T_1(N) = O\left(\frac{N}{PB}\,\frac{\log(N/B)}{\log(M/B)}\right) \tag{6}$$

with high probability. In all the instances of partitioning during the recursive levels of Phase 1, imbalance by more than a factor of l occurs *independently* with exponentially small probability. By convexity arguments, we can combine these probability bounds and bound the probability that $T_1(N)$ is more than l times the expression in the big-oh term in (6) by a quantity exponentially small in $l(\log l)\log(M/B)$.

In Theorem 8 we show that with high probability Phase 2 uses $O(N/PB)$ I/Os in order to perform the last level of partitioning. The remaining buckets each contain at most M records and can be sorted internally. This gives us

$$T(N) = O\left(\frac{N}{PB}\,\frac{\log(N/B)}{\log(M/B)}\right) \tag{7}$$

with high probability. Since the partitioning in Phase 2 is independent of those in Phase 1, we can bound the probability that $T(N)$ is more than l times the expression in the big-oh term in (7) by a quantity exponentially small in $l(\log l)\log(M/B)$.

The bound on internal processing time follows because the internal processing is done one memoryload at a time. Each memoryload accounts for $\Theta(M/PB)$ I/Os, and $O((M\log M)/P')$ time is used for the internal sorting, partitioning, and overhead of each memoryload. □

6.1. Phase 1. We use Phase 1 to partition the records of a file of N records when $N \geq \sqrt{MBP}/\ln(M/B)$. The number S of partitions is approximately $\sqrt{M/B}/\ln^2(M/B)$; in particular, we set $S - 1$ to be the maximum of 1 and the largest power of 2 that is $\leq \sqrt{M/B}/\ln^2(M/B)$. In order to read a file into internal memory using full parallelism, the records of the file must be evenly distributed over the disks, as a result of the previous pass of Phase 1. This is the crux of the problem. We show in Theorem 7 that Phase 1 does the partitioning using $O(N/PB)$ I/Os.

We read the records of the file into internal memory, one memoryload at a time. We assign the records to buckets based upon the partitioning elements and organize the records so that records in the each bucket are continguous in internal memory. We then write the records in each bucket of the memoryload to the disks, using full parallelism. We use a randomized approach to distribute the records. The main result of this subsection is showing that the requirement $N \geq \sqrt{MBP}/\ln(M/B)$ assures with high probability that the records of a bucket (among all the memoryloads) will be spread out evenly among the disks.

In order to link together the records in each bucket (to allow fast retrieval of the bucket in the next level of recursion), we need to remember the last track on each disk where a block belonging to that bucket was written; we store these pointers in internal memory. In order to reduce the number of pointers so that they can be kept in internal memory, we "cluster" the disks into C logical clusters, as shown in Figure 5. We set C to be $\min\{P, S\}$.

DEFINITION 3. A *cluster* is a logical grouping of consecutive disks. The C clusters are denoted $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_C$. The P/C disks in the kth cluster $\mathscr{C}_k$ are denoted $\mathscr{D}_{k,1}, \mathscr{D}_{k,2}, \ldots, \mathscr{D}_{k,P/C}$. The ith *track of a cluster* refers collectively to the ith tracks of all the disks that comprise the cluster. Records are written to the disks in cluster-size units of P/C blocks, which we call a *group*.

Algorithm—Phase 1. Let $last\_disk_{j,k}$ and $last\_track_{j,k}$ represent the last disk and the last track, respectively, written to in cluster $\mathscr{C}_k$ by bucket $\mathscr{S}_j$. Let $next\_track_k$ represent the first track on $\mathscr{C}_k$ that has not been assigned to a bucket. We initialize $last\_disk_{j,k} := last\_track_{j,k} := 0$ and $next\_track_k := 1$.

The file is processed memoryload by memoryload. For each $1 \leq i \leq N/M$, the ith memoryload is brought into internal memory. The records are partitioned into buckets, based upon the partitioning elements. The records in each bucket are formed into blocks, and the blocks within a bucket are formed into groups of size P/C, except possibly the last group which might be only partially filled. We choose

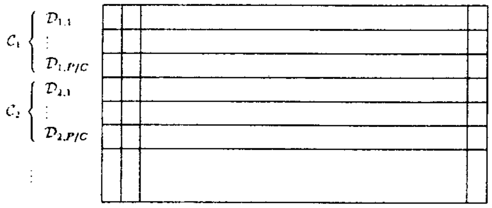

Fig. 5. Decomposition of the P disks into C clusters.

C groups to be written during this write cycle, and we assign these groups to clusters by choosing a random permutation of $\{1, 2, \ldots, C\}$. This is repeated C groups at a time until the memoryload is written. (This is the only place where randomness is used in Phase 1.)

What remains is to assign the blocks in a group to the disks in a cluster. In each group we have a maximum of P/C blocks. We do not want to have empty tracks on the disks, so we cycle through the disks in the cluster. Let us assume that a group belonging to bucket $\mathscr{S}_j$ is assigned to cluster $\mathscr{C}_k$. We assign the first $P/C - \mathit{last\_disk}_{j,k}$ blocks to disks $\mathit{last\_disk}_{j,k} + 1, \ldots, P/C$ on track $\mathit{last\_track}_k$; we assign the remaining blocks, if any, to disks $1, 2, \ldots$ on track $\mathit{next\_track}_k$. We then update the value of $\mathit{last\_disk}_{j,k}$, and when the current track gets filled, we set $\mathit{last\_track}_{j,k}$ to $\mathit{next\_track}_k$ and increment $\mathit{next\_track}_k$.

For each memoryload we retain partially filled blocks in internal memory until they are completely filled, but groups are written to the disks even if they contain fewer than P/C blocks. Each time a group from bucket $\mathscr{S}_j$ is written to cluster $\mathscr{C}_k$, we fill up the last track written on that cluster for $\mathscr{S}_j$ before we start a new track; that is, once a bucket writes to a particular track of a given cluster, it will not write to another track of that cluster until the current track is completely filled. This has the effect of making each track of each cluster completely filled, except possibly for the last track of the cluster for each bucket.

In order for the recursion to work, we must link together the records of each bucket. This will be done with pointers being made part of the blocks when they are written to disk. Since records from a bucket are written as a group, we only have to save pointers for the groups. Also, since an entire track in a cluster is written to by only one bucket, the linking can be done by pointers in the block on the first disk of each track in the cluster. To do this, we have one "previous group" pointer for each track of each cluster, which we call *pg*. Each *pg* pointer links together the groups of a given bucket that are on a given cluster, in reverse order. If a block in a group from $\mathscr{S}_j$ is written to the first disk of a cluster $\mathscr{C}_k$, the *pg* pointer of that block is set to $\mathit{last\_track}_{j,k}$.

