

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M7

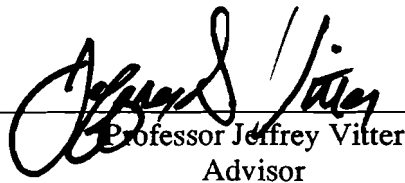
Optimal Disk I/O with Parallel Block Transfer

by
Elizabeth A. Shriver

Optimall Disk I/O with Parallel Block Transfer
Master's Thesis

Elizabeth A. Shriver
Department of Computer Science
Brown University
May 14, 1990

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science at Brown University



Professor Jeffrey Vitter
Advisor

Optimal Disk I/O with Parallel Block Transfer

Elizabeth A. M. Shriver

Department of Computer Science
Brown University
Providence, R. I. 02912-1910

May 14, 1990

Preface

I wish to thank a number of people without whose help and support this work would never have been finished: Jin Joo Lee, for all her help with the math analysis; Tom Swartz, for his patience; and most of all, my advisor Jeff Vitter.

This research was done in partial fulfillment of the requirements for my Master's degree at Brown University. It is joint work with Jeff Vitter.

Abstract

We provide the first optimal algorithms in terms of the number of input/outputs (I/Os) required between internal memory and multiple disk drives for 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 disks can simultaneously transfer a contiguous block of B records. We also introduce parallel variants of the hierarchical memory models of [AAC, ACS] and give optimal algorithms. In our parallel models, there are P hierarchies operating in parallel; communication among the hierarchies takes place at a base memory level.

The difficulty in developing optimal algorithms in our two-level and hierarchical models is to cope with the partitioning of memory into P separate physical devices. The popular technique of “disk striping” handles this easily, but it can be nonoptimal by a logarithmic factor in terms of the number of I/Os. Our optimal sorting algorithms are randomized, but practical; the probability of using more than an optimal number of I/Os falls off exponentially.

Contents

| | |
|--|-----------|
| Preface | i |
| Abstract | ii |
| 1 Introduction | 1 |
| 2 Problem Definitions | 2 |
| 3 The Two-Level Memory Model | 4 |
| 4 Hierarchical Memory Models | 6 |
| 5 Shuffle-Merge and its Applications | 8 |
| 6 Disk Sorting and Permuting | 10 |
| 6.1 Phase 1 | 12 |
| 6.2 Phase 2 | 18 |
| 6.3 Finding the Partitioning Elements | 26 |
| 7 Disk Standard Matrix Multiplication | 27 |
| 8 Algorithms for the Hierarchical Memory Models | 28 |
| 8.1 Sorting in PHMM | 29 |
| 8.2 Standard Matrix Multiplication in PHMM | 32 |
| 8.3 Sorting in PBT | 34 |
| 8.4 Standard Matrix Multiplication in PBT | 37 |
| 9 Conclusions | 38 |
| References | 38 |

1 Introduction

Sorting is the canonical information-processing application. It accounts for roughly 20–25 percent of the computing resources on large-scale computers [Knu, LiV]. 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. This bottleneck is accentuated as processors get faster and parallel computers are used. One remedy is to use secondary storage systems with parallel capabilities.

Previous work on I/O efficiency has concentrated on two-level and multilevel models. (Further references can be found in the papers mentioned below.) Aggarwal and Vitter [AgV] present optimal upper and lower bounds for sorting-related problems using a two-level memory model in which P physical blocks, each consisting of B contiguous records, can be transferred simultaneously in a single I/O. Their model is somewhat unrealistic, however, because secondary storage is usually partitioned into separate physical devices, each capable of transferring only one block per I/O.

Two multilevel hierarchical memory models are introduced in [AAC, ACS], the latter one taking into account block transfer. Access to a location x takes time $f(x)$, and in the blocking version, access to successive locations takes one unit of time per location. Optimal bounds are obtained for several problems. Parallel transfer using several hierarchies simultaneously was not considered in [AAC, ACS].

We are interested in optimal algorithms for two-level and hierarchical memory models that allow P simultaneous transfers of data. To be realistic, we require that each block transfer must be associated with a separate secondary storage device.

The problems we solve, which include sorting, permuting, matrix transposition, FFT, permutation networks, and standard matrix multiplication, are defined in Section 2. In Section 3 we define a realistic two-level memory model with parallel block transfer, and we state our main results, which give tight upper and lower bounds on the number of I/Os needed to solve the above important problems. The two-level model corresponds to having an internal memory and P disks, each disk capable of simultaneously transferring one block of B records. Our measure of performance is the number of parallel I/Os required; this ignores internal computation time, but the internal processing done by our algorithms is simple enough so that in practice it can be overlapped with the I/O time. Our algorithms can be significantly faster than those obtained by the well-known technique of disk striping.

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 [AgV]. This distinction is akin to the difference in parallel computation between the more realistic MPP (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 [AgV], the resulting algorithms would not be optimal in terms of I/O. The algorithms we develop on our more realistic model use the same number of I/Os as those in [AgV] for the less realistic model.

In Section 4 we define two uniform memory models, each consisting of P hierarchical memories connected together at their base levels. The P hierarchical memories are of the

type discussed in [AAC, ACS]. We give optimal time bounds for the problems in each model.

Sections 5–7 are devoted to the algorithms and analysis for the two-level model. 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 nonadaptive.

Our main result 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 $\ell(\log \ell) \log(M/B)$, where M is the internal memory size.¹ 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. In Section 7 we cover standard matrix multiplication.

In Section 8 we show how to apply the algorithms developed for the two-level model to get optimal algorithms for the hierarchical models. The hierarchical algorithms are optimal because the internal processing in the corresponding two-level algorithms is efficient. Conclusions and open problems are given in Section 9.

2 Problem Definitions

Most of the following problems have been well described in the literature. The following definitions are those from [AgV], 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 non-decreasing 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 $\{1, 2, \dots, N\}$.

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

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 in the first N locations of secondary storage.

Goal: The internal memory is empty, and the transposed matrix A^T is stored in the first N locations of secondary storage. (The $q \times p$ matrix A^T is called the transpose of A if $A_{i,j}^T = A_{j,i}$, for all $1 \leq i \leq q$ and $1 \leq j \leq p$.)

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

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 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 $\log N + 1$ columns each containing N nodes; column 0 contains the N input nodes, and column $\log N$ contains the N output nodes. Each non-input node has indegree 2, and each non-output node has outdegree 2. We shall denote the i th node ($0 \leq i \leq N - 1$) in column j ($0 \leq j \leq \log N$) in the FFT digraph by $n_{i,j}$. For $j \geq 1$ 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 i th node in each column 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 [Knu] 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 $J + 1$ columns, for some $J \geq \log N$, each containing N nodes. Column 0 contains the N input nodes, and column J contains the N output nodes. All edges are directed between adjacent columns, in the direction of increasing index. For $0 \leq i \leq N - 1$ and $0 \leq j \leq J$, we denote the i th node in column j as $n_{i,j}$. For each $j \geq 1$ 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 column 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 p_1, p_2, \dots, p_N 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 i th node in each column 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 and memory accesses 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.

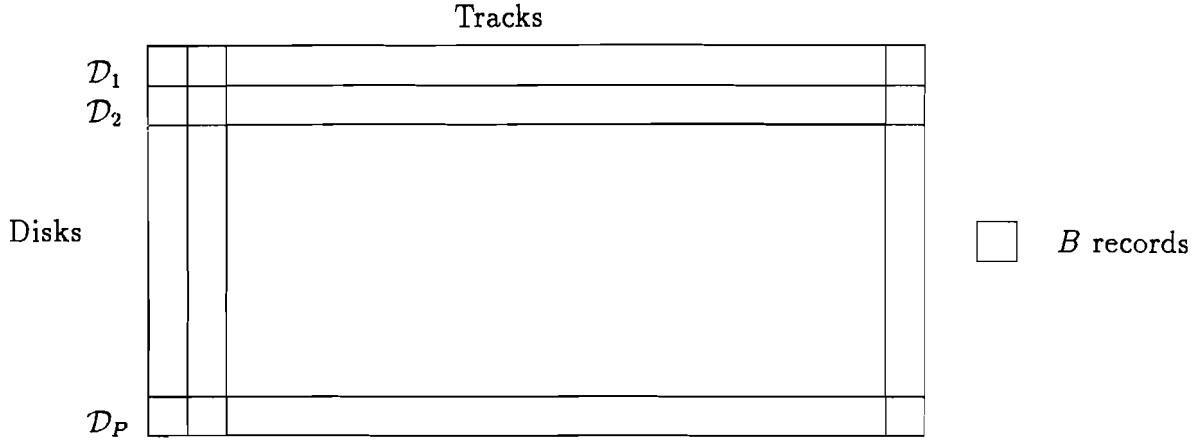


Figure 1: The disks are represented by horizontal lines and the tracks by vertical lines.

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.

3 The Two-Level Memory Model

First we define the parameters for our *two-level memory model* (or *disk model*) with *parallel block transfer*:

Definition 1 The parameters are defined by

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

where $1 \leq B \leq M/2$, $M < N$, and $1 \leq P \leq \lfloor M/B \rfloor$. The parameters N , M , B , and P are referred to as the *file size*, *memory size*, *block size*, and *number of disks* respectively. We denote the P disks by $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_P$. Each disk is partitioned into consecutive *tracks*, each capable of storing one block of B records, as pictured in Figure 1. If no disk is specified when we refer to the " k th track," we mean the k th track of all P disks collectively.

For purposes of making the problem definitions given in Section 2 more concrete, the locations on disk are numbered track-by-track in the following cyclical fashion. Track 1 contains the first PB locations:

track 1 of \mathcal{D}_1 contains locations $1, 2, \dots, B$;
 track 1 of \mathcal{D}_2 contains locations $B + 1, B + 2, \dots, 2B$;
 \dots
 track 1 of \mathcal{D}_P contains locations $(P - 1)B + 1, (P - 1)B + 2, \dots, PB$.

Track 2 contains the next PB locations:

track 2 of \mathcal{D}_1 contains locations $PB + 1, PB + 2, \dots, (P + 1)B$;
 track 2 of \mathcal{D}_2 contains locations $(P + 1)B + 1, (P + 1)B + 2, \dots, (P + 2)B$;
 \dots
 track 2 of \mathcal{D}_P contains locations $(2P - 1)B + 1, (2P - 1)B + 2, \dots, 2PB$.

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

Parallelism appears in our model in two basic ways. First, records are transferred concurrently in blocks of B contiguous records. This reflects the fact that seek time is a dominant factor in I/O. It takes roughly the same amount of time to access and transfer one block as it does one 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.

Our results for the two-level memory model are given below in Theorems 1–4. We give optimal algorithms in terms of the number of I/Os for the problems defined in the previous section. The lower bounds for Theorems 1–3 follow from the lower bounds proved in [AgV] 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 [AgV], the same lower bounds apply.

Theorem 1 *The number of I/Os required for sorting N records and for computing the N -input FFT digraph is*

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

The upper bound for sorting is given by a randomized algorithm; the probability of using more than an optimal number of I/Os falls off exponentially. The lower bounds apply to both the average case and the worst case. The sorting 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 average-case and worst-case number of I/Os required for computing any N -input permutation network is

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

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

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

Theorem 2 *The number of I/Os required to permute N records is*

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

The lower bound applies to both the average case and the worst case. The second term in the upper bound corresponds to the randomized algorithm of Theorem 1.

