Algorithms and Data Structures

David Shmoys Editor

The Input/Output Complexity of Sorting and Related Problems

ALOK AGGARWAL and JEFFREY SCOTT VITTER

ABSTRACT: We provide tight upper and lower bounds, up to a constant factor, for the number of inputs and outputs (I/Os) between internal memory and secondary storage required for five sorting-related problems: sorting, the fast Fourier transform (FFT), permutation networks, permuting, and matrix transposition. The bounds hold both in the worst case and in the average case, and in several situations the constant factors match. Secondary storage is modeled as a magnetic disk capable of transferring P blocks each containing B records in a single time unit; the records in each block must be input from or output to B contiguous locations on the disk. We give two optimal algorithms for the problems, which are variants of merge sorting and distribution sorting. In particular we show for P = 1 that the standard merge sorting algorithm is an optimal external sorting method, up to a constant factor in the number of I/Os. Our sorting algorithms use the same number of I/Os as does the permutation phase of key sorting, except when the internal memory size is extremely small, thus affirming the popular adage that key sorting is not faster. We also give a simpler and more direct derivation of Hong and Kung's lower bound for the FFT for the special case $B = P = O(1)$.

1. INTRODUCTION

The problem of how to sort efficiently has strong practical and theoretical merit and has motivated many studies in the analysis of algorithms and computational complexity. Recent studies [8] confirm that sorting continues to account for roughly one-fourth of all computer cycles. Much of those resources are consumed by external sorts, in which the file is too large to fit in internal memory and must reside in secondary storage (typically on magnetic disks). It is well documented that the bottleneck in external sorting is the time for input/output (I/O) between internal memory and secondary storage.

Sorts of extremely large size are becoming more and more common. For example, banks each night typically sort the checks of the current day into increasing order by account number. Then the accounting files can be updated in a single linear pass through the sorted file. In many cases, banks are required to complete this processing before opening for the next business day. Lindstrom and Vitter [8] point out that a typical sort from a few years ago might involve a file of two million records, totaling 800 megabytes, and take 1–2 hours; but in the near future typical file sizes are expected to contain ten million records, totaling 10,000 megabytes, and current sorting methods would take most of one day to do the sorting. (Banks would then have trouble completing this processing before the next business day!)

Two alternatives for coping with this problem present themselves. One approach is to relax the problem requirements and to investigate alternate computer architectures such as parallel or distributed systems, as done in [8], for example. The other approach, which we take in this article, is to examine the fundamental limits in terms of the number of I/Os for external sorting and related problems in current computing environments. We assume that there is a single central processing unit, and we model secondary storage as a generalized random-access magnetic disk. (For completeness, we also consider the case in which the disk has some parallel capabilities.)

Our parameters are

N = # records to sort;
M = # records that can fit into internal memory;
B = # records that can be transferred in a single block;
P = # blocks that can be transferred concurrently;

where $1 \le B \le M < N$ and $1 \le P \le \lfloor M/B \rfloor$. We denote the N records by $R_1, R_2, \ldots, R_N$. The parameters N, M, and B are referred to as the *file size*, *memory size*, and *block size*, respectively. Typical parameters for the two sorting examples mentioned earlier are $N = 2 \times 10^6$, $M = 2000$, $B = 100$, $P = 1$, and $N= 10^7$, $M = 3000$, $B = 50$, $P = 1$.

Each block transfer is allowed to access any contiguous group of B records on the disk. Parallelism is inherent in the problem in two ways: Each block can transfer B records at once, which models the well-known fact that a conventional disk can transfer a block of data via an I/O roughly as fast as it can transfer a single bit. The second factor is that there can be P block transfers at the same time, which partly models special features that the disk might potentially have, such as multiple I/O channels and read/write heads and an ability to access records in noncontiguous locations on disk in a single I/O.

Pioneering work in this area was done by Floyd [3], who demonstrated matching upper and lower bounds of $\Theta((N \log N)/B)$ I/Os for the problem of matrix transposition for the special case $P = O(1)$, $B = \Theta(M) = \Theta(N^c)$, where c is a constant $0 < c < 1$. Floyd's lower bound for transposition also applied to the problems of permuting and sorting (since they are more general problems), and the bound matched the number of I/Os used by merge sort. For these restricted values of M, B, and P, the bound showed that essentially $\Omega(\log N)$ passes are needed to sort the file (since each pass takes $O(N/B)$ I/Os), and that merge sorting and the permutation phase of key sorting both perform the optimum number of I/Os. However, for other values of B, M, and P, Floyd's upper and lower bounds did not match, thus leaving open the general question of the I/O complexity of sorting.

In this article we present optimal bounds, up to a constant factor, for all values of M, B, and P for the following five sorting-related problems: sorting, fast Fourier transform (FFT), permutation networks, permuting, and matrix transposition. We show that under mild restrictions the constant factors implicit in our upper and lower bounds are often equal. The five problems are similar, but the lower bounds require different bents, which illustrate precisely the relation of the five problems to one another. The upper bounds can be obtained by a variant of merge sort with P-block lookahead forecasting and by a distribution-sorting algorithm that uses a median finding subroutine. In particular, we can conclude that the dominant part of sorting, in terms of the number of I/Os, is the rearranging of the records, not determining their order, except when M is extremely small with respect to N. Thus, the permutation phase of key sorting typically requires as many I/Os as does general sorting.

Our results answer the pebbling questions posed in [9] concerning the optimum I/O time needed to perform the computation implied by the FFT directed graph (also called the butterfly or shuffle-exchange or Omega network). For lagniappe, we also give a simple direct proof of the lower bound for FFT when $B = P = O(1)$, which was previously proved in [4] using a complicated pebbling argument.

2. PROBLEM DEFINITIONS

We can picture the internal memory and secondary storage disk together as *extended memory*, consisting of a large array containing at least $M + N$ locations, each location capable of storing a single record. We arbitrarily number the M locations in internal memory by $x[1], x[2], \ldots, x[M]$ and the locations on the disk by $x[M + 1], x[M + 2], \ldots$. The five problems can be phrased as follows:

Sorting

Problem Instance: The internal memory is empty, and the N records reside at the beginning of the disk; that is, $x[i] = \textbf{nil}$, for $1 \le i \le M$, and $x[M + i] = R_i$, for $1 \le i \le N$.

Goal: The internal memory is empty, and the N records reside at the beginning of the disk in sorted nondecreasing order; that is, $x[i] = \textbf{nil}$, for $1 \le i \le M$, and the records in $x[M + 1], x[M + 2], \ldots, x[M + N]$ are ordered in nondecreasing order by their key values.

Fast Fourier Transform (FFT)

Problem Instance: Let N be a power of 2. The internal memory is empty, and the N records reside at the beginning of the disk; that is, $x[i] = \textbf{nil}$, for $1 \le i \le M$, and $x[M + i] = R_i$, for $1 \le i \le N$.