Once the assigning is done, we can write the chosen C groups to their assigned disk locations. When we finish processing the file, we save on the disks the pointers $\mathit{last\_disk}_{j,k}$ and $\mathit{last\_track}_{j,k}$, so that we can locate the records for each bucket during the next level of recursion.

```
$\mathit{last\_disk}_{j,k} := 0$ for all $j$, $k$;
$\mathit{next\_track}_k := 1$ for all $k$;
$\mathit{last\_track}_{j,k} := 0$ for all $j$, $k$;
**for** each memoryload of records $\mathscr{M}_i$ $(1 \le i \le N/M)$ **do**
  **begin**
  read $M_i$ into internal memory;
  partition the records into buckets based upon the partitioning elements;
  **for** each bucket $\mathscr{S}_j$ $(1 \le j \le S)$ **do**
    **begin**
    form the records into blocks of size $B$;
    form the blocks to groups of blocks of size $P/C$
    **end**;
    for each write cycle $\mathcal{W}_t$ do
      begin
      choose $C$ groups of blocks to be written in $\mathcal{W}_t$;
      assign the groups to clusters via a random permutation of $\{1, 2, \ldots, C\}$;
      { assign the blocks in each group to the disks in a cluster }
      for each cluster $\mathcal{C}_k$ $(1 \le k \le C)$ do
        begin
        let $\mathcal{S}_j$ be the bucket whose group is assigned to $\mathcal{C}_k$;
        for each disk $\mathcal{D}_{k,d}$ such that $P/C - last\_disk_{j,k} \le d \le P/C$ do
          schedule the next block to be assigned to $last\_track_{j,k}$ on $\mathcal{D}_{k,d}$;
        if still more blocks to be assigned then
          begin
          $temp\_pg := last\_track_{j,k}$;
          $last\_track_{j,k} := next\_track_k$;
          $next\_track_k := next\_track_k + 1$;
          for each disk $\mathcal{D}_{k,d}$ such that $1 \le d \le P/C - last\_disk_{j,k}$ do
            begin
            schedule the next block to be assigned to $last\_track_{j,k}$ on $\mathcal{D}_{k,d}$;
            if $d = 1$ then set the $pg$ pointer of block to $temp\_pg$
            end
          end;
        update $last\_disk_{j,k}$
        end;
      write the blocks in $\mathcal{W}_t$ to the desired disks
      end
    end;
  write pointers $last\_disk_{j,k}$ and $last\_track_{j,k}$, for all $j$, $k$
```

Analysis of Phase 1

THEOREM 7. *With overwhelming probability, each pass of Phase* 1 *uses* $O(N/PB)$ *I/Os to partiton a file of* N *records. In particular, the probability that the number of I/Os used is more than* l *times the average is exponentially small in* $l(\log l) \cdot \max\{\log(M/B), N/PDS\}$.

PROOF. The file is read into internal memory one memoryload at a time. The actual number of records read in each time might be less than a memoryload since the pointers (*last_disk*, *last_track*, and *next_track*) and the partially filled blocks are retained in memory during the partitioning process. There are $C(2S + 1)$ pointers needed; assuming each pointer does not exceed a record, the pointers take up $C(2S + 1)$ records. Since each of the S buckets might have a partially filled block of $B - 1$ records, the partially filled blocks can take up at most $S(B - 1)$ records, and we need space for the $S - 1$ partitioning elements. Therefore, at least $M - C(2S + 1) - SB + 1$ records can be read in. For convenience, we redefine M to be $M - C(2S + 1) - SB + 1$, so that a full memoryload can be read into or written from internal memory. This changes the value of M by at most a small constant factor.

Let Z be the number of I/Os required during the next pass of Phase 1 or Phase 2 to read in all the subfiles corresponding to the buckets formed from the current file by Phase 1. We want to show that

$$Z = O\left(\frac{N}{PB}\right)$$

with high probability. We do that by showing that

$$\Pr\left\{Z \geq l\,\frac{N}{PB}\right\}$$

is exponentially small in $l(\log l)\cdot\max\{\log(M/B), N/PBS\}$.

The number of inputs needed in the next pass of Phase 1 or Phase 2 in order to read into internal memory the subfile corresponding to some bucket formed by the current pass of Phase 1 is the maximum number of tracks devoted to that bucket among all the clusters. Let $X_{j,k}$ represent the number of tracks of cluster $\mathscr{C}_k$ that have been assigned to bucket $\mathscr{S}_j$. We have

$$Z = \sum_{1\leq j\leq S}\ \max_{1\leq k\leq C}\{X_{j,k}\}.$$

This gives us

$$\Pr\left\{Z \geq l\,\frac{N}{PB}\right\} = \Pr\left\{\sum_{1\leq j\leq S}\ \max_{1\leq k\leq C}\{X_{j,k}\} \geq l\,\frac{N}{PB}\right\}. \tag{8}$$

The max term in (8) is the difficult expression to analyze. We use the fact that $N \geq \sqrt{MBP}/\ln(M/B)$ in Phase 1 to show that the $X_{j,k}$ are very evenly distributed with respect to k. We have

$$\begin{aligned}\Pr\left\{\sum_{1\leq j\leq S}\ \max_{1\leq k\leq C}\{X_{j,k}\} \geq l\,\frac{N}{PB}\right\} &\leq \Pr\left\{\exists j,\ \max_{1\leq k\leq C}\{X_{j,k}\} \geq l\,\frac{N_j}{PB}\right\}\\ &\leq S\,\Pr\left\{\max_{1\leq k\leq C}\{X_{j',k}\} \geq l\,\frac{N_{j'}}{PB}\right\}\\ &\leq SC\,\Pr\left\{X_{j',1} \geq l\,\frac{N_{j'}}{PB}\right\},\end{aligned} \tag{9}$$

where N_j is the number of records in bucket $\mathscr{S}_j$, and j' is any value of j that *a priori* (before the partitioning elements are chosen) maximizes

$$\Pr\left\{\max_{1\le k\le C}\{X_{j,k}\}\ge lN_j/PB\right\}.$$

To bound (9), we use Chernoff's bound [7]:

LEMMA 1. *If X is a nonnegative random variable and $r \ge 0$ we have*

$$\Pr\{X \ge u\} \le \frac{E(e^{rX})}{e^{ru}}.$$

Before we apply Chernoff's bound, we must construct the appropriate scenario. We let g_t denote the number of clusters written to from bucket $\mathscr{S}_{j'}$ during write $\mathscr{W}_t$, for $1 \le t \le R$, where R is the total number of write cycles used in Phase 1. We have

$$(10)\qquad \sum_{1\le t\le R} g_t \le \frac{N_{j'}}{PB/C} + C = \frac{N_{j'}C}{PB} + C.$$

The extra "$+C$" term in (10) appears because the last track on each of the C clusters might be only partially filled. We define G_t to be the number of groups belonging to bucket $\mathscr{S}_{j'}$ in write $\mathscr{W}_t$ that are assigned to cluster $\mathscr{C}_1$. Because only one group can be written to any one cluster in a write cycle, G_t is restricted to the values 0 and 1. We have $\Pr\{G_t = 1\} = g_t/C$ and $\Pr\{G_t = 0\} = 1 - g_t/C$. Let $\mathscr{G}_{G_t}(z)$ be the probability generating function for G_t:

$$(11)\qquad \begin{aligned}\mathscr{G}_{G_t}(z) &= \Pr\{G_t = 0\}z^0 + \Pr\{G_t = 1\}z^1\\ &= 1 - \frac{g_t}{C} + \frac{g_t}{C}z\\ &= 1 + \frac{g_t}{C}(z-1).\end{aligned}$$

Let $\mathscr{G}_{X_{j',1}}(z)$ be the probability generating function for $X_{j',1}$. We can bound $X_{j',1}$ by the sum of independent random variables: $X_{j',1} \le G_1 + G_2 + \cdots + G_R$. For purposes of bounding (9), let us consider that $X_{j',1} = G_1 + G_2 + \cdots + G_R$. Using (11), we have

$$(12)\qquad \begin{aligned}\mathscr{G}_{X_{j',1}}(z) &= \mathscr{G}_{G_1+G_2+\cdots+G_R}(z)\\ &= \mathscr{G}_{G_1}(z) \times \mathscr{G}_{G_2}(z) \times \cdots \times \mathscr{G}_{G_R}(z)\\ &= \prod_{1\le t\le R}\left(1 + \frac{g_t}{C}(z-1)\right).\end{aligned}$$