Theorem 3 *The number of I/Os required to transpose a $p \times q$ matrix of $N = pq$ elements is*

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

We get the following lower bound for standard matrix multiplication by taking the bound for the case $P = 1$ in [SaV] and dividing by P :

Theorem 4 *The number of I/Os required to multiply two $k \times k$ matrices using the standard matrix multiplication algorithm is*

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

4 Hierarchical Memory Models

A *hierarchical memory model* is a uniform model consisting of memory whose locations take different amounts of time to access. The basic unit of transfer in the hierarchical memory model HMM [AAC] is the record; access to location x takes time $f(x)$. The BT model [ACS] represents a notion of block transfer applied to HMM; in the BT model, access to the $\ell + 1$ records at locations $x - \ell, x - \ell + 1, \dots, x$ takes time $f(x) + \ell$. Typical access cost functions are $f(x) = \log x$ and $f(x) = x^\alpha$, for some $\alpha > 0$.

Both of these hierarchical memory models can be augmented to allow parallel data transfer. One possibility is to have a discretized hierarchy, in which each memory component is connected to P larger but slower memory components at the next level. A cleaner extension is to consider P separate memories connected together at the base level of each hierarchy. We shall adopt this extension, since it is simpler to realize than the previous one, and it allows the same time performance for our algorithms.

More specifically, we assume that the P hierarchies can each function independently. Communication between hierarchies takes place at the *base memory level*, which consists of location 1 from each of the P hierarchies. We assume that the P base memory level locations are interconnected via a network such as a hypercube or cube-connected cycles so that the P records in the base memory level can be sorted in $O(\log P)$ time (perhaps via a randomized algorithm [ReV]) and so that two $\sqrt{P}/2 \times \sqrt{P}/2$ matrices can be multiplied in $O(\sqrt{P})$ time using the standard algorithm. We denote by PHMM and PBT the P -hierarchy variants of the hierarchical memory models HMM and BT, as described above.

The fundamental problem that arises in trying to take full advantage of parallel transfer in these models is how to distribute records among the P memories so that each memory is kept “busy.” We shall show later in Section 8 how the randomized distribution sort algorithm

of Section 6 for the two-level memory model can be used as a basic building block to get optimal algorithms for the hierarchical models. The lower bounds for PHMM and PBT follow from the approach used in [AAC] and [ACS].

Theorem 5 *In the PHMM model, the time for sorting and the FFT is*

$$\begin{aligned} \Theta \left(\frac{N}{P} \log N \log \left(\frac{\log N}{\log P} \right) \right) & \quad \text{if } f(x) = \log x; \\ \Theta \left(\left(\frac{N}{P} \right)^{\alpha+1} + \frac{N}{P} \log N \right) & \quad \text{if } f(x) = x^\alpha, \alpha > 0. \end{aligned}$$

The upper bound for sorting is given by a randomized algorithm; the probability of using more than an optimal number of I/Os falls off exponentially. In the lower bound for the $f(x) = x^\alpha$ case, the $(N/P) \log N$ term requires the comparison model of computation. The time for multiplying two $k \times k$ matrices together using the standard algorithm is

$$\begin{aligned} \Theta \left(\frac{k^3}{P} \right) & \quad \text{if } f(x) = \log x; \\ \Theta \left(\frac{k^3}{P} \right) & \quad \text{if } f(x) = x^\alpha, 0 < \alpha < \frac{1}{2}; \\ \Theta \left(\frac{k^3}{P^{3/2}} \log k + \frac{k^3}{P} \right) & \quad \text{if } f(x) = x^\alpha, \alpha = \frac{1}{2}; \\ \Theta \left(\left(\frac{k^2}{P} \right)^{\alpha+1} + \frac{k^3}{P} \right) & \quad \text{if } f(x) = x^\alpha, \alpha > \frac{1}{2}. \end{aligned}$$

Theorem 6 *In the PBT model, the time for sorting and the FFT is*

$$\begin{aligned} \Theta \left(\frac{N}{P} \log N \right) & \quad \text{if } f(x) = \log x; \\ \Theta \left(\frac{N}{P} \log N \right) & \quad \text{if } f(x) = x^\alpha, 0 < \alpha < 1; \\ \Theta \left(\frac{N}{P} \log N \log \frac{N}{P} \right) & \quad \text{if } f(x) = x^\alpha, \alpha = 1; \\ \Theta \left(\left(\frac{N}{P} \right)^\alpha + \frac{N}{P} \log \frac{N}{P} \log P \right) & \quad \text{if } f(x) = x^\alpha, \alpha > 1. \end{aligned}$$

The upper bounds for sorting are given by a randomized algorithm; the probability of using more than an optimal number of I/Os falls off exponentially. The $(N/P) \log N$ terms in the lower bounds require the comparison model of computation. The time for multiplying two

$k \times k$ matrices together using the standard algorithm is

$$\begin{aligned} \Theta \left(\frac{k^3}{P} \right) & \quad \text{if } f(x) = \log x; \\ \Theta \left(\frac{k^3}{P} \right) & \quad \text{if } f(x) = x^\alpha, \quad 0 < \alpha < \frac{3}{2}; \\ \Theta \left(\frac{k^3}{P^{3/2}} \log k + \frac{k^3}{P} \right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha = \frac{3}{2}; \\ \Theta \left(\left(\frac{k^2}{P} \right)^\alpha \right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha > \frac{3}{2}. \end{aligned}$$

These techniques can be extended to get optimal algorithms for other problems considered in [AAC, ACS], such as the “touch” problem and other “simple” problems, searching, and generating rational permutations.

5 Shuffle-Merge and its Applications

In this section we restrict ourselves to the two-level memory model, in which there are $P \leq M/B$ disk drives communicating with internal memory. We define a useful merging operation called *shuffle-merge* that can be used to achieve the optimal I/O bounds mentioned in Theorems 1 and 3 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 [AgV], 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.

For simplicity of exposition, we assume that N , M , P , and B are powers of 2. The operation of *shuffle-merge* consists of performing a perfect shuffle [Sto] of the elements of M/B ordered equally-sized runs of records, a_i^1, \dots, a_i^r , for $1 \leq i \leq M/B$, to get the final shuffled run $a_1^1, a_2^1, \dots, a_{M/B}^1, a_1^2, a_2^2, \dots, a_{M/B}^2, \dots, a_1^r, a_2^r, \dots, a_{M/B}^r$.

| Output Runs | | Input Runs |
|---|--------------|--|
| $a_1^1 \dots a_{M/B}^1 \quad a_1^2 \dots a_{M/B}^2 \dots a_1^r \dots a_{M/B}^r$ | \Leftarrow | $a_1^1 \quad a_1^2 \quad \dots \quad a_1^r$ \vdots $a_{M/B}^1 \quad a_{M/B}^2 \quad \dots \quad a_{M/B}^r$ |

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 k th block of records $a_i^{(k-1)B+1}, \dots, a_i^{kB}$ from the i th run is stored on track $(i-1)[r/PB] + [k/P]$ of disk $\mathcal{D}_{1+((k+i-2) \bmod P)}$. (If $r < \frac{1}{2}PB$, 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 $(k-1)M/PB + 1, \dots, kM/PB$, we bring into internal memory the k th 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 [WuF]. 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 digraph is defined in Section 2. 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 $\log M$ columns 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 $\log M$ columns, 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 $\log M$ columns. 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

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

I/Os, which can be shown by some algebraic manipulation to equal the bound given in Theorem 1.

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

$$x = \begin{cases} 1 & \text{if } B < \min\{p, q\}; \\ \frac{B}{\min\{p, q\}} & \text{if } \min\{p, q\} \leq B \leq \max\{p, q\}; \\ \frac{B^2}{N} & \text{if } \max\{p, q\} < B. \end{cases} \quad (1)$$

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

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

each requiring $O(N/PB)$ I/Os. The upper bound in Theorem 3 follows by substituting the three cases of (1) into (2).

6 Disk Sorting and Permuting

In this section we present and analyze the optimal algorithm for sorting and permuting on the two-level memory model. For simplicity of exposition, we assume that N , M , P , and B are powers of 2.

Permuting records is a special case of sorting. The bounds for sorting and permuting given in Theorems 1 and 2 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 trivially, record by record, in $O(N/P)$ I/Os. So we shall restrict our attention to the sorting problem.

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. And when the merges are not perfect shuffles, it is very 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 open problem is whether merge sort can be implemented using the optimal number of I/Os. (See the Addendum.)

A well-known alternative approach is *disk striping*, in which 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 effectively reduces our model to having only one disk, but with larger block size $B' = PB$. However, if we plug these values into Theorem 1, we find that the number of I/Os used is not optimal when PB is large. The number of I/Os can be nonoptimal by a relative factor of $\log(M/B)$, which can be significant in large-scale applications. For small values of PB , striping will work fine.

In this paper we are interested in solving the general problem of sorting for high performance systems in which PB is large. 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, \dots, 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 j th bucket consists of all the records R in the file whose key values are in the range

$$b_{j-1} \leq \text{key}(R) < b_j.$$

3. [Partition into buckets.] We partition the file into buckets based on 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 is used for the partitioning when $N \geq \sqrt{MBP}/\ln(M/B)$. It uses 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 in hashing [ViF], this happens intuitively 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 nonoptimal performance. In the latter case, when $N < \sqrt{MBP}/\ln(M/B)$, Phase 2 is used for one partitioning phase, after which each bucket will have at most M records and can be sorted internally. Phase 2 uses a partitioning technique motivated by routing on the FFT and works with overwhelming probability.

Definition 2 We denote the S buckets by S_1, S_2, \dots, S_S . The number of records in the file that belong to bucket S_j is denoted N_j . In our memory model, we can look at only M records at a time, so partitioning will be done one memoryload at a time. We denote the i th memoryload by \mathcal{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 $\mathcal{W}_1, \mathcal{W}_2, \dots$. Read cycles are defined analogously.

For the time being, let us assume that we can deterministically compute the approximate partitioning elements b_1, b_2, \dots, b_{S-1} , using $O(N/PB)$ I/Os. (The algorithm and the analysis for computing the partitioning elements are given in Section 6.3.) For Phase 1 we set $S = \sqrt{M/B}/\ln^2(M/B)$; we show later in Lemma 3 that

$$\frac{N}{2S} \leq N_j \leq \frac{3N}{2S}. \quad (3)$$

For Phase 2 we set $S = 8N/7M$; we show later in Lemma 4 that

$$\frac{3M}{4} \leq N_j \leq M. \quad (4)$$

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

Theorem 7 *The number of I/Os used by the above distribution sort algorithm to sort N records is*

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

with overwhelming probability. In particular, the probability that the number of I/Os used is more than $1 + \ell$ times the average is exponentially small in $\ell(\log \ell) \log(M/B)$.