Goal: The N output nodes of the FFT directed graph (digraph) are "pebbled" (to be explained below) and the memory configuration is exactly as in the original problem instance.

The FFT digraph and its underlying recursive construction are shown in Figure 1 for the case $N = 16$. It 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. (Unless explicitly speci-

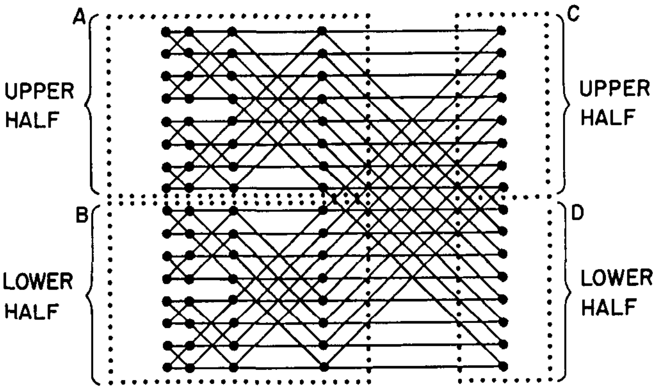

FIGURE 1. The FFT digraph for $N = 16$. Column 0 on the left contains the N input nodes, and column $\log N$ on the right contains the N output nodes. All edges are directed from left to right. The N-input FFT digraph can be recursively decomposed into two $N/2$-input FFT digraphs A and B followed by one extra column of nodes to which the output nodes of A and B are connected in a shuffle-like fashion.

fied, the base of the logarithm is 2.) Each non-input node has indegree 2, and each non-output node has outdegree 2. The FFT digraph is also known as the butterfly or shuffle-exchange or Omega network.

We shall denote the ith node ($0 \le i \le N-1$) in column j ($0 \le j \le \log N$) in the FFT digraph by $n_{i,j}$. 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. (Note that nodes $n_{i,j}$ and $n_{i\oplus 2^{j-1},j}$ each have the same two predecessors). The ith node in each column corresponds to record R_i. We can pebble node $n_{i,j}$ if its two predecessors have already been pebbled and if the records corresponding to those 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 Network

The *Problem Instance* and *Goal* are phrased the same as for FFT, except that the permutation network digraph is pebbled rather than the FFT digraph.

A complete description of permutation networks appears in [5]. A permutation network digraph consists of $J+1$ columns, for some $J \ge \log N$, 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. We denote the ith node in column j as $n_{i,j}$. For each $1 \le i \le N$, $1 \le j \le J$, 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 when that is the 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. In that case, 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 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 $p_1, p_2, \ldots, p_N$ 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 ith 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.

Permuting

The *Problem Instance* and *Goal* are the same as for sorting, except the key values of the N records are required to form a permutation of $\{1, 2, \ldots, N\}$.

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

Permuting is the second (and typically dominant) component of key sorting. The first component of key sorting consists of stripping away the key values of the records and sorting the keys. Ideally, the keys are small enough so that this sort can be done in internal memory and thus very quickly. In the second component of key sorting, the records are routed to their final positions based upon the ordering determined by the sorting of the keys.

Matrix Transposition

Problem Instance: A $p \times q$ matrix $A = (A_{i,j})$ of $N = pq$ records stored in row-major order on disk. The internal memory is empty, and the N records reside in row-major order at the beginning of the disk; that is, $x[i] =$ **nil**, for $1 \le i \le M$, and $x[M+1+i] = A_{1+\lfloor i/q \rfloor, 1+i-q\lfloor i/q \rfloor}$, for $0 \le i \le N-1$.

Goal: The internal memory is empty, and the transposed matrix A^{T} resides on disk in row-major order. (The $q \times p$ matrix A^{T} is called the transpose of A if $A^{\mathrm{T}}_{i,j} = A_{j,i}$, for all $1 \le i \le q$ and $1 \le j \le p$.) An equivalent formulation is for the original matrix A to reside in column-major order on disk.

3. THE MAIN RESULTS

Our model requires that each block transfer in an input can move at most B records from disk into internal memory, and that the transferred records must come from a contiguous segment $x[M+i], x[M+i+1], \ldots, x[M+i+B-1]$ of B locations on the disk, for some $i > 0$; similarly, in each output the transferred records must be deposited within a contiguous segment of B locations. We assume that the records are indivisible; that is, records are transferred in their entirety, and bit manipulations like exclusive-oring are not allowed.

Our characterization of the I/O complexity for the five problems is given in the following three main theorems. The constant factors implicit in the bounds are discussed at the end of the section.

THEOREM 3.1. *The average-case and worst-case number of I/Os required for sorting N records and for computing the N-input FFT digraph is*

$$\Theta\left(\frac{N}{PB}\,\frac{\log(1+N/B)}{\log(1+M/B)}\right). \qquad (3.1)$$

For the sorting lower bound, the comparison model is used, but only for the case when M and B are extremely small with respect to N, namely, when $B\log(1+M/B) = o(\log(1+N/B))$. *The average-case and worst-case number of I/Os required for computing any N-input permutation network is*

$$\Omega\left(\frac{N}{PB}\,\frac{\log(1+N/B)}{\log(1+M/B)}\right); \qquad (3.2)$$

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

$$O\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right). \tag{3.3}$$

THEOREM 3.2. *The average-case and worst-case number of I/Os required to permute N records is*

$$\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right\}\right). \tag{3.4}$$

It is interesting to note that the optimum bound for sorting in Theorem 3.1 matches the second of the two terms being minimized in Theorem 3.2. When the second term in (3.4) achieves the minimum, which happens except when M and B are extremely small with respect to N, the problem of permuting is as hard as the more general problem of sorting; the dominant component of sorting in this case, in terms of the number of I/Os, is the routing of the records, not the determination of their order. When instead M and B are extremely small (namely, when $B\log(1+M/B) = o(\log(1+N/B))$), the N/P term in (3.4) achieves the minimum, and the optimum algorithm for permuting is to move the records in the naïve manner, one record per block transfer. This is precisely the case where advance knowledge of the output permutation makes the problem of permuting easier than sorting. The lower bound for sorting in Theorem 3.1 for this case requires the use of the comparison model.

An interesting corollary comes from applying the bound for sorting in Theorem 3.1 to the case $M = 2$ and $B = P = 1$, where the number of I/Os corresponds to the number of comparisons needed to sort N records by a comparison-based internal sorting algorithm. Substituting $M = 2$ and $B = P = 1$ into (3.1) gives the well-known $\Theta(N\log N)$ bound.

THEOREM 3.3. *The number of I/Os required to transpose a $p \times q$ matrix stored in row-major order, is*

$$\Theta\left(\frac{N}{PB}\frac{\log\min\{M, 1+\min\{p, q\}, 1+N/B\}}{\log(1+M/B)}\right). \tag{3.5}$$

When B is large, matrix transposition is as hard as general sorting, but for smaller B, the special structure of the transposition permutation makes transposing easier.