By Chernoff's bound in Lemma 1, we have

$$\Pr\left\{X_{j',1} \geq l\frac{N_{j'}}{PB}\right\} \leq \frac{E(\exp(rX_{j',1}))}{\exp(rlN_{j'}/PB)} \tag{13}$$

for each $r \geq 0$. We can express the numerator in (13), using (12) and the definitions of expected value and probability generating function, as

$$\begin{aligned} E(\exp(rX_{j',1})) &= \sum_{t\geq 0} \Pr\{X_{j',1} = t\}e^{rt} \\ &= \mathscr{G}_{X_{j',1}}(e^r) \\ &= \prod_{1\leq t\leq R}\left(1 + \frac{g_t}{C}(e^r - 1)\right). \end{aligned} \tag{14}$$

To bound (14), we use the following lemma, which follows easily from convexity arguments.

Lemma 2. *If* $\sum_{1\leq i\leq R} a_i = Q$ *and* $a_i \geq 0$, *for* $1 \leq i \leq R$, *then* $\prod_{1\leq i\leq R} a_i$ *is maximized when* $a_1 = a_2 = \cdots = a_R = Q/R$.

By (10) and Lemma 2, we can maximize (14) by setting $g_t = N_{j'}C/RPB + C/R$ for each t. Thus

$$E(\exp(rX_{j',1})) \leq \prod_{1\leq t\leq R}\left(1 + \frac{(N_{j'} + PB)(e^r - 1)}{RPB}\right) = \left(1 + \frac{(N_{j'} + PB)(e^r - 1)}{RPB}\right)^R.$$

Substituting this bound into (13), we get

$$\Pr\left\{X_{j',1} \geq l\frac{N_{j'}}{PB}\right\} \leq \frac{(1 + ((N_{j'} + PB)(e^r - 1))/RPB)^R}{\exp(rlN_{j'}/PB)}.$$

Hence, by (8) and (9), we have

$$\Pr\left\{Z \geq l\frac{N}{PB}\right\} \leq SC\,\frac{(1 + ((N_{j'} + PB)(e^r - 1))/RPB)^R}{\exp(rlN_{j'}/PB)}. \tag{15}$$

By Lemma 3 in Section 6.4 and the Phase 1 bounds $N \geq \sqrt{MBP}/\ln(M/B)$ and $S - 1 \leq \sqrt{M/B}/\ln^2(M/B)$, we have

$$N_{j'} + PB = N_{j'}\left(1 + \frac{PB}{N_{j'}}\right) \leq N_{j'}\left(1 + \frac{2PB(S-1)}{N}\right) = N_{j'}(1 + \beta),$$

where $\beta = 2PB(S-1)/N \le 2/\ln(M/B)$. Substituting this bound into (15), we have

$$\Pr\left\{Z \ge l\frac{N}{PB}\right\} \le SC\frac{(1+(N_{j'}(1+\beta)(e^r-1))/RPB)^R}{\exp(rlN_{j'}/PB)}. \tag{16}$$

From the bound $(1+a)^b \le e^{ab}$, for $a > -1$, we can approximate the numerator in (16) and get

$$\begin{aligned}\Pr\left\{Z > l\frac{N}{PB}\right\} &\le SC\exp\left(\frac{N_{j'}(1+\beta)(e^r-1) - rlN_{j'}}{PB}\right)\\ &= SC\exp\left(((1+\beta)(e^r-1) - rl)\frac{N_{j'}}{PB}\right).\end{aligned} \tag{17}$$

Picking $r = \ln(l/(1+\beta))$, for $l \ge \beta + 1$, we have

$$\Pr\left\{Z > l\frac{N}{PB}\right\} \le SC\exp\left(\left(l - 1 - \beta - l\ln\frac{l}{1+\beta}\right)\frac{N_{j'}}{PB}\right). \tag{18}$$

By plugging into (18) the inequality $\ln(1+x) \ge x - x^2/2$, for $x \ge 0$, we get

$$\Pr\left\{Z > l\frac{N}{PB}\right\} \le SC\exp\left(\left(-\frac{(l-1-\beta)^2}{2(1+\beta)} + \frac{(l-1-\beta)^3}{2(1+\beta)^2}\right)\frac{N_{j'}}{PB}\right).$$

For example, when $1 + \beta \le l \le \frac{3}{2}$, we have the exponentially small bound

$$\Pr\left\{Z > l\frac{N}{PB}\right\} \le SC\exp\left(-\frac{(l-1-\beta)^2}{4(1+\beta)}\frac{N_{j'}}{PB}\right). \tag{19}$$

As l gets larger, we get directly from (18) the exponentially small bound

$$\Pr\left\{Z > l\frac{N}{PB}\right\} = O\left(SC\exp\left(-l(\log l)\frac{N_{j'}}{PB}\right)\right). \tag{20}$$

Note that $SC < M/B$ and by Lemma 3 we have $N_{j'}/PB \ge \frac{1}{2}N/PBS \ge \frac{1}{2}\ln(M/B)$; thus the exponential term in (20) gets smaller exponentially more quickly than the SC term grows, even for relatively small l, and, in the early passes of Phase 1, $N_{j'}/PB$ can be substantially larger than $\ln(M/B)$.

6.2. Phase 2. In Phase 2 we want to sort $N < \sqrt{MBP}/\ln(M/B)$ records in one pass of the distribution sort using $O(N/PB)$ I/Os, which is optimal. We use $S = 2N/M + 1$ partitions. This range of values of N is the case that cannot be

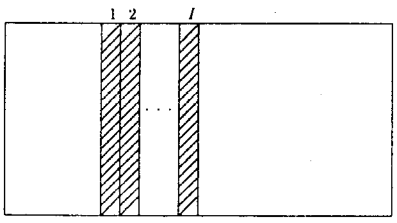

Fig. 6. The shaded tracks collectively represent a logical track.

handled by Phase 1, since when N is relatively small the records in each bucket will generally not be distributed evenly among the clusters. We want to guarantee that each bucket will consist of at most M records so that no further partitioning is needed, and we want to distribute the blocks of each of the S buckets evenly over the disks.

Before we give the algorithm, we first define the notions of "logical track" and "diagonal":

Definition 4. A *logical track* is defined as $I = M/PB$ consecutive tracks that are accessed in I consecutive I/Os. When accessing the ith logical track, we actually access tracks $I(i-1)+1, \ldots, Ii$. This is shown in Figure 6.

Definition 5. The *ith diagonal*, for $1 \le i \le N/M$, is defined as the memoryload of M/B blocks in which the first set of M^2/BN blocks consists of the ith logical track of disks $\mathscr{D}_1, \ldots, \mathscr{D}_{MP/N}$, the second set of M^2/BN blocks consists of the $(i+1)$st logical track of disks $\mathscr{D}_{MP/N+1}, \ldots, \mathscr{D}_{2MP/N}$, and so on, wrapping back to $i = 1$ when i exceeds N/M. Diagonal 1 is illustrated in Figure 7.