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 shall see from Theorem 8 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. The above construction gives us

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

In Phase 1 we set $S = \sqrt{M/B}/\ln^2(M/B)$; hence, $N_j = \Theta(N/S) = \Theta(N \ln^2(M/B)/\sqrt{M/B})$. Substituting this into (5) and using the fact that Phase 1 certainly ends if $N \leq M$, we get

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

In Theorem 9, 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

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

The probability bounds follow from Theorems 8 and 9. □

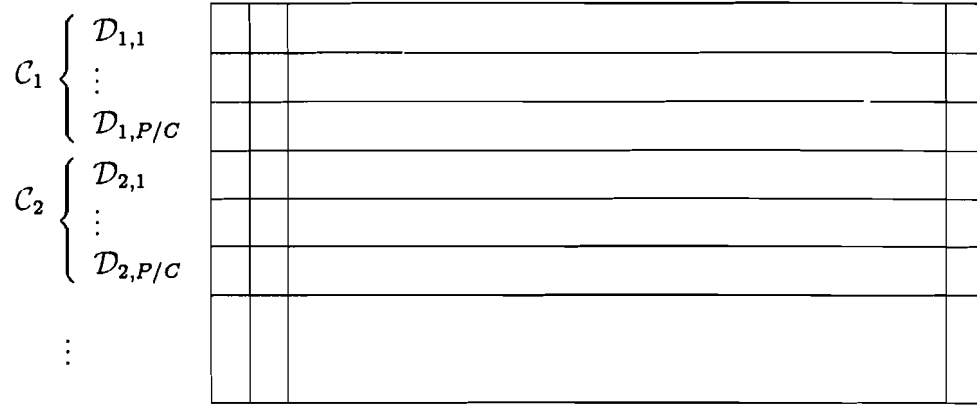
6.1 Phase 1

We use Phase 1 to partition the records of a file of N records when $N \geq \sqrt{MBP}/\ln(M/B)$. The number of partitions is set to $S = \sqrt{M/B}/\ln^2(M/B)$. 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 crux of the problem. We shall show in Theorem 8 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 on the partitioning elements and organize the records so that records in the each bucket are contiguous 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 so that the records of a bucket (among all the memoryloads) will be spread out evenly among the disks.

For each bucket, 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 pictured in Figure 2. We set $C = \min\{P, S\}$.

Definition 3 A *cluster* is a logical grouping of consecutive disks. The C clusters are denoted C_1, C_2, \dots, C_C . The P/C disks in the k th cluster C_k are denoted $D_{k,1}, D_{k,2}, \dots, D_{k,P/C}$. The i th *track of a cluster* refers collectively to the i th 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*.

Figure 2: Decomposition of the P disks into C clusters.**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 C_k by bucket S_j . Let $next_track_k$ represent the first track on 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 i th memoryload is brought into internal memory. The records are partitioned into buckets, based on 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 C groups to be written during this write cycle, and we assign these groups to clusters by choosing a random permutation of $\{1, 2, \dots, C\}$. 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 S_j is assigned to cluster C_k . We assign the first $P/C - last_disk_{j,k}$ blocks to disks $last_disk_{j,k} + 1, \dots, P/C$ on track $last_track_k$; we assign the remaining blocks, if any, to disks $1, 2, \dots$ on track $next_track_k$. We then update the value of $last_disk_{j,k}$, and when the current track gets filled, we set $last_track_{j,k}$ to $next_track_k$ and increment $next_track_k$.

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 S_j is written to cluster C_k , we fill up the last track written on that cluster for 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 S_j is written to the first disk of a cluster C_k , the *pg* pointer of that block is set to $last_track_{j,k}$. 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 $last_disk_{j,k}$ and $last_track_{j,k}$, so that we can locate the records for each bucket during the next level of recursion.

```

 $last\_disk_{j,k} := 0$  for all  $j, k$ ;
 $next\_track_k := 1$  for all  $k$ ;
 $last\_track_{j,k} := 0$  for all  $j, k$ ;
for each memoryload of records  $M_i$  ( $1 \leq i \leq N/M$ ) do
  begin
    read  $M_i$  into internal memory;
    partition the records into buckets based on the partitioning elements;
    for each bucket  $S_j$  ( $1 \leq j \leq S$ ) 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  $W_t$  do
      begin
        choose  $C$  groups of blocks to be written in  $W_t$ ;
        assign the groups to clusters via a random permutation of  $\{1, 2, \dots, C\}$ ;
        { assign the blocks in each group to the disks in a cluster }
        for each cluster  $C_k$  ( $1 \leq k \leq C$ ) do
          begin
            let  $S_j$  be the bucket whose group is assigned to  $C_k$ ;
            for each disk  $\mathcal{D}_{k,d}$  such that  $P/C - last\_disk_{j,k} \leq d \leq P/C$  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  $1 \leq d \leq P/C - last\_disk_{j,k}$  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;
              end;
          end;
        end;
      end;
    end;
  end;

```

```

    update last_diskj,k
  end;
  write the blocks in  $\mathcal{W}_t$  to the desired disks
end
end;
write pointers last_diskj,k and last_trackj,k, for all j, k

```

Analysis of Phase 1

Theorem 8 *With overwhelming probability, each pass of Phase 1 uses $O(N/PB)$ I/Os to partition a file of N records. In particular, the probability that the number of I/Os used is more than $1+\ell$ times the average is exponentially small in $\ell(\log \ell) \cdot \max\{\log(M/B), N/PBS\}$.*

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 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

$$Z \approx \frac{N}{PB}.$$

We do that by showing that

$$\Pr \left\{ Z \geq (1+\ell) \frac{N}{PB} \right\}$$

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

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 C_k that have been assigned to bucket S_j . We have

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

This gives us

$$\Pr \left\{ Z \geq (1+\ell) \frac{N}{PB} \right\} = \Pr \left\{ \sum_{1 \leq j \leq S} \max_{1 \leq k \leq C} \{X_{j,k}\} \geq \frac{(1+\ell)N}{PB} \right\} \quad (6)$$

The max term in (6) is the difficult expression to analyze. We shall 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

$$\begin{aligned} \Pr \left\{ \sum_{1 \leq j \leq S} \max_{1 \leq k \leq C} \{X_{j,k}\} \geq \frac{(1+\ell)N}{PB} \right\} &\leq \Pr \left\{ \exists j, \max_{1 \leq k \leq C} \{X_{j,k}\} \geq \frac{(1+\ell)N_j}{PB} \right\} \\ &\leq SC \Pr \left\{ X_{1,1} \geq \frac{(1+\ell)N_1}{PB} \right\}. \end{aligned} \quad (7)$$

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

Lemma 1 *If X is a nonnegative random variable and $r \geq 0$ we have*

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

Before we use Chernoff's bound, we must define more terminology. We let g_t denote the number of clusters written to from bucket S_1 during write \mathcal{W}_t , for $1 \leq t \leq R$, where R is the total number of write cycles used in Phase 1. We have

$$\sum_{1 \leq t \leq R} g_t \leq \frac{N_1}{PB/C} + C = \frac{N_1 C}{PB} + C. \quad (8)$$

The extra C term appears because the last track on each cluster might be only partially filled. We define G_t to be the number of groups belonging to bucket S_1 in write \mathcal{W}_t that are assigned to cluster 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 $\mathcal{G}_{G_t}(z)$ be the probability generating function for G_t :

$$\mathcal{G}_{G_t}(z) = \Pr\{G_t = 0\}z^0 + \Pr\{G_t = 1\}z^1 = 1 - \frac{g_t}{C} + \frac{g_t}{C}z. \quad (9)$$

Let $\mathcal{G}_{X_{1,1}}(z)$ be the probability generating function for $X_{1,1}$. We can bound $X_{1,1}$ by the sum of independent random variables: $X_{1,1} \leq G_1 + G_2 + \dots + G_R$. For purposes of bounding (7), let us consider that $X_{1,1} = G_1 + G_2 + \dots + G_R$. Using (9), we have

$$\mathcal{G}_{X_{1,1}}(z) = \mathcal{G}_{G_1+G_2+\dots+G_R}(z) = \mathcal{G}_{G_1}(z) \times \mathcal{G}_{G_2}(z) \times \dots \times \mathcal{G}_{G_R}(z) = \prod_{1 \leq t \leq R} \left(1 - \frac{g_t}{C} + \frac{g_t}{C}z\right). \quad (10)$$

By Lemma 1, we have

$$\Pr \left\{ X_{1,1} \geq \frac{(1+\ell)N_1}{PB} \right\} \leq \frac{E(\exp(rX_{1,1}))}{\exp\left(\frac{r(1+\ell)N_1}{PB}\right)}, \quad (11)$$

for each $r \geq 0$. Using the above definitions and (10), we have

$$E(\exp(rX_{1,1})) = \mathcal{G}_{X_{1,1}}(e^r) = \prod_{1 \leq t \leq R} \left(1 + \frac{g_t}{C}(e^r - 1)\right). \quad (12)$$

By convexity arguments, we have the following lemma:

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 $a_1 = a_2 = \dots = a_R = Q/R$.*

By (8) and Lemma 2, we can maximize (12) by setting $g_t = \frac{N_1 C}{R P B} + \frac{C}{R}$ for each t . Thus

$$E(\exp(r X_{1,1})) \leq \prod_{1 \leq t \leq R} \left(1 + \frac{(N_1 + P B)(e^r - 1)}{R P B} \right) = \left(1 + \frac{(N_1 + P B)(e^r - 1)}{R P B} \right)^R.$$

Substituting this into (11), we get

$$\Pr \left\{ X_{1,1} \geq \frac{(1 + \ell) N_1}{P B} \right\} \leq \frac{\left(1 + \frac{(N_1 + P B)(e^r - 1)}{R P B} \right)^R}{\exp \left(\frac{r(1 + \ell) N_1}{P B} \right)}.$$

Hence, by (6) and (7)

$$\Pr \left\{ Z \geq \frac{(1 + \ell) N}{P B} \right\} \leq S C \frac{\left(1 + \frac{(N_1 + P B)(e^r - 1)}{R P B} \right)^R}{\exp \left(\frac{r(1 + \ell) N_1}{P B} \right)}. \quad (13)$$

By Lemma 3 in Section 6.3 and the fact that $N \geq \sqrt{M B P} / \ln(M/B)$ in Phase 1, we have

$$N_1 + P B = N_1 \left(1 + \frac{P B}{N_1} \right) \leq N_1 \left(1 + \frac{2 P B S}{N} \right) = N_1(1 + \beta),$$

where $\beta = 2 P B S / N \leq 2 / \ln(M/B)$. Using (13), we have

$$\Pr \left\{ Z \geq \frac{(1 + \ell) N}{P B} \right\} \leq S C \frac{\left(1 + \frac{N_1(1 + \beta)(e^r - 1)}{R P B} \right)^R}{\exp \left(\frac{r(1 + \ell) N_1}{P B} \right)}. \quad (14)$$

Using the bound $(1 + a)^b \leq e^{ab}$, for $a > -1$, we can approximate the numerator in (14) and get

$$\begin{aligned} \Pr \left\{ Z > \frac{(1 + \ell) N}{P B} \right\} &\leq S C \exp \left(\frac{N_1(1 + \beta)(e^r - 1) - N_1 r(1 + \ell)}{P B} \right) \\ &= S C \exp \left(((1 + \beta)(e^r - 1) - r(1 + \ell)) \frac{N_1}{P B} \right). \end{aligned} \quad (15)$$

Picking $r = \ln((1 + \ell)/(1 + \beta))$, we get

$$\Pr \left\{ Z > \frac{(1 + \ell) N}{P B} \right\} \leq S C \exp \left(\left(\ell - \beta - \ln \left(\frac{1 + \ell}{1 + \beta} \right) \right) \frac{N_1}{P B} \right). \quad (16)$$