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

The algorithms we use in Section 5 to achieve the upper bounds in the above theorems follow a more restrictive model of I/O, in which all I/Os are "simple" and respect track boundaries.

DEFINITION 3.1. We call an input *simple* if each record transferred from disk is removed from the disk and deposited into an empty location in internal memory; similarly, an output is simple if the transferred records are removed from internal memory and deposited into empty locations on disk.

DEFINITION 3.2. We denote the kth set ($k \geq 1$) of B contiguous locations on the disk, namely, locations $x[M+(k-1)B+1], x[M+(k-1)B+2], \ldots, x[M+kB]$, as the *kth track.*

Each I/O performed by our algorithms transfers exactly B records, corresponding to a complete track. (Some records may be **nil** if the track is not full.) These assumptions are typically met (or could easily be met) in practical implementations.

If we enforce these assumptions and consider the case $P = 1$, which corresponds to conventional disks, the resulting lower bounds and upper bounds can be made *asymptotically tight*; that is, the constant factors implicit in the O and Ω bounds in the above theorems are the same: If $M = N^c$, $B = M^d$, $P = 1$, for some constants $0 < c, d < 1$, the average-case lower bound for permuting and sorting and the number of I/Os used by merge sort are both asymptotically $2(1-cd)/(c(1-d))N^{1-cd}$, which is a linear function of $2N/B$, the number of I/Os per pass. For example, if $M = \sqrt{N}$, $B = \sqrt{M}$, $P = 1$, the number of I/Os is asymptotically $4N^{3/4}$, which corresponds to two passes over the file to do the sort. If $M = N^c$, $B = M/\log M$, $P = 1$, the bounds are each asymptotically $2c(1-c)N^{1-c}\log^2 N/\log\log N$, and the number of passes is $\Theta(\log N/\log\log N)$. When $M = \sqrt{N}$, $B = ½M$, $P = 1$, the worst-case upper and lower bounds are asymptotically $2\sqrt{N}\log N$, which corresponds to ½ log N passes. In the above three examples, if $B = \Omega(\sqrt{N}/\log^b N)$, for some b, the permutation corresponding to the transposition of a $B \times N/B$ matrix is a worst-case permutation for the permuting and sorting problems.

The restrictions adhered to by our algorithms allow our upper bounds to apply to the pebbling-based model of I/O defined by Savage and Vitter [9]. Our results answer some of the open questions posed there for sorting and FFT by providing tight upper and lower bounds. The model in [9] corresponds to our model for $P = 1$ with the restriction that only records that were output together in a single block can be input together in a single block.

4. PROOF OF THE LOWER BOUNDS

Without loss of generality, we assume that B, M, and N are powers of 2 and that $B < M < N$. We shall consider the case $P = 1$ when there is only one I/O at a time; the general lower bound will follow by dividing the bound we obtain by P. For the average-case analysis of permuting and sorting, we assume that all $N!$ inputs are equally likely. The FFT, permutation network, and matrix transposition problems have no input distribution, so the average-case and worst-case models are the same.

Permuting

First we prove a useful lemma, which applies not only to permuting but also to the other problems. It allows us to assume, for purposes of obtaining the lower bound, that I/Os are simple (see Definition 3.1) and thus that exactly one copy of each record is present throughout the execution of the algorithm.

Lemma 4.1. *For each computation that implements a permutation of the N records $R_1, R_2, \ldots, R_N$ (or that sorts or that transposes or that computes the FFT digraph or a permutation network), there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.*

Proof. It is easy to construct the simple computation strategy by working backwards. We cancel the transfer of a record if its transfer is not needed for the final result. The resulting I/O strategy is simple.

Our approach is to bound the number of possible permutations that can be generated by t I/Os. If we take the value of t for which the bound reaches $N!$, we get a lower bound on the worst-case number of I/Os. We can get a lower bound on the average case in a similar way.

Definition 4.1. We say that a permutation $p_1, p_2, \ldots, p_N$ of the N records can be generated at time t if there is some sequence of t I/Os such that after the I/Os, the records appear in the correct permuted order in extended memory; that is, for all i, j, and k, we have

$$x[i] = R_{p_k} \quad \text{and} \quad x[j] = R_{p_{k+1}} \Rightarrow i < j.$$

The records do not have to be in contiguous positions in internal memory or on disk; there can be arbitrarily many empty locations between R_{p_k} and $R_{p_{k+1}}$.

As mentioned above, we assume that I/Os are simple. We also make the following assumption, which does not increase the number of I/Os by more than a small constant factor. We require that each input and output transfer exactly B records, some of the records being possibly **nil**, and that the B records come from or go to a single track. For example, an input of $b < B$ records, with b_1 records from one track and $b_2 = b - b_1$ records from the next track, can be simulated using an internal memory of size $M + B$ by an input of the first track, an output of the $B - b_1$ records that are not needed (plus an additional b_1 **nil** records to take the place of the b_1 desired records), and then a corresponding input and output for the next track. As a consequence, since I/Os are simple, a track immediately after an input or immediately before an output must be empty. We do not count internal computation time in our complexity model, so we can assume that the optimum algorithm, between I/Os, rearranges the records in internal memory however it sees appropriate.

Initially, the number of permutations generated is 1. Let us consider the effect of an I/O. There can be at most $N/B + t - 1$ full tracks before the tth output, and the records in the tth output can go into one of at most $N/B + t$ places relative to the full tracks. Hence, the tth output changes the number of permutations generated by at most a multiplicative factor of $N/B + t$, which can be bounded trivially by $N(1 + \log N)$.

For the case of input, we first consider an input of B records from a *specific track* on disk. If the B records were output together during some previous output, then by our assumptions this implies that at some earlier time they were together in internal memory and were arranged in an arbitrary order by the algorithm. Thus, the $B!$ possible orders of the B inputted records could already have been generated before the input took place. This implies in a subtle way that the increase in the number of permutations generated due to rearrangement in internal memory is at most a multiplicative factor of $\binom{M}{B}$, which is the number of ways to intersperse B indistinguishable items within a group of size M. If the B records were not output together previously, then the number of permutations generated is increased by an extra factor of $B!$, since the B records have not yet been permuted arbitrarily. It is important to note that this extra factor of $B!$ can appear only N/B times, namely once when the kth track is inputted for the first time, for each $1 \le k \le N/B$.