It follows from the condition $N < \sqrt{MBP}/\ln(M/B)$ of Phase 2 that $1 \le MP/N \le P$, and hence each diagonal is well defined. Every block in the file is a part of a unique diagonal, and every diagonal contains the same number of blocks from each logical track.

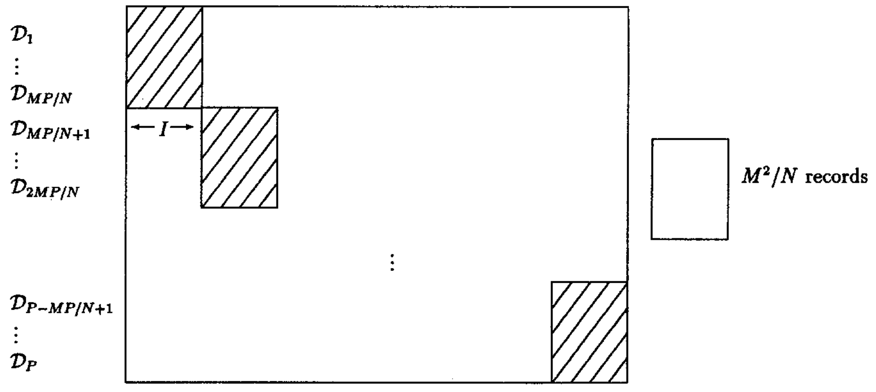

Fig. 7. The shaded areas collectively represent diagonal 1.

Our algorithm consists of two passes:

Pass 1. We scramble the records, memoryload by memoryload, and write them back to the disks. (This is where we use randomness in Phase 2.) This step can be done concurrently with the choosing of the partitioning elements.

Pass 2. We read in the file, one diagonal (memoryload) at a time. For each memoryload, we partition the records into buckets, based upon the partitioning elements. (The number of records in each bucket of a memoryload will be evenly distributed with high probability.) We write one block to disk from each bucket, cycling through the tracks on disk; we repeat this process until all the records of the memoryload are written.

At the end of Pass 2, each bucket will contain at most M records; the sorting can then conclude with a final series of internal sorts.

It is convenient to think of Phase 2 as an instance of the maximum occupancy problem in hashing, but different from the instance we used for motivation of Phase 1 in Section 6.1, in which we scrambled buckets randomly among the disks. Here we consider, for each separate diagonal, scrambling the M records of the diagonal randomly among the S buckets. The upper bound $N < \sqrt{MBP}/\ln(M/B)$ guarantees that each diagonal read during Pass 2 contains some number of blocks from each of the N/P logical tracks; the random scrambling of the logical tracks in Pass 1 means that any given diagonal can contain any given record in the file. Since the number M of records per diagonal is substantially more than the number S of buckets, the analogy to hashing suggests for each diagonal that the distribution of the records in the buckets will very likely be even. This general intuition is verified formally for the actual problem at hand in Theorem 8.

Algorithm—Pass 1 of Phase 2. For simplicity of exposition, we assume that the file resides in packed format across the disks, track by track. In reality the file formed by Phase 1 is not packed, but it can be read into internal memory using full parallelism, so our assumption is valid.

We read in all the records, processing them one logical track (memoryload) at a time. For each memoryload, we randomly permute the records in internal memory. Next we form blocks, based upon the permuted ordering, and we write the blocks back to the logical track from which they were read.

```
for each memoryload of records $\mathcal{M}_i$ $(1 \leq i \leq N/M)$ do
  begin
  read $\mathcal{M}_i$ from the $i$th logical track into internal memory;
  randomly permute the $M$ records;
  write $\mathcal{M}_i$ to the $i$th logical track
  end
```

Algorithm—Pass 2 of Phase 2. We read in all of the records, one diagonal (memoryload) at a time. The records for each memoryload are partitioned into buckets based upon the partitioning elements and then written to the disks as follows: The records within each bucket are formed into blocks. We then write the blocks to the disks, including partially filled blocks, in the following order:

block 1 of $\mathscr{S}_1$,
block 1 of $\mathscr{S}_2$,
$\vdots$
block 1 of $\mathscr{S}_S$,
block 2 of $\mathscr{S}_1$,
block 2 of $\mathscr{S}_2$,
$\vdots$
block 2 of $\mathscr{S}_S$,
and so on.

If one of the buckets runs out of blocks before the others, dummy blocks for that bucket are written. The disks are written to, track by track, in the cyclical order $1, 2, \ldots, P$.

```
$k := 1$;
for each diagonal $\mathscr{M}_i$ $(1 \le i \le N/M)$ do
    begin
    read $\mathscr{M}_i$ from the $i$th diagonal into internal memory;
    partition the $M$ records into buckets based upon the partitioning elements;
    form the records into blocks of size $B$;
    while a nonempty bucket remains do
        for $j := 1$ to $S$ do
            begin
            schedule next block from $\mathscr{S}_j$ to be written on next available track of $\mathscr{D}_k$,
                assigning a dummy block if $\mathscr{S}_j$ is empty;
            $k := (k \bmod P) + 1$
            end;
    write the memoryload to the desired disks
    end
```

By our assumptions, N, M, and P are powers of 2, and thus $S = 2N/M + 1$ is relatively prime to P. If, however, the least common multiple d of S and P were to satisfy $d < SP$, then the above code would have to be modified so that after each d/S write cycles the order that the disks are written to would be cyclically shifted by one. The shifting would prevent each bucket from being written to only a small subset of the disks.

Analysis of Phase 2

THEOREM 8. *With overwhelming probability, Phase 2 of the distribution sort algorithm uses $O(N/PB)$ I/Os to complete the sort of N records. The probability that the number of I/Os used is more than l times the average is exponentially small in $l(\log l)\log^2(M/B)$.*

PROOF. The actual number of records read in each memoryload might be less than M records, since in Pass 2 the $S - 1 = 2N/M$ partitioning elements are retained in memory. The maximum size of each bucket formed must be less than M in order for the sorting to be completed by a series of internal sorts in the next pass, as described in Section 6.3, since the final sorting pass requires that a track/disk pointer and partially filled block be retained in memory. At most $M - S - 1$ records can be read per memoryload during Pass 2, and it is possible that only $M - 1 - (B - 1) = M - B$ records can be read per memoryload during the final internal sorting step, assuming that the disk/track pointer does not exceed one record. For convenience, we redefine M to be $M - B - S$ so that a full memoryload can be read into or written from internal memory in Pass 2. This changes the value of M by at most a small constant factor.

The reading of the records in Pass 2, the reading and writing of the records in Pass 1, and the writing of the records in the final pass described in Section 6.3 use $O(N/PB)$ I/Os. We can restrict our attention to the remaining set of I/Os we have to consider, namely, the write operations in Pass 2 and the read operations in the final pass. These two quantities are equal, so we restrict our attention to the number of write operations in Pass 2. Let Z be the number of I/Os needed to write all the records, one bucket at a time, in Pass 2. We want to show that

$$Z = O\left(\frac{N}{PB}\right) \tag{21}$$

with high probability. In particular, we shall show that

$$\Pr\left\{Z \geq l\,\frac{N}{PB}\right\} \tag{22}$$

is exponentially small in $l(\log l)\log^2(M/B)$.