For small ℓ , plugging into (16) the inequality $\ln(1 + x) \geq x - x^2/2$, for $0 \leq x \leq 1$, we get

$$\Pr \left\{ Z > \frac{(1 + \ell) N}{P B} \right\} \leq S C \exp \left(\left(-\frac{(\ell - \beta)^2}{2(1 + \beta)} + \frac{(\ell - \beta)^3}{2(1 + \beta)^2} \right) \frac{N_1}{P B} \right).$$

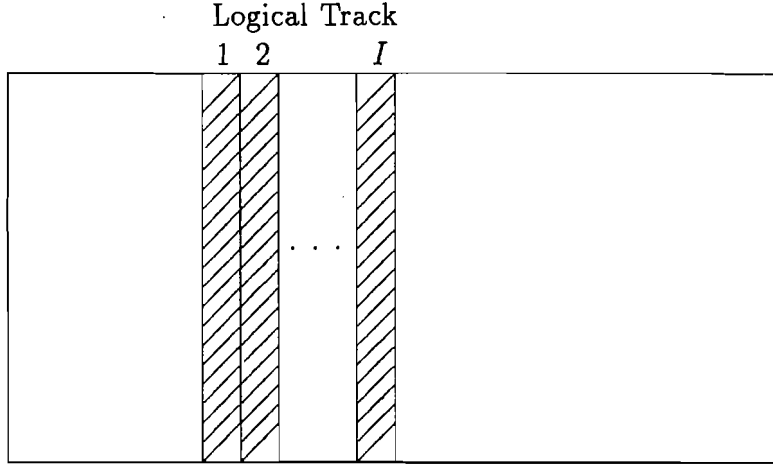


Figure 3: The shaded tracks below represent a logical track.

For example, when $\ell \leq 1/2$, we get the exponentially small bound

$$\Pr \left\{ Z > \frac{(1 + \ell)N}{PB} \right\} \leq SC \exp \left(-\frac{(\ell - \beta)^2}{4(1 + \beta)} \frac{N_1}{PB} \right).$$

For larger ℓ , the probability bound is proportional to $\exp(-\ell(\log \ell)N_1/PB)$. Note that by Lemma 3 we always have $N_1/PB \geq \frac{1}{2}N/PBS \geq \frac{1}{2}\ln(M/B)$; this causes the SC term to be quickly dwarfed. And in the early passes of Phase 1, N_1/PB can be substantially larger. \square

6.2 Phase 2

In Phase 2 we want to sort $N < \sqrt{MBP}/\ln(M/B)$ records in one pass of distribution sort using $O(N/PB)$ I/Os, which is optimal. We use $S = 8N/7M$ partitions. This is the case that cannot be handled by Phase 1, since when N is relatively small the records in each bucket would 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 as a group in I consecutive I/Os. When accessing the i th logical track, we actually access tracks $I(i - 1) + 1, \dots, Ii$. This is pictured in Figure 3.

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

Diagonal 1 is illustrated in Figure 4. It follows from the condition $N < \sqrt{MBP}/\ln(M/B)$

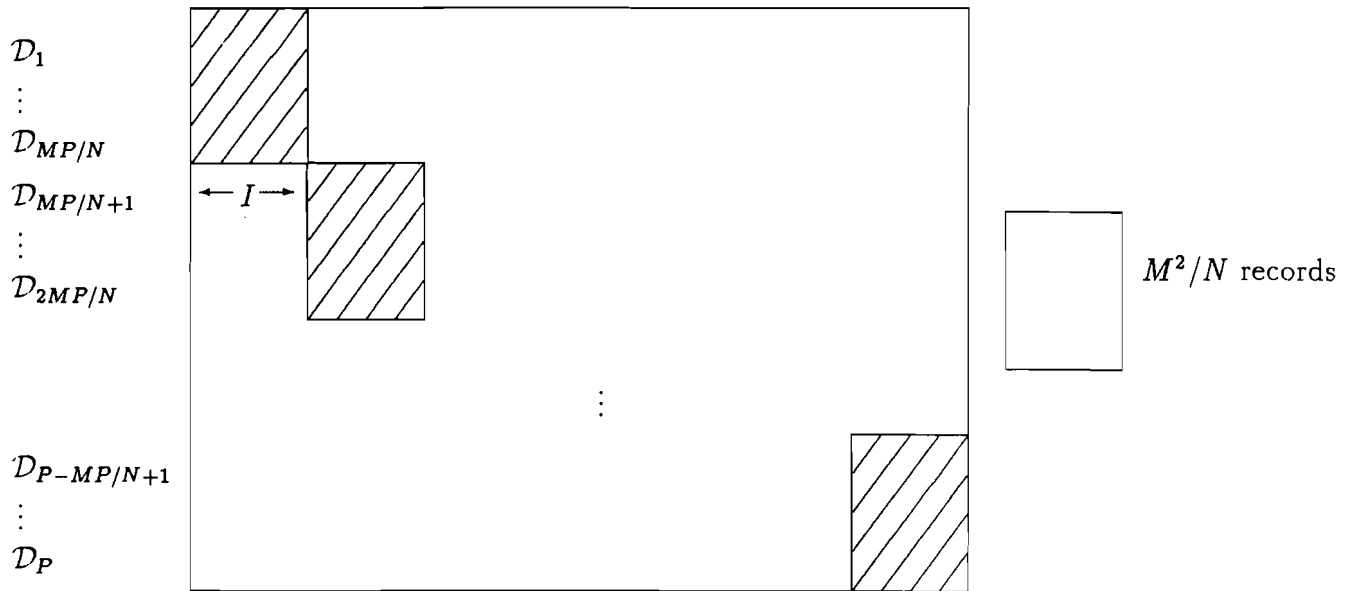


Figure 4: The shaded areas collectively represent diagonal 1.

of Phase 2 that $1 \leq MP/N \leq P$, 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.

Our algorithm consists of two passes:

1. 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.
2. Read in the file, one diagonal (memoryload) at a time. For each memoryload, partition the records into buckets, based on the partitioning elements. (The number of records in each bucket of a memoryload will be very evenly distributed with high probability.) Write one block to disk from each bucket; repeat this process until all the records of the memoryload are written.

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 the process of distributing the buckets among the disks as the problem of routing on the FFT digraph. The N/B nodes in each level of the FFT digraph can be divided up into groups of size M/B each. Each group of M/B nodes is associated with the M/B blocks of a memoryload. The routing internal to one memoryload is represented by moving forward $\log M$ stages in the network. Theorem 9 shows that the fact that each memoryload was randomly scrambled dramatically reduces the probability of “hot spots” during the routing.

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 on the permuted ordering, and we write the blocks back to a logical track.

```

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 on 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  $S_1$ ,
block 1 of  $S_2$ ,
...
block 1 of  $S_S$ ,
block 2 of  $S_1$ ,
block 2 of  $S_2$ ,
...
block 2 of  $S_S$ ,
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 $1, 2, \dots, P$. Let d be the least common multiple of P and S , divided by P . After d write cycles, the order that the disks are written to is cyclically shifted by one.

```

 $d := \text{lcm}(P, S) / P$ ;
 $k := 1$ ;
 $\text{num\_cycles} := 0$ ;
for each diagonal ( $1 \leq i \leq N/M$ ) do
  begin
    read the  $i$ th diagonal into internal memory;
    partition the records into buckets based on the partitioning elements;
    form the records into blocks of size  $B$ ;

```