The above analysis applies to input from a specific track. If the input is the tth I/O, there are at most $N/B + t - 1$ tracks to choose from for the I/O, plus one more because input from an empty track is also possible. Putting our results together, we find that the number of permutations generated at time t can be a multiplicative factor of at most

$$\left(\frac{N}{B} + t\right)B!\binom{M}{B} \le N(1 + \log N)B!\binom{M}{B} \qquad (4.1)$$

times greater than the number of permutations generated at time $t - 1$, if the tth I/O is the input of the kth track for the first time, for some $1 \le k \le N/B$. Otherwise, the multiplicative factor is bounded by

$$\left(\frac{N}{B} + t\right)\binom{M}{B} \le N(1 + \log N)\binom{M}{B}. \qquad (4.2)$$

For the worst case, we get our lower bound by using (4.1) and (4.2) to determine the minimum value T such that the number of permutations generated is at least $N!$:

$$(B!)^{N/B}\left(N(1 + \log N)\binom{M}{B}\right)^T \ge N!. \qquad (4.3)$$

The $(B!)^{N/B}$ term appears because (4.1) contributes an extra $B!$ factor over (4.2), but this can happen at most N/B times. Taking logarithms and applying Stirling's formula to (4.3), with some algebraic manipulation, we get

$$T\left(\log N + B \log \frac{M}{B}\right) = \Omega\left(N \log \frac{N}{B}\right). \qquad (4.4)$$

If $B \log(M/B) \le \log N$, then it follows that $B \le \sqrt{N}$ and from (4.4) we get

$$T = \Omega\left(\frac{N \log(N/B)}{\log N}\right) = \Omega(N). \qquad (4.5)$$

On the other hand, if $\log N < B \log(M/B)$, then (4.4) gives us

$$T = \Omega\left(\frac{N \log(N/B)}{B \log(M/B)}\right). \qquad (4.6)$$

Combining (4.5) and (4.6), we get

$$T = \Omega\left(\min\left\{N, \frac{N \log(N/B)}{B \log(M/B)}\right\}\right). \qquad (4.7)$$

We get the worst-case lower bound in Theorem 3.2 by dividing (4.7) by P.

For the average case, in which the $N!$ permutations are equally likely, we can bound the average running time by the minimum value T such that

$$(B!)^{N/B}\left(N(1 + \log N)\binom{M}{B}\right)^{2T} \geq N!/2 \qquad (4.8)$$

(cf. (4.3)). At least half of the permutations require $2T$ I/Os; hence the average time to permute is at least $\frac{1}{2}\, 2T = T$. The lower bound for T follows by the same steps we used to handle (4.3). Note that this lower bound for T is roughly a factor of ½ times the bound on T in the worst case that follows from (4.3), but it is straightforward to derive (using (4.3) and a more careful estimate of the expected value) an average-case bound that is asymptotically the same as the worst-case bound.

Our proof technique also provides the constant factors implicit in the bounds in Theorem 3.2. If we assume that I/Os are simple and respect track boundaries, then the upper and lower bounds are asymptotically exact in many cases, as mentioned at the end of Section 3. If $B = o(M)$ and $P = 1$ and if $\log M/B$ either divides $\log N/B$ or else is $o(\log N/B)$, then the average number of I/Os for permuting (and sorting) is asymptotically at least $2N \log(N/B)/(B \log(M/B))$, which is matched by merge sort. The proof of the lower bound follows from the observation that there must be as many outputs as inputs, coupled with a more careful analysis of (4.3). For the last case quoted in Section 3, the matching lower bound follows by an analysis of matrix transposition, which we do later in this section.

FFT and Permutation Networks

A key observation for obtaining the lower bound for the FFT is that we can construct a permutation network by stacking together three FFT digraphs, so that the output nodes of one FFT are the input nodes for the next [10]. Thus the FFT and permutation network problems are essentially equivalent, since as we shall see the lower bound for permutation networks matches the upper bound for FFT.

Let us consider an optimal I/O strategy for a permutation network. The second key observation is that the I/O sequence is *fixed*. This allows us to apply the lower-bound proof developed above for permuting, with the helpful restriction that each I/O cannot depend upon the desired permutation; that is, regardless of the permutation, the records that are transferred during an I/O and the track accessed during the I/O are fixed for each I/O. This eliminates the $(N/B + t)$ terms in (4.1) and (4.2). Each output can at most double the number of permutations generated. The lower bound on the number of I/Os follows for $P = 1$ by finding the smallest T such that

$$(B!)^{N/B}\binom{M}{B}^{T} \geq N!. \qquad (4.9)$$

By using Stirling's formula, we get the same bound as in (4.6). Dividing by P gives the lower bound in Theorem 3.1.

It is interesting to note that since the I/O sequence is fixed and cannot depend upon the particular permutation, we are not permitted to use the naïve method of permuting, in which each block transfer moves one record from its initial to its final destination. This is reflected in the growth rate of the number of permutations generated due to a single I/O: the $(N/B + t)$ term in the growth rate in (4.1) and (4.2) for permuting, which is dominant when the naïve method is optimal, does not appear in the corresponding growth rate for permutation networks.

Sorting

Permuting is a special case of sorting, so the lower bound for permuting in Theorem 3.2 also applies to sorting. However, when $B \log(M/B) = o(\log(N/B))$, the lower bound becomes $\Omega(N/P)$, which is not good enough. In this case, the specific knowledge of what goes where makes generating a permutation easier than sorting.

We can get a better lower bound for sorting for the $B \log(M/B) = o(\log(N/B))$ case by using an adversary argument, if we restrict ourselves to the comparison model of computation. Without loss of generality, we can make the following additional assumptions, similar to the ones earlier: All I/Os are simple. Each I/O transfers B records, some possibly **nil**, to or from a single track on disk. We also assume that between I/Os the optimal algorithm performs all possible comparisons among the records in internal memory.

Let us consider an input of B records into internal memory. If the B records were previously outputted together during an earlier output, then by our assumptions all comparisons were performed among the B records when they were together in internal memory, and their relative ordering is known. The records in internal memory before the input, which number at most $M - B$, have also had all possible comparisons performed. Thus, after the input, there are at most $\binom{M}{B}$ sets of possible outcomes to the comparisons between the records in memory. If the B records were not previously outputted together (that is, if the input is the first input of the kth track, for some $1 \leq k \leq N/B$), then there are at most $B!\binom{M}{B}$ sets of possible outcomes to the comparisons. The adversary chooses the outcome that maximizes the number of total orders consistent with the comparisons done so far. It follows that (4.9) holds at time T, which yields the desired lower bound. Dividing by P gives the lower bound stated in Theorem 3.1.

The same result holds in the average-case model. We consider the comparison tree with $N!$ leaves, representing the $N!$ total orderings. Each node in the tree represents an input operation. The nodes are constrained to have degree bounded by $\binom{M}{B}$, except that each node corresponding to the input of one of tracks $1, \ldots, N/B$ can have degree at most $B!\binom{M}{B}$; there can be at most N/B such high-degree nodes along any path from the root to a leaf. The external path length divided by $N!$, minimized over all possible computation trees, gives the desired lower bound for $P = 1$. Dividing by P gives the lower bound of Theorem 3.1.