Let α be the number of times a set of S blocks is written in Pass 2, and let $Y_{i,j}$ represent the number of records found in memoryload $\mathcal{M}_i$ belonging to bucket $\mathcal{S}_j$. We have

$$\alpha = \sum_{1 \leq i \leq N/M} \max_{1 \leq j \leq S} \left\{\left\lceil \frac{Y_{i,j}}{B} \right\rceil\right\}. \tag{23}$$

Since the last write for each bucket may be partial we get the bound

$$Z \leq S\left\lceil \frac{\alpha}{P} \right\rceil. \tag{24}$$

Note that Z is expressed in (24) in terms of α, which by (23) is the sum over i of $\max_{1 \leq j \leq S}\{\lceil Y_{i,j}/B \rceil\}$. The hard part of the analysis is showing that

$\max_{1 \le j \le S}\{\lceil Y_{i,j}/B \rceil\}$ is with very high probability comparable with the average value of each $\lceil Y_{i,j}/B \rceil$. We have

$$(25) \qquad \Pr\left\{Z \ge l\frac{N}{PB}\right\} \le \Pr\left\{S\left\lceil\frac{\alpha}{P}\right\rceil \ge l\frac{N}{PB}\right\}$$

$$\le \Pr\left\{\frac{\alpha}{P} \ge l\frac{N}{PBS} - \frac{P-1}{P}\right\}$$

$$= \Pr\left\{\alpha \ge l\frac{N}{BS} - (P-1)\right\}$$

$$= \Pr\left\{\sum_{1 \le i \le N/M} \max_{1 \le j \le S}\left\{\left\lceil\frac{Y_{i,j}}{B}\right\rceil\right\} \ge l\frac{N}{BS} - (P-1)\right\}.$$

Let us define j' to be any value of j that *a priori* (before the partitioning elements are chosen) maximizes the expression $\Pr\{\lceil Y_{1,j}/B \rceil \ge lM/BS - (P-1)M/N\}$. We can bound (25) as follows:

$$(26) \qquad \Pr\left\{\sum_{1 \le i \le N/M} \max_{1 \le j \le S}\left\{\left\lceil\frac{Y_{i,j}}{B}\right\rceil\right\} \le l\frac{N}{BS} - (P-1)\right\}$$

$$\le \frac{N}{M}\Pr\left\{\max_{1 \le j \le S}\left\{\left\lceil\frac{Y_{1,j}}{B}\right\rceil\right\} \ge l\frac{M}{BS} - \frac{(P-1)M}{N}\right\}$$

$$\le \frac{SN}{M}\Pr\left\{\left\lceil\frac{Y_{1,j'}}{B}\right\rceil \ge l\frac{M}{BS} - \frac{(P-1)M}{N}\right\}$$

$$\le \frac{SN}{M}\Pr\left\{Y_{1,j'} \ge l\frac{M}{S} - \frac{(P-1)MB}{N} - (B-1)\right\}.$$

Let us define l' so that

$$(27) \qquad l'\frac{M}{S} = l\frac{M}{S} - \frac{(P-1)MB}{N} - (B-1).$$

Substituting the Phase 2 bound $S = 2N/M + 1$ into (27), we find after some algebraic manipulation that $l' > l - 2PB/M - 2NB/M^2$. Since $N < \sqrt{MBP}/\ln(M/B)$ in Phase 2, it follows that l' is at most a small constant amount less than l; in particular, we have $l' > l - 2 - 2/\ln(M/B)$. Substituting this value of l' into (26) we get

$$(28) \qquad \Pr\left\{Z \ge l\frac{N}{PB}\right\} \le \frac{SN}{M}\Pr\left\{Y_{1,j'} \ge l'\frac{M}{S}\right\}.$$

Let $Y_{i,j,k}$ represent the number of records found in memoryload $\mathcal{M}_i$ belonging to bucket $\mathcal{S}_j$ read from the kth logical track. In particular we have $\sum_{1 \le k \le N/M} Y_{1,j',k} = Y_{1,j'}$. Let μ_k represent the expected value of $Y_{1,j',k}$. By Lemma 4, each bucket contains at most $\frac{5}{8}M$ records of the file, and there are N/M memoryloads. Thus, we have

$$\sum_{1 \le k \le N/M} \mu_k = E(Y_{1,j'}) \le \frac{\frac{5}{8}M}{N/M} = \frac{5}{8}\frac{M^2}{N}.$$

Let $T_{j,k}$ be the number of records belonging to bucket $\mathcal{S}_j$ on the kth logical track. We have $\mu_k = T_{j',k}M/N$. We want to bound $\Pr\{Y_{1,j'} \ge l'M/S\}$. We do that by considering two cases: (1) small μ_k and (2) large μ_k. The two cases are determined by comparing μ_k with the average size that $Y_{1,j',k}$ would be if all the $T_{j',k}$ terms were equal to M^2/N. Let $\delta = \frac{8}{39}l' - \frac{15}{39}$. We have

$$(29) \qquad \Pr\left\{Y_{1,j'} \ge l'\frac{M}{S}\right\} \le \sum_{\substack{1 \le k \le N/M \\ \mu_k < M^3/N^2}} \Pr\left\{Y_{1,j',k} \ge \mu_k + \delta\frac{M^3}{N^2}\right\} + \sum_{\substack{1 \le k \le N/M \\ \mu_k \ge M^3/N^2}} \Pr\{Y_{1,j',k} \ge \mu_k + \delta\mu_k\}.$$

The above bound (29) holds since

$$\begin{aligned}\sum_{\substack{1 \le k \le N/M \\ \mu_k < M^3/N^2}} \left(\mu_k + \delta\frac{M^3}{N^2}\right) + \sum_{\substack{1 \le k \le N/M \\ \mu_k \ge M^3/N^2}} (\mu_k + \delta\mu_k) &\le \left(\tfrac{5}{8} + \tfrac{13}{8}\delta\right)\frac{M^2}{N} \\ &= \left(\tfrac{5}{4} + \tfrac{13}{4}\delta\right)\frac{M}{S-1} \\ &< \left(\tfrac{5}{4} + \tfrac{13}{4}\delta\right)\frac{3}{2}\frac{M}{S} \\ &= l'\frac{M}{S}.\end{aligned}$$

The probability term in (28) is expressed in (29) as a sum of tails of distributions of $Y_{1,j',k}$, where the starting point of each tail is sufficiently far from the mean μ_k so that the result is exponentially small, as we shall see. Intuitively, in order to get a small bound on the probability term, when μ_k is small the tail should start at some absolute distance from μ_k, and when μ_k is larger the tail should start at some multiple of μ_k.