```

while a nonempty bucket remains do
  begin
    for  $j := 1$  to  $S$  do
      begin
        schedule next block from  $S_j$  to be written on next available track of  $\mathcal{D}_k$ ,
        assigning a dummy block if  $S_j$  is empty;
         $k := (k \bmod P) + 1$ 
      end;
       $\text{num\_cycles} := \text{num\_cycles} + 1$ ;
      if  $\text{num\_cycles} = d$  then
        begin  $\text{num\_cycles} := 0$ ;  $k := (k \bmod P) + 1$  end
      end
    write the memoryload to the desired disks
  end

```

Analysis of Phase 2

Theorem 9 *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 $\frac{8}{7} + \ell$ times the average is exponentially small in $\ell(\log \ell) \log^2(M/B)$.*

Proof: The actual number of records read in each memoryload might be less than M records, since in Pass 2 the $S - 1$ 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 6.2, 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 in during Pass 2 and it is possible that only $M - 1 - (B - 1) = M - B$ records can be read per memoryload during the next pass, 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 6.2 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. 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

$$Z \approx \frac{8N}{7PB}. \quad (17)$$

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 S_j . We have

$$\alpha = \sum_{1 \leq i \leq N/M} \max_{1 \leq j \leq S} \left\{ \left\lceil \frac{Y_{i,j}}{B} \right\rceil \right\}. \quad (18)$$

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

$$Z \leq S \left\lceil \frac{\alpha}{P} \right\rceil. \quad (19)$$

To show that (17) is true, we shall show that

$$\Pr \left\{ Z \geq \frac{(\frac{8}{7} + \ell)N}{PB} \right\} \quad (20)$$

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

Note that Z is expressed in (19) in terms of α , which by (18) 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 $\max_{1 \leq j \leq S} \{ \lceil Y_{i,j}/B \rceil \}$ is with very high probability asymptotically equal to the average value of each $\lceil Y_{i,j}/B \rceil$. We have

$$\begin{aligned} \Pr \left\{ Z \geq \frac{(\frac{8}{7} + \ell)N}{PB} \right\} &\leq \Pr \left\{ S \left\lceil \frac{\alpha}{P} \right\rceil \geq \frac{(\frac{8}{7} + \ell)N}{PB} \right\} \\ &\leq \Pr \left\{ \frac{\alpha}{P} \geq \frac{(\frac{8}{7} + \ell)N}{PBS} - \frac{P-1}{P} \right\} \\ &= \Pr \left\{ \alpha \geq \frac{(\frac{8}{7} + \ell)N}{BS} - (P-1) \right\} \\ &= \Pr \left\{ \sum_{1 \leq i \leq N/M} \max_{1 \leq j \leq S} \left\{ \left\lceil \frac{Y_{i,j}}{B} \right\rceil \right\} \geq \frac{(\frac{8}{7} + \ell)N}{BS} - (P-1) \right\}. \end{aligned}$$

Since the $Y_{i,j}$'s are equally distributed, we have

$$\begin{aligned} \Pr \left\{ \sum_{1 \leq i \leq N/M} \max_{1 \leq j \leq S} \left\{ \left\lceil \frac{Y_{i,j}}{B} \right\rceil \right\} \geq \frac{(\frac{8}{7} + \ell)N}{BS} - (P-1) \right\} \\ \leq \frac{N}{M} \Pr \left\{ \max_{1 \leq j \leq S} \left\{ \left\lceil \frac{Y_{1,j}}{B} \right\rceil \right\} \geq \frac{(\frac{8}{7} + \ell)M}{BS} - \frac{(P-1)M}{N} \right\} \\ \leq \frac{SN}{M} \Pr \left\{ \left\lceil \frac{Y_{1,1}}{B} \right\rceil \geq \frac{(\frac{8}{7} + \ell)M}{BS} - \frac{(P-1)M}{N} \right\} \\ \leq \frac{SN}{M} \Pr \left\{ Y_{1,1} \geq \frac{(\frac{8}{7} + \ell)M}{S} - \frac{(P-1)MB}{N} - (B-1) \right\}. \end{aligned}$$

Let us define ℓ' so that

$$\frac{(\frac{8}{7} + \ell')M}{S} \geq \frac{(\frac{8}{7} + \ell)M}{S} - \frac{(P-1)MB}{N} - (B-1)$$

It is easy to see that $\ell' \geq \ell - 8/7(PB/M + NB/M^2)$. Since $N < \sqrt{MBP} \ln(M/B)$, it follows that ℓ' is at most a small constant amount less than ℓ . We have $\ell' \geq \ell - 8/7(1 + 1/\ln(M/B))$. Substituting this value of ℓ' we get

$$\Pr \left\{ Z \geq \frac{(\frac{8}{7} + \ell)N}{PB} \right\} \leq \frac{SN}{M} \Pr \left\{ Y_{1,1} \geq \frac{(\frac{8}{7} + \ell')M}{S} \right\}. \quad (21)$$

Let $Y_{i,j,k}$ represent the number of records found in memoryload \mathcal{M}_i belonging to bucket S_j read from k th logical track. In particular we have $\sum_{1 \leq k \leq N/M} Y_{1,1,k} = Y_{1,1}$. Let μ_k represent the expected value of $Y_{1,1,k}$. Note that

$$\sum_{1 \leq k \leq N/M} \mu_k = E(Y_{1,1}) \leq \frac{8}{7} \frac{M}{S} = \frac{M^2}{N}.$$

Let $T_{j,k}$ be the number of records belonging to bucket S_j on the k th logical track. We have $\mu_k = T_{1,k}M/N$. We want to bound $\Pr\{Y_{1,1} \geq (8/7 + \ell')M/S\}$. 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 $Y_{1,1,k}$ would be if all the $T_{j,k}$ terms were equal to M^2/N . Let $\delta = \frac{7}{16}\ell' - \frac{1}{2}$. We have

$$\begin{aligned} \Pr\left\{Y_{1,1} \geq \frac{(\frac{8}{7} + \ell')M}{S}\right\} &\leq \sum_{\substack{1 \leq k \leq N/M \\ \mu_k < M^3/N^2}} \Pr\left\{Y_{1,1,k} \geq \mu_k + \delta \frac{M^3}{N^2}\right\} \\ &\quad + \sum_{\substack{1 \leq k \leq N/M \\ \mu_k \geq M^3/N^2}} \Pr\{Y_{1,1,k} \geq \mu_k + \delta \mu_k\}. \end{aligned} \quad (22)$$

The above bound (22) holds since

$$\sum_{\substack{1 \leq k \leq N/M \\ \mu_k < M^3/N^2}} \left(\mu_k + \delta \frac{M^3}{N^2}\right) + \sum_{\substack{1 \leq k \leq N/M \\ \mu_k \geq M^3/N^2}} (\mu_k + \delta \mu_k) \leq (2 + 2\delta) \frac{M^2}{N} = \frac{(\frac{8}{7} + \ell')M}{S}.$$

The probability term in (21) is expressed in (22) as a sum of tails of distributions of $Y_{1,1,k}$, 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 $Y_{1,1,k}$ listed in the summations in (22). Both summands have the form $\Pr\{X \geq \mu + v\}$, where $X = Y_{1,1,k}$, $\mu = \mu_k$ is the mean of X , and v is a positive value. Let $L = M^2/N$ be the number of records read from i th logical track by any memoryload. The random variable $X = Y_{1,1,k}$ has the hypergeometric probability distribution

$$\Pr\{X = l\} = \frac{\binom{T_{1,k}}{l} \binom{M-T_{1,k}}{L-l}}{\binom{M}{L}} \quad (23)$$

with mean

$$\mu = \frac{T_{1,k}L}{M} = \frac{T_{1,k}M}{N}. \quad (24)$$

We define $R(t)$ to be the ratio

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

We have

$$\begin{aligned}
\Pr\{X \geq \mu + v\} &= \Pr\{X = \mu + v\} + \Pr\{X = \mu + v + 1\} + \dots \\
&= \Pr\{X = \mu\} \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}} \\
&\quad + \Pr\{X = \mu + 1\} \frac{\Pr\{X = \mu + v + 1\}}{\Pr\{X = \mu + 1\}} + \dots \quad (25)
\end{aligned}$$

Using (23), 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 into (25), we get

$$\begin{aligned}
\Pr\{X \geq \mu + v\} &\leq \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}} (\Pr\{X = \mu\} + \Pr\{X = \mu + 1\} + \dots) \\
&\leq \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu\}}. \quad (26)
\end{aligned}$$

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

$$\begin{aligned}
\Pr\{X = \mu + v\} &= \Pr\{X = \mu\} \frac{\Pr\{X = \mu + 1\}}{\Pr\{X = \mu\}} \frac{\Pr\{X = \mu + 2\}}{\Pr\{X = \mu + 1\}} \dots \frac{\Pr\{X = \mu + v\}}{\Pr\{X = \mu + v - 1\}} \\
&= \Pr\{X = \mu\} \prod_{0 \leq t \leq v-1} R(\mu + t). \quad (27)
\end{aligned}$$

Using (27) and (26), we get

$$\Pr\{X \geq \mu + v\} \leq \prod_{0 \leq t \leq v-1} R(\mu + t). \quad (28)$$

Using (23), (24), the definition of $R(t)$, and the bounds $0 \leq t \leq T \leq M$, we find after some algebraic manipulation that

$$R(\mu + t) = \frac{(T_{1,k} - (\mu + t))(L - (\mu + t))}{(\mu + t + 1)(M - T_{1,k} - L + \mu + t + 1)} \leq \frac{1}{1 + \frac{t}{\mu + 1}}. \quad (29)$$

Thus, by (28)

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

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

$$\begin{aligned}
\ln \Pr\{X \geq \mu + v\} &\leq \sum_{0 \leq t \leq v-1} \ln \left(\frac{1}{1 + \frac{t}{\mu + 1}} \right) \\
&\leq - \int_0^{v-1} \ln \left(1 + \frac{y}{\mu + 1} \right) dy \\
&= -(\mu + v) \ln \left(\frac{\mu + v}{\mu + 1} \right) + v - 1 \\
&\leq -(\mu + v) \ln \left(\frac{\mu + v}{\mu} \right) + v - 1 - (\mu + v) \ln \left(1 - \frac{1}{\mu + 1} \right). \quad (30)
\end{aligned}$$

Taking the exponential of both sides, we get

$$\Pr\{X \geq \mu + v\} \leq \left(\frac{\mu + v}{\mu}\right)^{-(\mu+v)} e^{v-1-(\mu+v)\ln(1-1/(\mu+1))}. \quad (31)$$

Let us start with the summand of the second sum in (22). By applying (31) and the bound $\ln(1 - 1/(\mu + 1)) \geq -1/\mu$, we get

$$\Pr\{Y_{1,1,k} \geq \mu_k + \delta\mu_k\} \leq (1 + \delta)^{-(1+\delta)\mu_k} e^{\delta\mu_k + \delta}. \quad (32)$$

Since $\mu_k \geq M^3/N^2$, we get by some analysis that (32) is maximized when $\mu_k = M^3/N^2$, and we get

$$\Pr\{Y_{1,1,k} \geq \mu_k + \delta\mu_k\} \leq (1 + \delta)^{-(1+\delta)M^3/N^2} e^{\delta M^3/N^2 + \delta}. \quad (33)$$

Note that in Phase 2 we always have $M^3/N^2 > \ln^2(M/B)$. The bound in (33) 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 $\delta < 1/2$, we can use the bound $\ln(1 + x) \geq x - x^2/2$ to get

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

Similarly, by (31) and the bound $\ln(1 - 1/(\mu + 1)) \geq -1/\mu$, the summand of the first sum in (22) for $\mu_k \geq 1/2$ is

$$\Pr\left\{Y_{1,1,k} \geq \mu_k + \delta \frac{M^3}{N^2}\right\} \leq \left(1 + \delta \frac{M^3/N^2}{\mu_k}\right)^{-(\mu_k + \delta M^3/N^2)} e^{(\delta M^3/N^2)(1+1/\mu_k)}, \quad (34)$$

which can be bounded by an expression similar to the right-hand side of (33). For smaller μ_k , we have from (31)

$$\begin{aligned} \Pr\left\{Y_{1,1,k} \geq \mu_k + \delta \frac{M^3}{N^2}\right\} \\ \leq \left(1 + \delta \frac{M^3/N^2}{\mu_k}\right)^{-(\mu_k + \delta M^3/N^2)} e^{\delta M^3/N^2 - 1 - (\mu_k + \delta M^3/N^2)\ln(\mu/2)}, \end{aligned} \quad (35)$$

which is exponentially decreasing in a similar way.

In conclusion, by (33), (34), (35), and (22), we can bound the probability (21) to be exponentially small in $\ell(\log \ell) \log^2(M/B)$:

$$\Pr\left\{Z \geq \frac{(\frac{8}{7} + \ell)N}{PB}\right\} \leq \frac{16N^3}{7M^3} \left((1 + \delta)^{-(1+\delta)M^3/N^2} e^{\delta M^3/N^2}\right).$$

The $16N^3/7M^3$ term is very quickly dwarfed by the exponentially small term, since $N < \sqrt{MBP}/\ln(M/B)$ in Phase 2. \square

Completing the Sort

After Phase 2 is completed, we can read the blocks belonging to S_j using an optimal number 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 was deterministic. We can guarantee that the records are packed on disk by retaining in internal memory the last partially-filled block and the disk location of the last block written to disk.

To sort bucket S_j , we first compute the disk and track location of the blocks belonging to the bucket. Using full parallelism, we read the blocks from the disks. Each bucket 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 partially-filled block from S_j can be treated as members of S_{j+1} when S_{j+1} is processed.