Matrix Transposition

We prove the lower bound using a potential function argument similar to the one used by Floyd [3]. It suffices to consider the case $P = 1$; the general lower bound will follow by dividing by P. Without loss of generality, we assume that p and q are powers of 2, and that all I/Os are simple and transfer exactly B records, some possibly **nil**.

We define the ith *target group*, for $1 \le i \le N/B$, to be the set of records that will ultimately be in the ith track at the termination of the algorithm. We define the continuous function

$$f(x) = \begin{cases} x \log x, & \text{if } x > 0; \\ 0, & \text{if } x = 0. \end{cases} \tag{4.10}$$

We assign a "togetherness rating"

$$C_k(t) = \sum_{1 \le i \le N/B} f(x_{i,k}) \tag{4.11}$$

to the kth track at time t if after t I/Os the kth track contains $x_{i,k}$ records belonging to the ith target group. Similarly, we assign a togetherness rating of

$$C_M(t) = \sum_{1 \le i \le N/B} f(y_i) \tag{4.12}$$

to the internal memory at time t, where y_i is the number of records belonging to the ith target group that are in internal memory after the tth I/O. We define the potential at time t to be the sum of the togetherness ratings

$$\mathrm{POT}(t) = C_M(t) + \sum_{k \ge 1} C_k(t). \tag{4.13}$$

We denote by T the total number of I/Os performed by the end of the algorithm. At time T each track has togetherness rating $C_k(T) = B \log B$, and the internal memory is empty; hence we have

$$\mathrm{POT}(T) = N \log B. \tag{4.14}$$

Now let us compute the initial potential POT(0). If $B < \min\{p, q\}$, no target group has two records that are initially in the same track. If $\min\{p, q\} \le B \le \max\{p, q\}$, each target group is partitioned into $\min\{p, q\}$ equal-sized groups, such that the records in the same group initially reside in the same track. If $B > \max\{p, q\}$, there are N/B groups. This gives

$$\mathrm{POT}(0) = \begin{cases} 0, & \text{if } B < \min\{p, q\}; \\ N \log \dfrac{B}{\min\{p, q\}}, & \text{if } \min\{p, q\} \le B \le \max\{p, q\}; \\ N \log \dfrac{B^2}{N}, & \text{f } \max\{p, q\} < B; \end{cases} \tag{4.15}$$

If a block is output from internal memory to disk at time t, then the potential function does not increase at that point; that is, $\mathrm{POT}(t) \le \mathrm{POT}(t - 1)$. Let us assume that the tth I/O is an input from the kth track of the disk to internal memory. After the input the togetherness rating $C_k(t)$ of the kth track is 0. The increase in potential $\nabla\mathrm{POT}(t)$ is thus

$$\nabla\mathrm{POT}(t) = C_M(t) - C_M(t - 1) - C_k(t - 1). \tag{4.16}$$

The contribution of a target group to the togetherness rating of internal memory increases when some of the records were present in internal memory before the input and some others were included in the input. We use y_i' and y_i'' to denote the number of records from the ith target group that are, respectively, present in internal memory at time $t - 1$ and input into internal memory from the kth track at time t. We have

$$\nabla\mathrm{POT}(t) = \sum_{1 \le i \le N/B} \left(f(y_i' + y_i'') - f(y_i') - f(y_i'')\right), \tag{4.17}$$

where

$$\sum_{1 \le i \le N/B} y_i' \le M - B \quad \text{and} \quad \sum_{1 \le i \le N/B} y_i'' \le B. \tag{4.18}$$

A simple convexity argument shows that (4.17) is maximized when $y_i' = (M - B)B/N$ and $y_i'' = B^2/N$, for each $1 \le i \le N/B$. For $0 \le y \le x \le 1$, we have

$$\begin{aligned} f(x + y) - f(x) - f(y) &= x \log\left(1 + \frac{y}{x}\right) + y \log\left(1 + \frac{x}{y}\right) \\ &\le \frac{y}{\ln 2} + y \log\left(1 + \frac{x}{y}\right) \\ &= O\left(y \log\left(1 + \frac{x}{y}\right)\right). \end{aligned} \tag{4.19}$$

Substituting $x = (M - B)B/N$ and $y = B^2/N$ into (4.19), it follows from (4.17) that

$$\nabla\mathrm{POT}(t) = O\left(B \log \frac{M}{B}\right). \tag{4.20}$$

At the end of the algorithm, there are at least $T \ge N/B$ outputs, and thus by (4.20)

$$T - \frac{N}{B} = \Omega\left(\frac{\mathrm{POT}(T) - \mathrm{POT}(0)}{B \log(M/B)}\right). \tag{4.21}$$

The lower bound

$$T=\Omega\left(\frac{N}{B}\frac{\log\min\{M,1+\min\{p,q\},1+N/B\}}{\log(1+M/B)}\right) \quad (4.22)$$

follows by substituting (4.14) and the different cases of (4.15) into (4.21). The general lower bound in Theorem 3.3 for $P > 1$ follows by dividing (4.22) by P.

The constant factor implicit in the above analysis matches the constant factor 2 for merge sort in several cases, if we require that all I/Os be simple and respect track boundaries, as defined at the end of Section 3. When $P = 1$ and $B = \Omega(\sqrt{N}/\log^b N)$, for some b, and when $\log M/B$ either divides $\log N/B$ or else is $o(\log N/B)$, the lower bound for transposing a $B \times N/B$ matrix is asymptotically at least $2N \log(N/B)/(B \log(M/B))$, which matches the performance of merge sort. The proof follows from the above analysis (which gives a lower bound on the number of inputs required) and the observation that there must be as many outputs as inputs. If, in addition, we substitute the bound $f(x + y) - f(x) - f(y) \le x + y$, for positive integers x and y, in place of (4.19), we get the same asymptotic lower bound formula for the case $M = \sqrt{N}$, $B = \frac{1}{2}M$, $P = 1$, which matches merge sort.

5. OPTIMAL ALGORITHMS

In this section, we describe variants of merge sort and distribution sort that achieve the bounds in Theorems 3.1–3.3. As mentioned in Section 3, the algorithms follow the added restriction that records input in the same block must have been output previously in a single block, except for the first input of each track. It suffices to consider worst-case complexity, since the average-case result follows immediately. We first discuss the sorting problem and then apply our results to get optimum algorithms for permuting, FFT, permutation networks, and matrix transposition. Without loss of generality, we can assume that B, M, and N are powers of 2 and that $B < M < N$.