We want to get tight upper bounds for the tails of the probability distribution of $Y_{1,j',k}$ listed in the summations in (29). Both summands have the form $\Pr\{X \ge \mu + v\}$, where $X = Y_{1,j',k}$, $\mu = \mu_k$ is the mean of X, and v is a positive value. Let $L = M^2/N$ be the number of records read from the ith logical track by any memoryload. The random variable $X = Y_{1,j',k}$ has the hypergeometric probability distribution

$$\Pr\{X = t\} = \frac{\binom{T_{j',k}}{t}\binom{M - T_{j',k}}{L-t}}{\binom{M}{L}} \tag{30}$$

with mean

$$\mu = \frac{T_{j',k}L}{M} = \frac{T_{j',k}M}{N}. \tag{31}$$

One approach to bounding $\Pr\{X \geq \mu + v\}$ is to use Chernoff-type bounds, as we did in Section 6.1, except in this case we would have to use a two-variable version involving the supergenerating function of (30), which has a simple closed form. A simpler approach is to consider the ratio $R(t)$ defined by

$$\frac{\Pr\{X = t + 1\}}{\Pr\{X = t\}}.$$

We have

$$\begin{aligned} \Pr\{X \geq \mu + v\} &= \Pr\{X = \mu + v\} + \Pr\{X = \mu + v + 1\} + \cdots \\ &= \Pr\{X = \mu\}\,\frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}} \\ &\quad + \Pr\{X = \mu + 1\}\,\frac{\Pr\{X = \mu + v + 1\}}{\Pr\{X = \mu + 1\}} + \cdots. \end{aligned} \tag{32}$$

Using (30), it is easy to see that $R(\mu + v)$ is monotone decreasing in v, and hence we have $\Pr\{X = \mu + v\}/\Pr\{X = \mu\} > \Pr\{X = \mu + v + 1\}/\Pr\{X = \mu + 1\}$. Substituting this bound into (32), we get

$$\begin{aligned} \Pr\{X \geq \mu + v\} &\leq \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}}\,(\Pr\{X = \mu\} + \Pr\{X = \mu + 1\} + \cdots) \\ &\leq \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}}. \end{aligned} \tag{33}$$

Note that we can write $\Pr\{X = \mu + v\}$ in the following form:

$$\text{(34)}\qquad \Pr\{X = \mu + v\} = \Pr\{X = \mu\} \frac{\Pr\{X = \mu + 1\}}{\Pr\{X = \mu\}} \frac{\Pr\{X = \mu + 2\}}{\Pr\{X = \mu + 1\}} \cdots \times \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu + v - 1\}} = \Pr\{X = \mu\} \prod_{0 \le t \le v-1} R(\mu + t).$$

Using (34) and (33), we get

$$\text{(35)}\qquad \Pr\{X \ge \mu + v\} \le \prod_{0 \le t \le v-1} R(\mu + t).$$

From (30), (31), the definition of $R(t)$, and the bounds $0 \le t \le T_{j',k} \le M$, we find after some algebraic manipulation that

$$\text{(36)}\qquad R(\mu + t) = \frac{(T_{j',k} - (\mu + t))(L - (\mu + t))}{(\mu + t + 1)(M - T_{j',k} - L + \mu + t + 1)} \le \frac{1}{1 + t/(\mu + 1)}.$$

Thus, by (35)

$$\Pr\{X \ge \mu + v\} \le \prod_{0 \le t \le v-1} \left(\frac{1}{1 + t/(\mu + 1)}\right).$$

Taking the logarithm of both sides and bounding the sum by an integral, we get

$$\text{(37)}\qquad \begin{aligned} \ln \Pr\{X \ge \mu + v\} &\le \sum_{0 \le t \le v-1} \ln\left(\frac{1}{1 + t/(\mu + 1)}\right) \\ &\le -\int_0^{v-1} \ln\left(1 + \frac{y}{\mu + 1}\right) dy \\ &= -(\mu + v) \ln\left(\frac{\mu + v}{\mu + 1}\right) + v - 1 \\ &\le -(\mu + v) \ln\left(\frac{\mu + v}{\mu}\right) + v - 1 - (\mu + v) \ln\left(1 - \frac{1}{\mu + 1}\right). \end{aligned}$$

Taking the exponential of both sides, we get

$$\text{(38)}\qquad \Pr\{X \ge \mu + v\} \le \left(\frac{\mu + v}{\mu}\right)^{-(\mu + v)} \exp\left(v - 1 - (\mu + v) \ln\left(1 - \frac{1}{\mu + 1}\right)\right).$$

Let us start with the summand of the second sum in (29). By applying (38) and the bound $\ln(1 - 1/(\mu + 1)) \geq -1/\mu$, we get

$$\Pr\{Y_{1,j',k} \geq \mu_k + \delta\mu_k\} \leq (1 + \delta)^{-(1+\delta)\mu_k} \exp(\delta\mu_k + \delta). \tag{39}$$

Since $\mu_k \geq M^3/N^2$, we get by some analysis that (39) is maximized when $\mu_k = M^3/N^2$, and we get

$$\Pr\{Y_{1,j',k} \geq \mu_k + \delta\mu_k\} \leq (1 + \delta)^{-(1+\delta)M^3/N^2} \exp\left(\delta \frac{M^3}{N^2} + \delta\right). \tag{40}$$

Note that in Phase 2 we always have $M^3/N^2 > \ln^2(M/B)$. The bound in (40) is exponentially small in $\delta^2 \log^2(M/B)$ when δ is small and exponentially small in $\delta(\log \delta) \log^2(M/B)$ when δ is large. For example, if $\delta < \frac{1}{2}$, we can use the bound $\ln(1 + x) \geq x - x^2/2$, for $x \geq 0$, to get

$$\begin{aligned}\Pr\{Y_{1,j',k} \geq \mu_k + \delta\mu_k\} &\leq \exp\left(\left(-(1 + \delta) \ln^2 \frac{M}{B}\right)(\delta - \delta^2/2)\right) \exp\left(\delta \ln^2 \frac{M}{B} + \delta\right) \\ &= \exp\left(\delta - \frac{\delta^2(1 - \delta)}{2} \ln^2 \frac{M}{B}\right) \\ &\leq \exp\left(\delta - \frac{\delta^2}{4} \ln^2 \frac{M}{B}\right),\end{aligned}$$

which is exponentially small when $\delta > 4/\ln^2(M/B)$.

Similarly, by (38) and the bound $\ln(1 - 1/(\mu + 1)) \geq -1/\mu$, the summand of the first sum in (29) for $\mu_k \geq \frac{1}{2}$ is

(41)
$$\Pr\left\{Y_{1,j',k} \geq \mu_k + \delta \frac{M^3}{N^2}\right\} \leq \left(1 + \delta \frac{M^3/N^2}{\mu_k}\right)^{-(\mu_k + \delta M^3/N^2)} \exp\left(\left(\frac{\delta M^3}{N^2}\right)\left(1 + \frac{1}{\mu_k}\right)\right),$$

which can be bounded by an expression similar to the right-hand side of (40). For smaller μ_k, we have from (38)

$$\begin{aligned}&\Pr\left\{Y_{1,j',k} \geq \mu_k + \delta \frac{M^3}{N^2}\right\} \\ &\quad\leq \left(1 + \delta \frac{M^3/N^2}{\mu_k}\right)^{-(\mu_k + \delta M^3/N^2)} \exp\left(\frac{\delta M^3}{N^2} - 1 - \left(\mu_k + \frac{\delta M^3}{N^2}\right) \ln \frac{\mu_k}{2}\right),\end{aligned} \tag{42}$$

which is exponentially decreasing in a way similar to (40).