```

 $k := 1;$ 
for each bucket  $S_j$  ( $1 \leq j \leq S$ ) do
  begin
    determine which tracks/disks to read from;
    read  $S_j$  into internal memory;
    sort 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  $S_j$ 
  end

```

6.3 Finding the Partitioning Elements

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

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

where $b_0 = -\infty$ and $b_S = +\infty$. We need to show that conditions (3) and (4) of Section 6 are satisfied. Inequality (3) is shown in [AgV]. For completeness, we reproduce the algorithm and its analysis.

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 . First we describe a subroutine that uses $O(n/PB)$ I/Os to find the record with the k th smallest key (or simply the k th 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 [BFP]. The number of I/Os required for these operations is $O(n/PB + (n/B)/P + n/M) = O(n/PB)$. 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 list) 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 k th smallest record; the total number of I/Os is $O(n/PB)$.

We now describe how to apply this subroutine to find the S approximate partitioning elements in a set containing N records. Let p and q denote positive integer parameters to be specified later. As above, we start out by sorting N/M memoryloads of records, which can be done with $O(N/PB + (N/B)/P) = O(N/PB)$ I/Os. Let us denote the i th sorted set by \mathcal{M}_i . We construct a new set \mathcal{M}' of size at most N/p consisting of the k pth records (in sorted order) of \mathcal{M}_i , for $1 \leq k \leq M/p$ and $1 \leq i \leq N/M$. Each memoryload of M records contributes $M/p > B$ records to \mathcal{M}' , so these records 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 $p \geq S$, 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 pseudo-code:

```

for each memoryload of records  $\mathcal{M}_i$  ( $1 \leq i \leq N/M$ ) do
  begin
    read  $\mathcal{M}_i$  into internal memory;
    sort 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;
 $\mathcal{M}' := \mathcal{M}'_1 + \dots + \mathcal{M}'_{N/M}$ ;
for  $j := 1$  to  $S - 1$  do
   $b_j :=$  record of rank  $jq$  in  $\mathcal{M}'$ 

```

Lemma 3 *If $p = (S - 1)/4$ and $q = 4N/(S - 1)^2$ in the above algorithm, then condition (3) of Section 6 is true; that is,*

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

Lemma 4 *If $p = M^2/8N$ and $q = 7N/M$ in the above algorithm, then condition (4) of Section 6 is true; that is,*

$$\frac{3M}{4} \leq N_j \leq M.$$

7 Disk Standard Matrix Multiplication

The following is a basic divide and conquer approach for matrix multiplication.

1. If $k \leq \sqrt{M}$, we multiply the matrices internally. Otherwise we do the following steps:

2. We subdivide A and B into $8 \times k/2 \times k/2$ submatrices: A_1 – A_4 and B_1 – B_4 .

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

We reposition the records so that A_1 – A_4 and B_1 – B_4 are each stored in row-major order.

3. We recursively multiply the 8 pairs of submatrices.
4. We add the 4 pairs of submatrices which resulted from the above multiplications, giving C_1 – C_4 .
5. We reposition C_1 – C_4 so that C is stored in row-major order.

We partition secondary memory into 4 parts (which span the P disks), one part for each submatrix. 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 4 blocks that are assigned to be written to the same disk. The number of I/Os needed to recursively multiply the 8 submatrices in Step 3 is $8T(k/2)$. It is easy to see that Step 4 takes $O(k^2/PB)$ I/Os, since all of the submatrices are packed in blocks. Step 5 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 bound from Theorem 4.

8 Algorithms for the Hierarchical Memory Models

In this section we prove Theorems 5 and 6 of Section 4. We give optimal algorithms and the corresponding matching lower bounds for solving the problems of Section 2 in the PHMM and PBT hierarchical memory models. The sorting and FFT algorithms are applications of the algorithms of Section 6 for the two-level model, applied to the algorithms given in [AAC, ACS] for the $P = 1$ case. The optimality of the resulting PHMM and PBT algorithms reflects the fact that the two-level algorithms on which they are based are very efficient in terms of internal computation.

We can think of each of the P memory hierarchies as being organized into discrete levels; for each $k \geq 1$, level k contains 2^{k-1} locations. We restrict our attention to *well-behaved* access costs $f(x)$, in which there are constants c and x_0 such that $f(2x) \leq cf(x)$, for all $x \geq x_0$. For such $f(x)$, access to any location on level k takes $O(f(2^k))$ time. The access cost functions $f(x) = \log x$ and $f(x) = x^\alpha$, for some $\alpha > 0$, are well-behaved.

We shall refer to the P locations, one per hierarchy, at the same relative position in each of the P hierarchies as a *track*, by analogy with the two-level disk model. The base memory level is the track at level 1. The P memory locations in the base memory level are associated with P processors connected by a network that allows the P records stored there to be sorted in $O(\log P)$ time (perhaps via a randomized algorithm) and that allows two $\sqrt{P}/2 \times \sqrt{P}/2$ matrices to be multiplied in $O(\sqrt{P})$ time using the standard algorithm.

8.1 Sorting in PHMM

Logarithmic Access Cost

Let us first consider the access cost function $f(x) = \log x$. By our definition of level, access to any location on level k takes time $\Theta(k)$. The following algorithm sorts optimally in the PHMM model for $P \geq 2$. It is a modified version of the one-hierarchy algorithm given in [AAC]. The key component of our algorithm is our partitioning technique of Section 6, which we use to spread the records in each bucket evenly among the P memory hierarchies so that the next level of recursion can proceed optimally.

1. We assume without loss of generality that the N records are situated in level $\lceil \log(N/P) \rceil + 1$ on the P hierarchies. If $N \leq P$, we sort the file in the base memory level. Otherwise we do the following steps:
2. We subdivide the file of N records into $t = \lceil \min\{\sqrt{N}, N/P\} \rceil$ subsets, $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_t$, each of size $\lfloor N/t \rfloor$ or of size $\lfloor N/t \rfloor + 1$. We sort each \mathcal{G}_i recursively, after bringing its records to level $\lceil \log(N/Pt) \rceil + 1$ of the hierarchical memory.
3. [Find partitioning elements.] We set the number S of partitioning elements as follows: If $N \geq P^2$, we use $S = \lceil \sqrt{N}/\ln N \rceil$; if $P^{3/2}/\ln P \leq N < P^2$, we use $S = \lceil \sqrt{P}/\ln^2 N \rceil$; otherwise we set $S = \lceil 2N/P \rceil$. We form a set A of $\lceil N/\log N \rceil$ elements by adding to A the key value of every $(\lfloor \log N \rfloor)$ th record from each \mathcal{G}_i . We sort A using two-way merge sort with “hierarchy striping,” which is analogous to disk striping. We set the j th partitioning element b_j to be the $(\lfloor jN/(S \log N) \rfloor)$ th smallest element of A . We move the partitioning elements to level $O(\log(S/P) + 1)$.
4. [Phase 1 or Phase 2?] If $N \geq P^{3/2}/\ln P$, we do Step 5 corresponding to Phase 1; otherwise we do Steps 6 and 7, corresponding to Phase 2.
5. [Phase 1.] For each $1 \leq i \leq t$ in sequence, we process \mathcal{G}_i in sorted order. We move P records at a time to the base memory level, and we determine which bucket each record belongs to, by merging in the partitioning elements. The partitioning elements can be processed P elements at a time, so that the merging proceeds optimally; whenever the next track of partitioning elements is needed, it is read into the base memory level. We then randomly scramble the P records and write the records back to level $\lceil \log(N/P) \rceil + 1$. If the record written in the k th hierarchy belongs to the j th bucket S_j , we update the pointers *last_track_{j,k}* and *next_track_k*, which are stored in hierarchy k , so that all the records of each bucket are linked together in the hierarchy, which is necessary for the next recursive application of the algorithm. (These pointers are analogous to the pointers *last_track_{j,k}* and *next_track_k* in the two-level disk algorithm.) We then proceed to Step 8.
6. [Phase 2—Pass 1.] We scramble the N records, P at a time, by reading the file to the base memory level, track by track, randomly permuting the P records there, and writing them back to level $\lceil \log(N/P) \rceil + 1$.
7. [Phase 2—Pass 2.] We move P records at a time to the base memory level, taking the records from the memory in a diagonal fashion, as in Figure 4 for parameter

value $M = P$. We partition the records into buckets by sorting the P records and the partitioning elements. (The number of partitioning elements is $2N/P \leq P$, for $P \geq 4$.) We write the P records back to level $\lceil \log(N/P) \rceil + 1$, cycling through the buckets; if a bucket is empty, a dummy record is written. Let d be the least common multiple of P and S/P . After every d cycles, we skip over the next hierarchy before beginning the next cycle.

8. [Sort recursively.] For each $1 \leq j \leq S$ in sequence, we sort the j th bucket S_j recursively, after bringing the records of S_j to level $\lceil \log(N/SP) \rceil + 1$ of the hierarchical memory. (With high probability, the records in each bucket are distributed evenly among the P hierarchies, and thus each bucket can be accessed in $O((N/SP) \log(N/P))$ time.) The sorted list of N records consists of the concatenated sorted buckets.

Theorem 10 *The time used by the above algorithm to sort $N \geq P$ records in the PHMM model with $f(x) = \log x$ is*

$$O\left(\frac{N}{P} \log N \log\left(\frac{\log N}{\log P}\right)\right)$$

with overwhelming probability. In particular, the probability that the number of I/Os used is more than $1 + \ell$ times the average is exponentially small in $\ell(\log \ell) \log P$.

Proof: Let $T(N)$ be the time used by this algorithm to sort a file of N records. For $N \geq P^2$, the time needed in Step 2 to subdivide the set of N records and sort the subsets \mathcal{G}_i is $\sqrt{N} T(\sqrt{N}) + O((N/P) \log(N/P))$. The time for the two-way merge sort used in Step 3 to sort n elements is $((n/P) \log n (\log(n/P) + \log P)) = ((n/P) \log^2 n)$. Since $n = N/\log N$, the resulting time to find the S partitioning elements is $O((N/P) \log N)$. The time needed to partition the file in Steps 5, 6, and 7 is $O((N/P) \log(N/P) + (N/P) \log P) = O((N/P) \log N)$. The time for sorting the buckets recursively in Step 8 is with high probability $\sum_{1 \leq j \leq S} T(N_j) + O((N/P) \log(N/P))$, where N_j is the size of the j th bucket, $N_j \leq 2N/S$ for each j , and $\sum_{1 \leq j \leq S} N_j = N$. Hence, for $N \geq P^2$, with high probability we have

$$T(N) = \sqrt{N} T(\sqrt{N}) + \sum_{1 \leq j \leq \sqrt{N}/\ln N} T(N_j) + O\left(\frac{N}{P} \log N\right).$$

If $N < P^2$, there will be at most one more application of Phase 1 and of Phase 2, each phase taking $O((N/P) \log N)$ time. The remaining subfiles will have size at most P and can be sorted in the base memory level in time $O((N/P) \log P)$ time. This yields as desired the time bound

$$T(N) = \Theta\left(\frac{N}{P} \log N \log\left(\frac{\log N}{\log P}\right)\right).$$

The probability bounds follow from those in Theorem 7 and its derivation. \square

The algorithm's running time in Theorem 10 matches the following lower bound, and thus the algorithm is optimal:

Theorem 11 *The time required to sort $N \geq P$ records in the PHMM model with $f(x) = \log x$ is*

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

Proof: Let A be a sorting algorithm that is optimal in the PHMM model. Let us define the “sequential time” of A to be the sum of its time costs for each of the P hierarchies; the sequential time of A is at most P times its running time. By [AgV], the I/O complexity of sorting N records with one disk, no blocking, and an internal memory of size M is

$$T_M(N) = \Omega \left(\frac{N \log N}{\log M} - M \right). \quad (36)$$

The “ $-M$ ” term permits M records to reside initially in the internal memory. Following the approach in [AAC], we can imagine superimposing onto the PHMM-type hierarchical memory a sequence of two-level memories with internal memory size M , for $P \leq M < N$. In each hierarchy, every transfer done by A that corresponds to an I/O with respect to an internal memory of size M contributes $\delta f(M/P) = f((M+1)/P) - f(M/P)$ to its sequential time. In other words, if we let $T_f(N)$ denote the sequential time for A , we have

$$T_f(N) \geq \sum_{P \leq M < N} \delta f\left(\frac{M}{P}\right) T_M(N). \quad (37)$$

For $f(x) = \log x$, we have $\delta f(M/P) = \log((M+1)/M) = \Theta(1/M)$. Plugging this and (36) into (37) we get

$$T_f(N) = \Omega \left(\sum_{P \leq M < N} \left(\frac{N \log N}{M \log M} - 1 \right) \right) = \Omega \left(N \log N \log \left(\frac{\log N}{\log P} \right) \right).$$

Dividing the sequential time $T_f(N)$ by P gives us the desired lower bound. \square

Other Access Costs

First we show the lower bound corresponding to case $f(x) = x^\alpha$, for $\alpha > 0$:

Theorem 12 *The time required to sort $N \geq P$ records in the PHMM model with $f(x) = x^\alpha$, for $\alpha > 0$, is*

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

The $(N/P) \log N$ term depends on using the comparison model of computation.

Proof: We apply the same approach as in Theorem 11, except that we use $f(x) = x^\alpha$. Substituting $\delta f(M/P) = \Theta(M^{\alpha-1}/P^\alpha)$ and (36) into (37), we get

$$T_f(N) = \Omega \left(\sum_{P \leq M < N} \left(\frac{M^{\alpha-1} N \log N}{P^\alpha \log M} - \left(\frac{M}{P} \right)^\alpha \right) \right) = \Omega \left(\frac{N^{\alpha+1}}{P^\alpha} \right).$$

Dividing the sequential time $T_f(N)$ by P gives us the first term of the desired lower bound. The second term follows from the $N \log N$ lower bound for sorting in the comparison model of computation. \square

The sorting algorithm given earlier for the logarithmic access cost is *uniformly optimal* (in the language of [AAC]) in that it is optimal for all well-behaved access costs $f(x)$, as defined at the beginning of Section 8:

Theorem 13 *The algorithm given at the beginning of Section 8.1 is optimal for all well-behaved access costs $f(x)$. In particular, its running time meets the lower bound given in Theorem 12 for the case $f(x) = x^\alpha$, where $\alpha > 0$.*

Proof: We use the idea behind the lower bound proofs in Theorems 11 and 12. Let $T_{M,P}(N)$ be the average number of I/Os done by the sorting algorithm with respect to an internal memory of size M , where each hierarchy can simultaneously read or write in a single I/O. From the algorithm, for $N \geq M^2$, we get the recurrence

$$T_{M,P}(N) = \sqrt{N} T_{M,P}(\sqrt{N}) + \sum_{1 \leq j \leq \sqrt{N}/\log N} T_{M,P}(N_j) + \frac{N}{P},$$

with high probability, where $N_j \leq 2N/S$ for each j , and $\sum_{1 \leq j \leq S} N_j = N$. For smaller N we have $T_{M,P}(N) = O(N/P)$. The solution of this recurrence is

$$T_{M,P}(N) = O\left(\frac{N \log N}{P \log M}\right), \quad (38)$$

which by (36) is within an $O(M/P)$ additive term of optimal. The time used by the algorithm in base memory level computations is

$$O\left(\frac{N \log N}{P \log P} \log P\right) = O\left(\frac{N}{P} \log N\right). \quad (39)$$

The extra time used by the algorithm over and above the lower bound resulting from (37) is thus

$$\begin{aligned} O\left(\frac{N}{P} \log N + \sum_{P \leq M < N} \frac{M}{P} \delta f\left(\frac{M}{P}\right)\right) &= O\left(\frac{N}{P} \log N + \frac{N}{P} f\left(\frac{N}{P}\right) - f(1) - \frac{1}{P} \sum_{P < M < N} f\left(\frac{M}{P}\right)\right) \\ &= O\left(\frac{N}{P} \log N + \frac{N}{P} f\left(\frac{N}{P}\right)\right). \end{aligned} \quad (40)$$

The first term in (40) corresponds to the lower bound that arises from the comparison model of computation. The second term in (40) is the time to “touch” all the records in the file (that is, bring all the records at least once to the base memory level) when the access cost $f(x)$ is well-behaved, and thus it is dominated by the lower bound resulting from (37). It follows that the algorithm is optimal. \square

8.2 Standard Matrix Multiplication in PHMM

Upper Bounds

Before we present the optimal standard matrix multiplication algorithm, we must present a lemma which is needed to prove that our algorithm is optimal.

Lemma 5 *The time used to add two $k \times k$ matrices, where $k > P$, is*

$$\begin{aligned} O\left(\frac{k^2}{P} \log \frac{k}{P}\right) & \quad \text{if } f(x) = \log x; \\ O\left(\left(\frac{k^2}{P}\right)^{\alpha+1}\right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha > 0. \end{aligned}$$

Proof Sketch: Two matrices can be added by touching the corresponding elements of the matrices simultaneously, using the naive touching algorithm applied to the hierarchies independently. Once two elements are in base memory level together, they can be added. \square

The algorithm presented in Section 7 can be adapted to run on the PHMM model. The repositioning of the matrices can be done in the same time as the touching problem. We define $T(k)$ to be the time used by the algorithm to multiply two $k \times k$ matrices together. For $f(x) = \log x$, we get from Lemma 5

$$T(k) = 8T\left(\frac{k}{2}\right) + O\left(\frac{k^2}{P} \log \frac{k}{P}\right).$$

Using the stopping condition $T(\sqrt{P}) = \sqrt{P}$, we get the desired upper bound $O(k^3/P)$ of Theorem 5. The upper bounds for the other access cost functions follow by using the other cases of Lemma 5.

Lower Bounds

The standard matrix multiplication algorithm given earlier for the logarithmic access cost is *uniformly optimal*, that is, it is optimal for all well-behaved access cost functions. This can be proved using the same approach as in Theorem 13. By [SaV], the I/O complexity of multiplying two $k \times k$ matrices with one disk, no blocking, and an internal memory of size M is

$$T_M(k^2) = \Omega\left(\frac{k^3}{\sqrt{M}} - M\right). \quad (41)$$

Let $T_{M,P}$ be the average number of I/Os done by the standard matrix multiplication algorithm with respect to an internal memory size of M , where all P hierarchies can simultaneously read or write in a single I/O. From the algorithm, for $k \geq M^2$, we get the recurrence