Merge Sort

The standard merge sort algorithm works as follows: In the "run" formation phase, the N/B tracks are inputted into memory, in groups of one memoryload at a time; each memoryload is sorted into a "run," which is then output to consecutive positions on disk. At the end of the run formation phase, there are N/M runs on disk. (In actual implementations, the "replacement-selection" technique [5] can be used to get runs of $2M$ records, on the average, when $M \gg B$.) In each pass of the merging phase, M/B^{-1} runs are merged into one longer run. During the processing, one block from each of the runs being merged resides in internal memory. When the records of a block expire, the next track for that run is input. The resulting number of I/Os is

$$\frac{2N}{B}\left(1+\left\lceil\log_{M/B^{-1}}\frac{N}{M}\right\rceil\right)=O\left(\frac{N\ \log(N/B)}{B\ \log(M/B)}\right). \quad (5.1)$$

This does not yield an optimal algorithm, however, when P is not bounded by a constant, since there is no way of knowing which P tracks should be inputted next. The solution is to modify the information that goes into each track. Besides the records themselves, we also place into each track $P - 1$ "endmarkers," which are the key values of the last record in the each of the next $P - 1$ tracks of the run. Using a generalization of the forecasting technique described in [5], we can then determine the P tracks that will expire next. Note, however, that several of these tracks might not yet be present in internal memory. Merging proceeds until a track not currently in memory is needed. An input can then be performed to transfer the next P tracks needed, using the forecasting information, and the process continues.

First we consider the case $P \le B/2$. In each pass, the endmarkers are not output at the same time that the track is output, since they are not yet determined at that time. Instead, when we output the records of the lth output track, we also output the endmarkers for the $(l - P)$th output track. To do that, we have to store in internal memory the addresses and the largest key values of the last $P - 1$ tracks. This consumes $O(B)$ space, under our assumption that $P \le B/2$, so the number of tracks and the number of I/Os needed to store a run of a given length do not change by more than a constant factor. The number of passes in the merging phase also does not change by more than a constant factor. The resulting speedup is $\Theta(P)$, as desired.

However, if $P > B/2$, then there may not be enough room to store the endmarkers without increasing the number of tracks per run by too large an amount. In this case, we form "metatracks" of size $B' = \lceil\sqrt{2M}\rceil \ge B$. The number of metatracks that can be input concurrently is $P' = \lfloor PB/(\lceil B'/B\rceil B)\rfloor$, which is bounded by $M/B' \le B'/2$. This satisfies the requirement for the construction in the previous paragraph, using P' and B' in place of P and B. The result is that the number of I/Os is reduced by $\Theta(P')$ from the number used by standard merge sort. By (5.1), the number of I/Os performed by the standard merge sort would be

$$O\left(\frac{N\ \log(N/B')}{B'\log(M/B')}\right). \quad (5.2)$$

Dividing (5.2) by $P' = PB/B'$ and with some algebraic manipulation, we get the desired upper bound stated in Theorem 3.1.

Distribution Sort

For simplicity, we assume that M/B is a perfect square, and we use S to denote the quantity $\sqrt{M/B}$. The main idea in the algorithm is that with $O(N/(PB))$ I/Os we can find S *approximate partitioning elements* $b_1, b_2, \ldots, b_S$ that break up the file into roughly equal-sized "buckets." (For completeness, we define the dummy partitioning elements $b_0 = -\infty$ and $b_{S+1} = +\infty$). More precisely, we shall prove later, for $1 \le i \le S + 1$, that the number of records whose key value is $\le b_i$ is between

$(i - 1/4)N/S$ and $(i + 1/4)N/S$. Hence, the number N_i of records in the ith bucket (that is, the number N_i of records whose key value K is in the range $b_{i-1} < K \le b_i$) satisfies

$$\frac{1}{2}\frac{N}{S} \le N_i \le \frac{3}{2}\frac{N}{S}. \tag{5.3}$$

For the time being, we assume that we can compute the approximate partitioning elements using $O(N/(PB))$ I/Os. Then with $O(M/(PB))$ additional I/Os we can input M records from disk into internal memory and partition them into the S bucket ranges. The records in each bucket range can be stored on disk in contiguous groups of B records each (except possibly for the last group) with a total of $O(M/(PB) + S/P) = O(M/(PB))$ I/Os. This procedure is repeated for another $N/M - 1$ stages, in order to partition all N records into buckets. The ith bucket will thus consist of $G_i \le N_i/B + N/M = O(N_i/B)$ groups of at most B contiguous records, by using inequality (5.3). The buckets are totally ordered with respect to one another. The remainder of the algorithm consists of recursively sorting the buckets one-by-one and appending the results to disk. The number of I/Os needed to input the contents of the ith bucket into internal memory during the recursive sorting is bounded by $G_i/P = O(N_i/(PB))$. Let us define $T(n)$ to be the number of I/Os used to sort n records. The above construction gives us

$$T(N) = \sum_{1 \le i \le S+1} T(N_i) + O\left(\frac{N}{PB}\right). \tag{5.4}$$

Using the facts that $N_i = O(N/S) = O(N/\sqrt{M/B})$ and $T(M) = O(M/(PB))$, we get the desired upper bound given in Theorem 3.1.

All that remains to show is how to get the S approximate partitioning elements via $O(N/(PB))$ I/Os. 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 $O(N/B)$ groups 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 kth smallest key (or simply the kth smallest record) in a set containing n records, in which the records are stored on disk in at most $O(n/B)$ groups, each group consisting of $\le B$ contiguous records: We load the n records into memory, one memoryload at a time, and sort each of the $\lceil n/M \rceil$ memoryloads internally. We pick the median record from each of these sorted sets and find the median of the medians using the linear-time sequential algorithm developed in [2]. The number of I/Os required for these operations is $O(n/(PB) + (n/B)/P + n/M) = O(n/(PB))$. 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 groups of size B (except possibly for the last group) in which each group is stored contiguously on disk. It is also easy to see that each of the two sets has size bounded by $3n/4$. The algorithm is recursively applied to the appropriate half to find the kth largest 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. 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 jth sorted set by U_j. We construct a new set U' of size at most $4N/S$ consisting of the $1/4kS$th records (in sorted order) of U_j, for $1 \le k \le 4M/S - 1$ and $1 \le j \le N/M$. Each memoryload of M records contributes $4M/S > B$ records to U', so these records can be output one block at a time. The total number of contiguous groups of records comprising U' is $O(|U'|/B)$, so we can apply the subroutine above to find the record of rank $4iN/S^2$ in U' with only $O(|U'|/(PB)) = O(N/(SPB))$ I/Os; we call its key value b_i. The S b_i's can thus be found with a total of $O(N/(PB))$ I/Os. It is easy to show that the b_i's satisfy the conditions for being approximate partitioning elements, thus completing the proof.

Permuting

The permuting problem is a special case of the sorting problem, and thus can be solved by using a sorting algorithm. To get the upper bound of Theorem 3.2, we use either of the sorting algorithms described above, unless $B \log(M/B) = o(\log(N/B))$, in which case it is faster to move the records one-by-one in the naïve manner to their final positions, using $O(N/P)$ I/Os.

FFT and Permutation Networks

As mentioned in Section 4, three FFT digraphs concatenated together form a permutation network. So it suffices to consider optimum algorithms for FFT.