In conclusion, by combining (29), (40), (41), and (42), we can bound the probability (28) by $4N^3/M^3$ times a term exponentially small in $l(\log l) \log^2(M/B)$. The $4N^3/M^3$ term is very quickly masked by the exponentially small term, since in Phase 2 we have $N < \sqrt{MBP}/\ln(M/B)$, which implies that $N^3/M^3 < (M/B)^{3/2}$. □

6.3. Completing the Sort. After Phase 2 is completed, we can read the blocks belonging to each bucket $\mathcal{S}_j$ using an optimal number $O(N/(PBS)$ of I/Os; the disk and track location of every block (including the dummy blocks) belonging to each partition can be easily computed because the placement of the blocks was deterministic. Bucket $\mathcal{S}_j$ contains at most M records, so it can be sorted internally. We sort the records of the bucket, form blocks, and write the blocks to the next available track/disk, cycling through the disks. We retain in internal memory the last block if partially full. The records in the final partially filled block from $\mathcal{S}_j$ can be treated as members of $\mathcal{S}_{j+1}$ when $\mathcal{S}_{j+1}$ is processed.

```
k := 1;
for each bucket $\mathcal{S}_j$ ($1 \le j \le S$) from Phase 2 do
  begin
  read $\mathcal{S}_j$ into internal memory;
  sort the records in internal memory by key values;
  form blocks of size $B$;
  for each full block do
    begin
    schedule the block to be written to the next available track on $\mathcal{D}_k$;
    $k := (k \bmod P) + 1$
    end;
  write the full blocks of $\mathcal{S}_j$
  end
```

6.4. Finding the Partitioning Elements. All that remains is to show how to compute with $O(N/PB)$ I/Os the $S - 1$ partitioning elements $b_1, b_2, \ldots, b_{S-1}$ that break up the file into S roughly equal-sized buckets. The jth bucket $\mathcal{S}_j$ consists of those records R such that

$$b_{j-1} \le key(R) < b_j,$$

where $b_0 = -\infty$ and $b_S = +\infty$. We need to show that conditions (3) and (4) of Section 6 are satisfied. Without loss of generality, we assume for simplicity of exposition that N, M, and $S - 1$ are powers of 2.

Our procedure for computing the approximate partitioning elements must work for the recursive step of the algorithm, so we assume that the N records are stored in $O(N/B)$ blocks of contiguous records, each of size at most B. Using the approach of [1], we first describe a subroutine that uses $O(n/PB)$ I/Os to find the record with the kth smallest key (or simply the kth smallest record) in a set containing

n records, in which the records are stored on disk in at most $O(n/B)$ blocks: We load the n records into memory, one memoryload at a time, and sort each of the $\lceil n/M \rceil$ memoryloads internally. We pick the median record from each of these sorted sets and find the median of the medians using the linear-time sequential algorithm developed in [2]. The number of I/Os required for these operations is $O(n/PB + n/M) = O(n/PB)$. We use the key value of this median record to partition the n records into two sets. It is easy to verify that each set can be partitioned into blocks of size B (except possibly for the last block) in which each group is stored contiguously on disk. It is also easy to see that each of the two sets has size bounded by $3n/4$. The algorithm is recursively applied to the appropriate half to find the kth smallest record; the total number of I/Os is $O(n/PB)$.

We now describe how to apply this subroutine to find the $S - 1$ approximate partitioning elements in a set containing N records. Let p and q denote powers of 2 to be specificed later. As above, we start out by sorting N/M memoryloads of records, which can be done with $O(N/PB)$ I/Os. Let us denote the ith sorted set by $\mathcal{M}_i$. We construct a new set $\mathcal{M}'$ of size at most N/p consisting of the kpth record (in sorted order) of $\mathcal{M}_i$, for $1 \le k \le M/p$ and $1 \le i \le N/M$. The records in $\mathcal{M}'_i$ can be output one block at a time. The total number of contiguous blocks of records comprising $\mathcal{M}'$ is $O(|\mathcal{M}'|/B)$, so we can apply the subroutine above to find the record of rank jq in $\mathcal{M}'$ with only $O(|\mathcal{M}'|/PB) = O(N/pPB)$ I/Os; we call its key value b_j. Thus, if $p = \Omega(S)$, the $S - 1$ b_j's can be found with a total of $O(SN/pPB) = O(N/PB)$ I/Os.

The above description can be expressed in the following pseudocode:

```
for each memoryload of records $\mathcal{M}_i$ $(1 \le i \le N/M)$ do
  begin
  read $\mathcal{M}_i$ into internal memory;
  sort the records in internal memory by key values;
  construct $\mathcal{M}'_i$ so that it consists of every $p$th record in memory;
  write $\mathcal{M}'_i$
  end;
$\mathcal{M}' := \mathcal{M}'_1 + \cdots + \mathcal{M}'_{N/M}$;
for $j := 1$ to $S - 1$ do
  $b_j :=$ record of rank $qj$ in $\mathcal{M}'$
```

The two lemmas below show that the partitioning is done evenly in Phases 1 and 2, respectively.

LEMMA 3. *In the above partitioning algorithm, the number of partitioning elements* S *satisfies* $S = N/pq + 1$. *If we choose* $p = \max\{2, (S-1)/4\}$ *and* $q = N/(S-1)p$ *in Phase* 1, *where* $(S-1)^2 \le 2M$, *then condition* (3) *of Section* 6 *is true*; *that is*

$$\frac{N}{2(S-1)} < N_j < \frac{3N}{2(S-1)}.$$

LEMMA 4. *If $p = M^2/8N$ and $q = 4N/M$ in the above partitioning algorithm for Phase 2, then $S = 2N/M + 1$ and condition (4) of Section 6 is true; that is,*

$$\tfrac{3}{8}M < N_j < \tfrac{5}{8}M.$$

The choices of p in Lemmas 3 and 4 satisfy $p = \Omega(S)$, as can be verified by using the condition $N < \sqrt{MBP}/\ln(M/B)$ for Phase 2, and thus the partitioning can be done with $O(N/PB)$ I/Os, as mentioned above.

Lemmas 3 and 4 are special cases of the following general partitioning lemma:

LEMMA 5. *The size N_j of the* jth *bucket $\mathscr{S}_j$ produced by the above partitioning algorithm satisfies*

$$pq - p\frac{N}{M} < N_j < pq + p\frac{N}{M}.$$

PROOF. Each element in $\mathscr{M}'$ corresponds to a collection of p elements in the original file. Since the chosen partitioning elements are q apart in $\mathscr{M}'$, this gives us pq elements that could be in $\mathscr{S}_j$. Let e_i and e_{i+1} represent the pith record and $p(i + 1)$st record from the file in some memoryload. If $e_i < b_j < e_{i+1}$, then the $p - 1$ elements between e_i and e_{i+1} may also be in $\mathscr{S}_j$. Thus there may be $p - 1$ additional records from each of the memoryloads, except from the memoryload that contributed b_j. This gives us the upper bound

$$N_j \le pq + \left(\frac{N}{M} - 1\right)(p - 1) < pq + p\frac{N}{M}.$$

By similar reasoning, we get the lower bound

$$N_j \ge pq - \frac{N}{M}(p - 1) > pq - p\frac{N}{M}. \qquad \square$$

Lemmas 3 and 4 follow directly from Lemma 5. The condition $(S - 1)^2 \le 2M$ in Lemma 3 is satisfied by the setting $S \le \sqrt{M/B}/\ln^2(M/B) + 1$ for Phase 1, and we have $pN/M = (S - 1)N/4M \le N/2(S - 1)$.

6.5. Permuting for Very Small P and B. Aggarwal and Vitter [1] show in their one-disk model with P block transfers per I/O that the optimal way to permute when P and B are very small is the naive method of repeatedly moving P records in each I/O from their inputed positions to the desired final positions. This makes no use of blocking; each block transfer is used to transfer a single record. The resulting number of I/Os is $O(N/P)$.