$$T_{M,P}(k) = 8T_{M,P}\left(\frac{k}{2}\right) + \frac{k^2}{P}.$$

For smaller $k = \sqrt{M}$, we have $T_{M,P}(\sqrt{M}) = M/P$. The solution of this recurrence is

$$T_{M,P}(k) = O\left(\frac{k^3}{P\sqrt{M}}\right),$$

which by (41) is within an $O(M/P)$ additive term of optimal. The time used by the algorithm in base memory level computations is

$$O\left(\frac{k^3}{P\sqrt{P}}\sqrt{P}\right) = O\left(\frac{k^3}{P}\right).$$

Therefore, the extra time used by the algorithm over and above the optimal amount is

$$O\left(\frac{k^3}{P} + \sum_{P \leq M < k^2} \frac{M}{P} \delta f\left(\frac{M}{P}\right)\right) = O\left(\frac{k^3}{P} + \frac{k^2}{P} f\left(\frac{k^2}{P}\right)\right). \quad (42)$$

The first term in the right-hand side of (42) is bounded by the number of operations performed, and the second term is the time required to access all the elements; thus the running time is within a constant factor of optimal.

8.3 Sorting in PBT

Before we present the optimal sorting algorithm, we must prove a lemma which is needed to prove that our algorithm is optimal. This lemma is a modified version of a theorem in [ACS].

Lemma 6 *The time used to merge two sorted lists of $n \geq P$ elements in the PBT model is*

$$\begin{aligned} O\left(\frac{n}{P}(\log^* n + \log P)\right) & \quad \text{if } f(x) = \log x; \\ O\left(\frac{n}{P}(\log \log n + \log P)\right) & \quad \text{if } f(x) = x^\alpha, \quad 0 < \alpha < 1; \\ O\left(\frac{n}{P} \log n\right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha = 1; \\ O\left(\left(\frac{n}{P}\right)^\alpha + \frac{n}{P} \log P\right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha > 1. \end{aligned}$$

Proof Sketch: The lists are stored on the P hierarchies in such a manner that they are striped across the tracks. We merge the two lists one track at a time, accessing all P hierarchies. To do the merging, we use $3P$ stacks, three stacks per hierarchy. A stack can be maintained in each individual hierarchy with an amortized cost per operation of

$$\begin{aligned} O(\log^* n) & \quad \text{if } f(x) = \log x; \\ O(\log \log n) & \quad \text{if } f(x) = x^\alpha, \quad 0 < \alpha < 1; \\ O(\log n) & \quad \text{if } f(x) = x^\alpha, \quad \alpha = 1; \\ O(n^{\alpha-1}) & \quad \text{if } f(x) = x^\alpha, \quad \alpha > 1, \end{aligned}$$

where n is the number of operations [ACS]. The cost of merging two lists of P elements in base memory is $\log P$. \square

Access Cost $f(x) = x^\alpha$, where $0 < \alpha < 1$

Let us first consider the access cost function $f(x) = x^\alpha$, where $0 < \alpha < 1$. We can access the levels within our hierarchies optimally if we read and write x^α elements when we access element x . The following algorithm sorts optimally in the PBT model. It is a modified version of the one-hierarchy algorithm presented in [ACS]. The key component of the algorithm is our use of the partitioning technique of Section 6 to spread the records in each bucket evenly among the P hierarchies.

1. We assume without loss of generality that the N records are situated in level $\lceil \log(N/P) \rceil + 1$ on the P hierarchies. If $N \leq P$, we sort the file in the base memory level. Otherwise we do the following steps:
2. We subdivide the file of N records into $t = \lceil \min\{N^{1-\alpha}, N/P\} \rceil$ subsets $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_t$, each of size $\lfloor N/t \rfloor$ or of size $\lfloor N/t \rfloor + 1$. We sort each \mathcal{G}_i recursively, after bringing its records to level $\lceil \log(N/Pt) \rceil + 1$ of the hierarchical memory.

3. [Find partitioning elements.] We set the number S of partitioning elements as follows: If $N \geq P^2$, we use $S = \lceil N^\alpha / \ln N \rceil$; if $P^{3/2} / \ln P \leq N < P^2$, we use $S = \lceil \sqrt{P} / \ln^2 N \rceil$; otherwise we set $S = \lceil 2N/P \rceil$. We form a set A which consists of the key value of every $(\lfloor \log N \rfloor)$ th record from each \mathcal{G}_i . We form A by touching and accumulating the desired key values recursively, with each hierarchy processing up to $\lceil N/P \rceil$ elements of the N elements in the file. We sort A using two-way merge sort by recursively applying the algorithm presented in Lemma 6. We set the j th partitioning element b_j to be the $(\lfloor jN/(S \log N) \rfloor)$ th smallest element of A . We move the partitioning elements to level $\lceil \log(S/P) \rceil + 1$.
4. [Phase 1 or Phase 2?] If $N \geq P^{3/2} / \ln P$, we do Step 5 corresponding to Phase 1; otherwise we do Steps 6 and 7, corresponding to Phase 2.
5. [Phase 1.] For each $1 \leq i \leq t$ in sequence, we process the i th subset \mathcal{G}_i in sorted order. We move P records at a time to the base memory level, and we determine which bucket each record belongs to, by merging in the partitioning elements. The partitioning elements can be processed P elements at a time, so that the merging proceeds optimally; whenever the next track of partitioning elements is needed, it is read into the base memory level. We then randomly scramble the P records and write the records back to level $\lceil \log(N/P) \rceil + 1$. If the group of records written in the k th hierarchy belongs to the j th bucket S_j , we update the pointers $last_track_{j,k}$ and $next_track_k$, which are stored in hierarchy k , so that all the records of each bucket are linked together in the hierarchy, which is necessary for the next recursive application of the algorithm. (These pointers are analogous to the pointers $last_track_{j,k}$ and $next_track_k$ in the two-level disk algorithm.) We then proceed to Step 8.
6. [Phase 2—Pass 1.] We scramble the N records, P at a time, by reading the file to the base memory level, track by track, randomly permuting the P records there, and writing them back to level $\lceil \log(N/P) \rceil + 1$.
7. [Phase 2—Pass 2.] We move P records at a time to the base memory level, taking the records from the memory in a diagonal fashion, as in Figure 4 for parameter value $M = P$. We partition the records into buckets by sorting the P records and the partitioning elements. (The number of partitioning elements is $2N/P \leq P$, for $P \geq 4$.) We write the P records back to level $\lceil \log(N/P) \rceil + 1$, cycling through the buckets; if a bucket is empty, a dummy record is written. Let d be the least common multiple of P and S_j/P . After every d cycles, we skip over the next hierarchy before beginning the next cycle.
8. [Reposition buckets.] For each $1 \leq j \leq S$ in sequence, we reposition the elements within the j th bucket S_j so that they are stored in contiguous memory locations in each hierarchy.
9. [Sort recursively.] For each $1 \leq j \leq S$ in sequence, we sort the j th bucket S_j recursively, after bringing the records of S_j to faster memory. (With high probability, the records in each bucket are distributed evenly among the P hierarchies.) The sorted list of N records consists of the concatenated sorted buckets.

Theorem 14 *The time used by the above algorithm to sort $N \geq P$ records in the PBT model with $f(x) = x^\alpha$, for $0 < \alpha < 1$, is*

$$O\left(\frac{N}{P} \log N\right)$$

with overwhelming probability. In particular, the probability that the number of I/Os used is more than $1 + \ell$ times the average is exponentially small in $\ell(\log \ell) \log P$.

Proof: Let $T(N)$ be the time used by this algorithm to sort a file of N records. For $N \geq P^2$, the time needed in Step 2 to subdivide the set of N records, move the subsets to faster memory, and sort the subsets G_i is $N^{1-\alpha}T(N^\alpha) + O(N/P^\alpha)$. The time for the touching and accumulating of the $N/\log N$ elements of set A in Step 3 is $((N/P) \log \log(N/P))$. Using Lemma 6, we see the time for the two-way merge sort used in Step 3 to sort n elements is $((n/P) \log n (\log \log n + \log P))$. Since $n = N/\log N$, the resulting time to find the S partitioning elements is $O((N/P)(\log \log N + \log P))$. The time needed to partition the file in Steps 5, 6, and 7 is $O(N^{1-\alpha}(N^\alpha/P)(\log \log N^\alpha + \log P)) = O((N/P) \log \log N + (N/P) \log P)$, by Lemma 6. The data movement in Step 8 can be done by the same method used by the one-hierarchy algorithm [ACS], that is, by computing the generalized matrix transposition for each hierarchy independently. The time needed to reposition the buckets in Step 8 is thus $O((N/P)(\log \log(N/P))^4)$. The time for sorting the buckets recursively in Step 9 is with high probability $\sum_{1 \leq j \leq S} T(N_j) + O((N/P) \log \log N)$, where N_j is the size of the j th bucket S_j , $\sum_{1 \leq j \leq S} N_j = N$, and $N_j \leq 2N/S$ for each j . Hence, for $N \geq P^2$, with high probability we have

$$T(N) = N^{1-\alpha}T(N^\alpha) + \sum_{1 \leq j \leq N^\alpha/\log N} T(N_j) + O\left(\frac{N}{P}(\log \log N)^4 + \frac{N}{P} \log P\right),$$

and $N_j \leq 2N^{1-\alpha} \log N$ for each j . If $N < P^2$, there will be at most one more application of Phase 1 and of Phase 2, each phase taking $O((N/P) \log N)$ time. The remaining subfiles will have size at most P and can be sorted in the base memory level in time $O((N/P) \log P)$ time. This yields as desired the time bound

$$T(N) = \Theta\left(\frac{N}{P} \log N\right).$$

The probability bounds follow directly from those in Theorem 7. □

A lower bound of $\Omega((N/P) \log P)$ time for sorting with $f(x) = x^\alpha$, for $0 < \alpha < 1$, follows from the well-known lower bound for sorting in a RAM under the comparison model of computation.

Other Access Cost Functions

Since the above algorithm is optimal for the access cost function $f(x) = x^{1/2}$, it is also optimal for $f(x) = \log x$.

Now let us consider the access cost function $f(x) = x^\alpha$, where $\alpha > 0$. For the case $\alpha \geq 1$, we use a simple application of divide-and-conquer merge sort. The upper bound of Theorem 6 follows by using the algorithm of Lemma 6 for merging two sorted lists.

The lower bound in Theorem 6 for the case $\alpha = 1$ follows by a modification of the argument in [ACS] for $P = 1$. The first term in the lower bound in Theorem 6 for the case $\alpha > 1$ is the time needed to access the farthest elements in memory, and the second term is dominated by the lower bound for $\alpha = 1$.

8.4 Standard Matrix Multiplication in PBT

Before we present the optimal standard matrix multiplication algorithm, we must present the following lemma, which is needed to prove that our algorithm is optimal.

Lemma 7 *The time used to add two $k \times k$ matrices, where $k > P$, is*

$$\begin{aligned} O\left(\frac{k^2}{P} \log^* \frac{k^2}{P}\right) & \quad \text{if } f(x) = \log x; \\ O\left(\frac{k^2}{P} \log \log k\right) & \quad \text{if } f(x) = x^\alpha, \quad 0 < \alpha < 1; \\ O\left(\frac{k^2}{P} \log k\right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha = 1; \\ O\left(\left(\frac{k^2}{P}\right)^\alpha\right) & \quad \text{if } f(x) = x^\alpha, \quad \alpha > 1. \end{aligned}$$

Proof Sketch: We apply the same approach as in Lemma 5. Two matrices can be added by touching the corresponding elements of the matrices simultaneously, using the touching algorithm of [AAC] applied to the hierarchies independently. Once two elements are in base memory level together, they can be added. The resultant matrix moves to slower memory in the same manner as the two matrices being added moved to faster memory, only backwards. \square

Let us consider the access cost function $f(x) = x^\alpha$, where $0 < \alpha < 1$. The algorithm presented in Section 7 can be adapted to run on the PBT model. The repositioning of the matrices can be done in the same time as the touching problem. We define $T(k)$ to be the time used by the algorithm to multiply two $k \times k$ matrices together. It is easy to see that

$$T(k) = 8T\left(\frac{k}{2}\right) + O\left(\frac{k^2}{P} \log \log k\right).$$

Using the stopping condition $T(\sqrt{P}) = \sqrt{P}$, we get the desired upper bound $O(k^3/P)$ of Theorem 6 for the access cost function $f(x) = x^\alpha$, where $0 < \alpha < 1$. The lower bound of $\Omega(k^3/P)$ clearly holds since the number of operations performed is $\Theta(k^3)$.

Since the above algorithm is optimal for the access cost function $f(x) = x^{1/2}$, it is also optimal for $f(x) = \log x$. The upper bounds for the remaining cases of Theorem 6 follow by applying the other cases of Lemma 7. The lower bound for the access cost function $f(x) = x^\alpha$, where $\alpha = 3/2$, for the BT model [ACS] can be modified for the PBT model. When $\alpha > \frac{3}{2}$ we get a lower bound of $\Omega((k^2/P)^\alpha)$ since that is the time needed to access the farthest elements in memory.

9 Conclusions

In this paper we have introduced new and realistic two-level and hierarchical memory models for parallel block transfer in internal memory and secondary storage. For each model we present 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 on a randomized version of distribution sort. The partitioning is done by a combination of two interesting probabilistic techniques in order to guarantee that accesses are spread uniformly over the disks.

A tantalizing open question is whether there is an optimal deterministic sorting algorithm. Preliminary work suggests that the amount of randomness in our distribution sort algorithm can be reduced by applying universal hashing [CaW] in an interesting way. But the problem of removing randomness completely is a fundamental one.

The study of I/O efficiency has many applications besides the ones we studied in this paper. For example, graphics applications 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. Our sorting approach also applies to other hierarchical memory models, such as the uniform memory hierarchies of [ACF].

Addendum

An optimal deterministic sorting algorithm was recently developed by Nodine and Vitter [NoV], settling the open problem listed above.

References

- [AAC] A. Aggarwal, B. Alpern, A. K. Chandra & M. Snir, "A Model for Hierarchical Memory," IBM Watson Research Center, Technical Report RC 15118, October 1989, also appears in *Proceedings of 19th Annual ACM Symposium on Theory of Computing*, New York (May 1987), 305-314.
- [ACS] A. Aggarwal, A. Chandra & M. Snir, "Hierarchical Memory with Block Transfer," *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science*, Los Angeles, CA (October 1987).
- [AgV] A. Aggarwal & J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* (September 1988), also appears in *Proceedings of 14th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987.
- [ACF] B. Alpern, L. Carter & E. Feig, "Uniform Memory Hierarchies," November 1989, manuscript.
- [BFP] M. Blum, R. Floyd, V. Pratt, R. Rivest & R. E. Tarjan, "Time Bounds for Selection," in *Complexity of Computer Calculations*, Miller & Thatcher, eds., Plenum, NY, 1973, 105-109.

- [CaW] J. L. Carter & M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences* 18 (April 1979), 143–154, also appears in *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, (May 1977), 106–112.
- [Kle] L. Kleinrock, in *Queueing Systems, Volume I: Theory*, Wiley and Sons, New York, 1979.
- [Knu] D. Knuth, in *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [LiV] E. E. Lindstrom & J. S. Vitter, "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Transactions on Computers* C-34 (March 1985), 218–233.
- [NoV] M. H. Nodine & J. S. Vitter, "Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks," Department of Computer Science, Brown University, Technical Report CS-90-04, February 1990.
- [ReV] J. H. Reif & L. G. Valiant, "A Logarithmic Time Sort on Linear Size Networks," *Journal of the ACM* 34 (January 1987), 60–76, also appears in *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, Boston (April 1983), 10–16.
- [SaV] J. Savage & J. S. Vitter, "Parallelism in Space-Time Tradeoffs," in *Advances in Computing Research, Volume 4*, F. P. Preparata, ed., JAI Press, 1987, 117–146, also appears in *Proceedings of the International Workshop on Parallel Computing and VLSI*, Amalfi, Italy (May 1984), P. Bertolazzi and F. Luccio, ed., Elsevier Science Press, 1985, 49–58.
- [Sto] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers* C-20 (February 1971), 153–161.
- [ViF] J. S. Vitter & Ph. Flajolet, "Average-Case Analysis of Algorithms and Data Structures," in *Handbook of Theoretical Computer Science*, Jan van Leeuwen, ed., North-Holland, 1990.
- [WuF] C. Wu & T. Feng, "The Universality of the Shuffle-Exchange Network," *IEEE Transactions on Computers* C-30 (May 1981), 324–332.