For simplicity, we assume that $\log M$ divides $\log N$. The FFT digraph can be decomposed into $(\log N)/\log M$ stages, as pictured in Figure 2. Stage k, for $1 \le k \le (\log N)/\log M$, corresponds to the pebbling of columns $(k - 1)\log M + 1$, $(k - 1)\log M + 2, \ldots, k \log M$ in the FFT digraph. The M nodes in column $(k - 1)\log M$ that share common ancestors in column $k \log M$ are processed together in a phase. The corresponding M records are brought into internal memory via a transposition permutation, and then the next $\log M$ columns can be pebbled.

The I/O requirement for each stage is thus due to the transpositions needed to rearrange the records into the proper groups of size M. The transpositions can be collectively done via a simple merging procedure described in the next subsection, which requires a total of $O((N/(PB))\log_{M/B}\min\{M, N/M\})$ I/Os. There are $(\log N)/\log M$ stages, making the total number of I/Os

$$O\left(\frac{N \log N}{PB \log M}\left(\frac{\log \min\{M, N/B\}}{\log(M/B)}\right)\right), \tag{5.5}$$

which can be shown by some algebraic manipulation to equal the upper bound of Theorem 3.1.

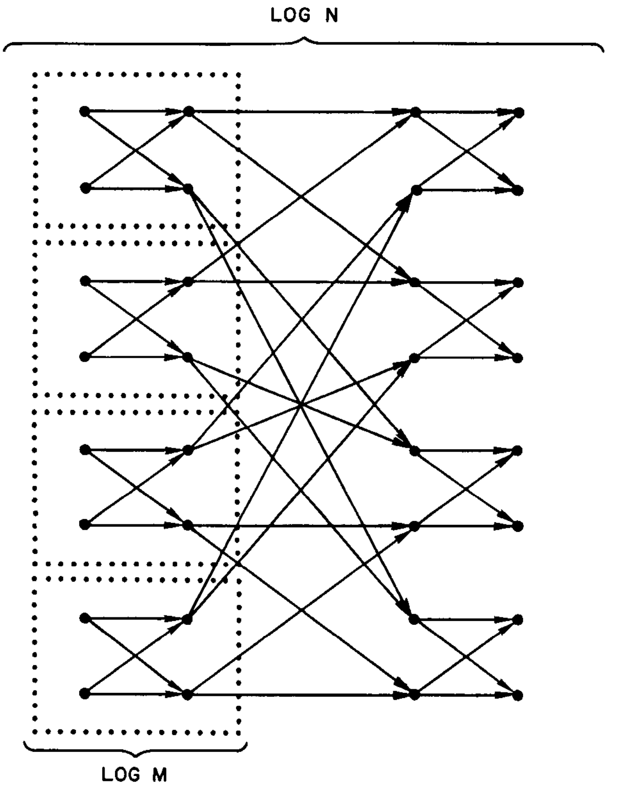

FIGURE 2. Decomposition of the FFT digraph into stages, for $N = 8, M = 2$

Matrix Transposition

Without loss of generality, we assume that p and q are powers of 2. Matrix transposition is a special case of permuting. The intuition gained from the lower bound proof in Section 4 can be used to develop a simple algorithm for achieving the upper bound in Theorem 3.3. In each track, the B records are partitioned into different target groups; each group in the decomposition is called a *target subgroup*. Before the start of the algorithm, the size of each target subgroup is (cf. (4.15))

$$x = \begin{cases} 1, & \text{if } B < \min\{p, q\}; \\ \dfrac{B}{\min\{p, q\}}, & \text{if } \min\{p, q\} \le B \le \max\{p, q\}; \\ \dfrac{B^2}{N}, & \text{if } \max\{p, q\} < B. \end{cases} \qquad (5.6)$$

The algorithm uses a merging procedure. The records in the same target subgroup remain together throughout the course of the algorithm. In each pass, target subgroups are mergeĉ and become bigger. The algorithm terminates when each target subgroup is complete, that is, when each target subgroup has size B. In each pass, which takes $2N/PB$ I/Os, the size of each target subgroup increases by a multiplicative factor of M/B. The number of passes made by the algorithm is thus

$$\left\lceil \log_{M/B} \frac{B}{x} \right\rceil. \qquad (5.7)$$

We get the upper bound in Theorem 3.3 by substituting the values of x from (5.6) into (5.7) and by multiplying by $2N/PB$, the number of I/Os per pass.

6. ALTERNATE PROOF OF HONG AND KUNG'S RESULT

In this section, we give a simple proof that the FFT requires $\Omega(N(\log N)/\log M)$ I/Os for the special case $B = P = O(1)$, which was proved in [4] using a complicated pebbling argument.

Our model for this special case can be phrased in terms of the red-blue pebble game, introduced in [4]. There are M red pebbles, representing internal memory storage, and an unlimited supply of blue pebbles, representing information stored on disk. The FFT digraph must be pebbled using the standard pebbling rules applied to the red pebbles, except that the following special I/O operations are allowed: A blue pebble may be placed on any node containing a red pebble, and a red pebble may be placed on any node containing a blue pebble, each at the cost of one I/O. The "cost" of the red-blue pebbling game is the number of I/Os performed; the red pebbling moves are free.

Our simplified proof of Hong and Kung's result rests on the following intuitive lemma:

LEMMA 6.1. *Given any initial configuration of M red pebbles on the FFT digraph, at most $2M \log M$ red pebbling moves can be made without I/O.*

PROOF. To bound the number of red pebbling moves, we use a dynamic charging strategy to allocate the moves to individual red pebbles. Let $\text{num}(p)$ denote the number of moves currently allocated to pebble p. A generic red pebbling move in the FFT digraph has the following form: Two pebbles p_1 and p_2 rest on nodes l_1 and l_2, and they share common parents u_1 and u_2. Both p_1 and p_2 are then moved to the upper level nodes u_1 and u_2. (Keeping one of the pebbles behind might only reduce the number of possible red pebbling moves, which we are trying to maximize.) Our charging strategy is to charge 1 to p_1 if $\text{num}(p_1) \le \text{num}(p_2)$, and 1 to p_2 if $\text{num}(p_2) \le \text{num}(p_1)$. The total number of red pebbling moves is therefore bounded by

$$2 \sum_{\text{pebbles } p} \text{num}(p). \qquad (6.1)$$

The lemma below can be proved easily by induction; the proof is therefore omitted.

LEMMA 6.2. *For each pebble p on node n in the FFT digraph, the number of nodes that contained a red pebble in the initial configuration and that are connected by a directed path to n is at least $2^{\text{num}(p)}$.*

There are M red pebbles, so each pebble p can "cover" at most M original placements. By Lemma 6.2, we have $\text{num}(p) \le \log M$. Plugging this into (6.1) completes the proof of Lemma 6.1.