This algorithm does not translate directly to our more realistic two-level model with parallel block transfer, because there is no way to guarantee that the P block transfers involve separate disks. Instead, we achieve the desired $O(N/P)$ I/O bound by using the following technique inspired by Phase 1: In the first pass, the records are read, one memoryload at a time. Each memoryload is permuted randomly in internal memory and written back to disk in the new permuted order. As a result, the records that need ultimately to end up on a particular disk i (call them $\mathscr{R}_i$) are spread with high probability uniformly among all the disks. This is true for each $1 \le i \le P$. (It might take two passes to do this, using $\sqrt{P}$ bins each time instead of P so that there is enough internal data structure space to manage the placement of all the bins on the disks.)

In the deterministic second pass, for each $1 \le i \le P$ in parallel, one block of $\mathscr{R}_i$ is read into internal memory. Then B writes are executed; during each write, one record from $\mathscr{R}_i$ is written to disk i, in parallel for each $1 \le i \le P$.

The total number of I/Os is $O(N/P)$, as desired, asuming that the first pass spreads each $\mathscr{R}_i$ uniformly among the P disks. This uniformity condition can be proven using a modification of the analysis of Section 6.1.

7. Standard Matrix Multiplication.

The following is a basic divide-and-conquer approach for scheduling the multiplication of two $k \times k$ matrices using the standard algorithm:

1. If $k \le \sqrt{M}$, we multiply the matrices internally. Otherwise we do the following steps:
2. We subdivide A and B into eight $k/2 \times k/2$ submatrices: A_1–A_4 and B_1–B_4.

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, \qquad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}.$$

   We reposition the records of the eight submatrices so that each submatrix is stored in row-major order.
3. We use the algorithm recursively to compute

$$\begin{aligned} C_1 &= A_1B_1 + A_2B_3, \\ C_2 &= A_1B_2 + A_2B_4, \\ C_3 &= A_3B_1 + A_4B_3, \\ C_4 &= A_3B_2 + A_4B_4. \end{aligned}$$

4. We reposition C_1–C_4 so that C is stored in row-major order.

We partition secondary storage into four contiguous parts, one part for each set of submatrices. We define $T(k)$ to be the number of I/Os used to add two $k \times k$

matrices. Step 2 takes $O(k^2/PB)$ I/Os since, in the worst case, we can have at most four blocks that are assigned to be written to the same disk. The number of I/Os needed to do eight multiplications of submatrices recursively in Step 3 is $8T(k/2)$, and the additions take a linear number $O(k^2/PB)$ of I/Os, since all of the submatrices are packed in blocks. Step 4 takes $O(k^2/PB)$ I/Os; it is similar to Step 2. When $k > \sqrt{M}$, we get the following recurrence:

$$T(k) = 8T\left(\frac{k}{2}\right) + O\left(\frac{k^2}{PB}\right),$$

where $T(\sqrt{M}) = M/PB$. This gives us the desired I/O bound from Theorem 5.

The recurrence for $I(k)$, the amount of work done internally, satisfies

$$I(k) = 8I\left(\frac{k}{2}\right) + O(k^2 \log P'),$$

where $I(\sqrt{M}) = M/PB$, which yields the desired upper bound $I(k) = O(k^3)$. The internal processing time is $O(I(k)/P)$.

8. Conclusions.

In this paper we have introduced a new and realistic model of two-level storage with parallel block transfer between the internal memory and secondary storage. We have developed practical algorithms for sorting, permuting, matrix transposition, FFT, permutation networks, and standard matrix multiplication, that use an optimal number of I/O steps. The algorithms for sorting and permuting are based upon a randomized version of distribution sort. The partitioning is done by a combination of two interesting probabilistic techniques in order to guarantee that the accesses are spread uniformly over the disks. Applications of these techniques to obtain optimal algorithms for the P-HMM and P-BT hierarchical memory models are developed in the companion paper [19].

Preliminary work suggests that the amount of randomness in our distribution sort algorithm can be greatly reduced by applying universal hashing [3] in an interesting way. However, the problem of removing randomness completely from this technique is more difficult.

The study of I/O efficiency has many applications besides the ones we studied in this paper. For example, graphics applications, multidimensional search problems, and iterated lattice computations often involve I/O-bound tasks. We expect that the algorithms and insights we develop in this paper will have many applications in those domains.

Addendum. At the beginning of Section 6 we gave some "intuitions" as to why merge sort seemed especially hard to implement with an optimal number of I/Os in our two-level disk model. Oddly enough, a practical and optimal deterministic sorting algorithm was recently developed by Nodine and Vitter [12] using a "greedy" merge sort. Unfortunately this merge sort algorithm does not seem to lead to optimal deterministic sorting algorithms in most cases of the P-HMM, P-BT, and other parallel hierarchical memory models. Nodine and Vitter have subsequently developed an optimal distribution sort algorithm that is deterministic and that does generalize to give optimal deterministic parallel hierarchy algorithms [13].

Acknowledgments. We thank Bowen Alpern, Larry Carter, Tom Cormen, Mike Goodrich, and Mark Nodine for several helpful comments.

References

[1] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM* **31**(9) (September 1988), 1116–1127.

[2] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. E. Tarjan, Time Bounds for Selection, *Journal of Computer and System Sciences* **7**(4) (1973), 448–461.

[3] J. L. Carter and M. N. Wegman, Universal Classes of Hash Functions, *Journal of Computer and System Sciences* **18** (April 1979), 143–154.

[4] R. W. Floyd, Permuting Information in Idealized Two-Level Storage, in *Complexity of Computer Calculations*, R. Miller and J. Thatcher, eds., Plenum, New York, 1972, pp. 105–109.

[5] L. Hellerstein, G. A. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson, Coding Techniques for Handling Failures in Large Disk Arrays, *Algorithmica*, this issue, pp. 182–208.

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

[7] L. Kleinrock, *Queueing Systems*, Vol. I, Wiley, New York, 1979.

[8] D. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, MA, 1973.

[9] F. T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers* **34** (April 1985), 344–354.

[10] E. E. Lindstrom and J. S. Vitter, The Design and Analysis of BucketSort for Bubble Memory Secondary Storage, *IEEE Transactions on Computers* **34** (March 1985), 218–233.

[11] N. B. Maginnis, Store More, Spend Less: Mid-Range Options Around, *Computerworld*, November 16, 1986, p. 71.

[12] M. H. Nodine and J. S. Vitter, Large-Scale Sorting in Parallel Memories, *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1991, pp. 29–39.

[13] M. H. Nodine and J. S. Vitter, Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors, *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1993, pp. 120–129.

[14] D. A. Patterson, G. Gibson, and R. H. Katz, A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, June 1988, pp. 109–116.

[15] J. Savage and J. S. Vitter, Parallelism in Space–Time Tradeoffs, in *Advances in Computing Research*, Vol. 4, F. P. Preparata, ed., JAI Press, Greenwich, CT, 1987, pp. 117–146.

[16] H. S. Stone, Parallel Processing with the Perfect Shuffle, *IEEE Transactions on Computers* **20** (February 1971), 153–161.

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

[18] J. S. Vitter and Ph. Flajolet, Average-Case Analysis of Algorithms and Data Structures, in *Handbook of Theoretical Computer Science*, Jan van Leeuwen, ed., North-Holland, Amsterdam, 1990, pp. 431–524.

[19] J. S. Vitter and E. A. M. Shriver, Algorithms for Parallel Memory, II: Hierarchical Multilevel Memories, *Algorithmica*, this issue, pp. 148–169.

[20] C. Wu and T. Feng, The Universality of the Shuffle-Exchange Network, *IEEE Transactions on Computers* **30** (May 1981), 324–332.