Each node in the FFT digraph must be red pebbled at least once. Since there are $N \log N$ nodes, Lemma 6.1 implies that the number of I/Os required for the $P = M$, $B = 1$ case is at least

$$\frac{N \log N}{2M \log M}. \qquad (6.2)$$

For the case $P = B = 1$, which is what we want to consider, we appeal to the following lemma.

LEMMA 6.3. *Any fixed I/O schedule can be simulated by consecutive groups of I/O operations, in which each group consists either of M inputs or M outputs, and the total number of I/Os does not increase by more than a constant factor.*

PROOF. The lemma follows from the fact that the I/O schedule for the FFT is *fixed* and does not depend on the input; thus, caching can be done in order to group the I/Os in the desired fashion.

If we treat each group of inputs and each group of outputs as a single operation, we find ourselves in the case $P = M$; a lower bound on the number of groups is given by (6.2). In terms of the $P = 1$ model, each group represents M I/Os, and our lower bound follows by multiplying (6.2) by M.

7. CONCLUSIONS

We have derived matching upper and lower bounds, up to a constant factor, for the average-case and worst-case number of I/Os needed to perform sorting-related tasks, which include sorting, FFT, permutation networks, permuting, and matrix transposition. Under mild restrictions on the types of I/O possible, the constant factors implicit in our upper and lower bounds are often equal. Our bounds also apply if the disk has a special capability to access up to S groups of contiguous regions on disk in a single I/O. This situation corresponds to a disk without the special capability that has block size $B' = B/S$ and degree of parallelization $P' = PS$.

The optimal upper bounds for $B = 1$ when $M = N^{\Omega(1)}$, can be obtained via a recursive application of Columnsort [7]; however, for smaller M the upper bound is greater than optimal by a factor of roughly $\log \log N$.

Recently, Beigel and Gill [1] have independently considered the problem of determining how many applications of a black box capable of sorting k records are necessary to sort N records. Their problem corresponds to the sorting problem for the special case $P = M = k$ and $B = 1$. They have shown that $\Theta((N \log N)/(k \log k))$ I/Os are optimal in that case (cf. Theorem 3.1). In addition, they have derived bounds on the constant factors involved in their version of the problem.

Kwan and Baer [6] study an alternative disk model, in which $P = 1$ and the disk is decomposed into contiguous cylinders, each composed of several tracks. (The track size is a hardware parameter, and can be different from the logical block size used for data transfer, unlike our use of the term in Definition 3.2.) The tracks all revolve at a constant rate. There is one read/write head per track, and the set of heads can move in unison from cylinder to cylinder. Seek time in an I/O is proportional to the number of cylinders traversed by the heads, and rotational latency time is proportional to the radial distance between the head positions at the start of an I/O request and the head positions at the beginning of the actual data transfer. An algorithm for permuting records that takes advantage of locality of reference on the disk is given; it achieves better running times than merge sort in this model when the file size is large.

We believe, however, that the simpler model we use in this paper gives more meaningful results. Kwan and Baer's model [6], in comparison with current technology, is overly pessimistic in how it models a random seek. For example, for the large-capacity magnetic disks made by IBM, the time to do a seek between adjacent cylinders is of the same order of magnitude as the time for a random seek or for a complete revolution. In this more realistic context, the permutation algorithm in [6] is slower than merge sort. In addition, the I/O block size in large external sorts is often on the order of the disk track size. Thus, the time for the data transmission during an I/O is as large in magnitude as the seek and latency times, which justifies the simpler model we study in this paper.

We conclude this paper with a challenging open problem: Can we remove our assumption that the records are indivisible and allow, for example, arbitrary bit manipulations and dissections of the records? Intuitively, the lower bound should still hold in this more general model, since it is unlikely that these operations are of any great help, but no proof is known. Such a proof would no doubt provide great insight into the nature of information transfer and sorting-related computations.

REFERENCES

1. Beigel, R., and Gill, J. *Personal Communication.* 1986.
2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E. Time bounds for selection. *J. Comput. Syst. Sci. 7* (1973), 448–461.
3. Floyd, R.W. Permuting information in idealized two-level storage. In *Complexity of Computer Calculations*, R. Miller and J. Thatcher, Eds. Plenum, New York, 1972, pp. 105–109.
4. Hong, J.W., and Kung, H.T. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, Oct.). pp. 326–333. 1981.
5. Knuth, D.E. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
6. Kwan, S.C., and Baer, J.L. The I/O performance of multiway mergesort and tag sort. *IEEE Trans. Comput. C-34*, 4 (Apr. 1985), 383–387.
7. Leighton, F.T. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput. C-34*, 4 (Apr. 1985).
8. Lindstrom, E.E., and Vitter, J.S. The design and analysis of BucketSort for bubble memory secondary storage. *IEEE Trans. Comput. C-34*, 3 (Mar. 1985), 218–233.
9. Savage, J.E., and Vitter, J.S. Parallelism in space-time tradeoffs. In *Advances in Computing Research, Volume 4: Special Issue on Parallel and Distributed Computing.* JAI Press, Greenwich, Conn., 1987, pp. 117–146.
10. Wu, C.L., and Feng, T.Y. The universality of the shuffle-exchange network. *IEEE Trans. Comput. C-30*, 5 (May 1981), 324–332.

An extended abstract of this work appeared in the I/O complexity of sorting and related problems. In *Proceedings of the 14th Annual Colloquium on Automata, Languages, and Programming.* (Karlsruhe, West Germany, July), 1987. Support for Jeff Vitter was provided in part by NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, and by a Guggenheim Fellowship. Research was also done while the authors were at the Mathematical Sciences Research Institute in Berkeley, CA and while Jeff Vitter was on sabbatical at INRIA in Rocquencourt, France, and Ecole Normale Supérieure in Paris, France.

CR Categories and Subject Descriptors: D.4.2 [**Operating systems**]: Storage Management—*main memory, secondary storage devices*; D.4.4 [**Operating Systems**]: Communications Management—*input/output*; E.5 [**Data**]: Files—*sorting and searching*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms, permutations and combinations.*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Distribution sort, fast Fourier transform, input, magnetic disk, merge sort, networks, output, pebbling, secondary storage, sorting

Received 8/87; revised 2/88; accepted 3/88

ABOUT THE AUTHORS

ALOK AGGARWAL is a research staff member at the IBM Watson Research Center. He received his Ph.D. in Electrical Engineering/Computer Science from Johns Hopkins University in 1984. His research interests include computational geometry, parallel algorithms, and VLSI theory.

JEFFREY VITTER is a professor of computer science at Brown University. He is a Guggenheim Fellow and a recipient of an NSF Presidential Young Investigator Award. He is coauthor of *Design and Analysis of Coalesced Hashing* (1987). His research interests include analysis of algorithms, computational complexity, parallel computing, and machine learning. Authors' present addresses: Alok Aggarwal, IBM Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; Jeffrey S. Vitter, Department of Computer Science, Brown University, Providence, RI 02912.