Paradigms for Efficient Design of External Memory Algorithms

Jeffrey Scott Vitter[1,2*]

Purdue University, Department of Computer Sciences, West Lafayette, IN 47907-2066, USA
http://www.science.purdue.edu/jsv/
jsv@purdue.edu

Abstract. Data sets in large applications are often too massive to fit completely inside the computer's internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. We survey the state of the art in the design and analysis of *external memory* (or *EM*) *algorithms*, where the primary goal is to reduce the number of input/output (or I/O) operations, which tend to be a bottleneck in data-intensive applications. Two important avenues to reduce I/O costs are to exploit locality in EM algorithm design, in order to increase the amount of usefulinformation transferred in each I/O, and to take advantage of parallel disk drives, which can be accessed simultaneously. We examine several useful design paradigms and fundamental performance bounds for I/O performance.

1 Introduction

1.1 Background

For reasons of economy, general-purpose computer systems usually contain a hierarchy of memory levels, each level with its own cost and performance characteristics. At the lowest level, CPU registers and caches are built with the fastest but most expensive memory. For internal main memory, dynamic random access memory (DRAM) is typical. At a higher level, inexpensive but slower magnetic disks are used for external mass storage, and even slower but larger-capacity devices such as tapes and optical disks are used for archival storage. Figure 1 depicts a typical memory hierarchy and its characteristics.

Most modern programming languages are based upon a programming model in which memory consists of one uniform address space. The notion of virtual memory allows the address space to be far larger than what can fit in the internal memory of the computer. Programmers have a natural tendency to assume that all memory references require the same access time. In many cases, such an assumption is reasonable (or at least doesn't do any harm), especially when the data sets are not large. The utility

* Supported in part by the Army Research Office through grant DAAD19–01–1–0725, and by the National Science Foundation through research grants CCR–9877133 and EIA–9870734. Earlier versions of some of this material appeared in [141, 140, 142–144, 146, 145]. Part of this work was done at Duke University and INRIA in Sophia-Antipolis, France.

and elegance of this programming model are to a large extent why it has flourished, contributing to the productivity of the software industry.

However, not all memory references are created equal. Large address spaces span multiple levels the of memory hierarchy, and accessing the data in the lowest levels of memory is orders of magnitude faster than accessing the data at the higher levels. For example, loading a register takes on the order of a nanosecond (10^{-9} seconds), and accessing internal memory takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds (10^{-3} seconds), which is about one million times slower! In applications that process massive amounts of data, the *Input/Output* communication (or simply *I/O*) between levels of memory is often the bottleneck.

Many computer programs exhibit some degree of *locality* in their pattern of memory references: Certain data are referenced repeatedly for a while, and then the program shifts attention to other sets of data. Modern operating systems take advantage of such access patterns by tracking the program's so-called "working set"—a vague notion that roughly corresponds to the recently referenced data items [49]. If the working set is small, it can be cached in high-speed memory so that access to it is fast. Caching and prefetching heuristics have been developed to reduce the number of occurrences of a "fault", in which the referenced data item is not in the cache and must be retrieved by an I/O from a higher level of memory. For example, in a page fault, an I/O is needed to retrieve a disk page from disk and bring it into internal memory.

Caching and prefetching methods are typically designed to be general-purpose, and thus they cannot be expected to take full advantage of the locality present in every computation. Some computations themselves are inherently nonlocal, and even with

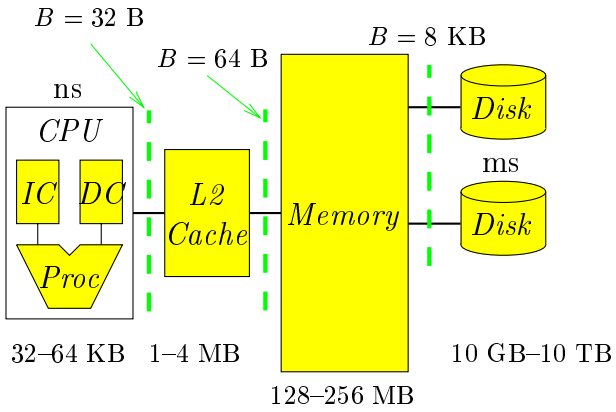

Fig. 1. The memory hierarchy of a typical uniprocessor system, including registers, instruction cache, data cache (level 1 cache), level 2 cache, internal memory, and disks. Below each memory level is the range of typical sizes for that memory level. Each value of B at the top of the figure denotes the block transfer size between two adjacent levels of the hierarchy. All sizes are given in units of bytes (B), kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB). (In the PDM model described in in Section 2, we measure B in units of items rather than in units of bytes.) In this figure, 8 KB is the indicated physical block transfer size between internal memory and the disks. However, in batched applications it is often more appropriate to use a substantially larger logical block transfer size.

omniscient cache management decisions they are doomed to perform large amounts of I/O and suffer poor performance. Substantial gains in performance may be possible by incorporating locality *directly* into the algorithm design and by explicit management of the contents of each level of the memory hierarchy, thereby bypassing the virtual memory system.

We refer to algorithms that explicitly manage data placement and movement as *external memory* (or *EM*) *algorithms*. Some authors use the terms *I/O algorithms* or *out-of-core algorithms*. We concentrate in this survey on the I/O communication between the random access internal memory and the magnetic disk external memory, where the relative difference in access speeds is most apparent. We therefore use the term I/O to designate the communication between the internal memory and the disks.

1.2 Overview

In this article we survey several paradigms for exploiting locality and thereby reducing I/O costs when solving problems in external memory. (See Table 1.) We give the fundamental performance bounds required to solve various problems of interest. We concentrate in this article on *batched problems*, in which no preprocessing is done and the entire file of data items must be processed, often by streaming the data through the internal memory in one or more passes. Typical examples of batched problems include sorting, FFT, computing Voronoi diagrams, and computing intersecting segments. We refer the reader to Chapter ??? for a discussion of *online problems*, such as the dictionary and priority queue problems, in which computation is done in response to a continuous series of query operations.

In the next section we describe the *parallel disk model* (PDM) that we use as the basis for our algorithm design. The three main performance measures of PDM are *the number of I/O operations, the disk space usage,* and *the CPU time*. We focus in this article on the number of I/O operations. In Section 3 we list the fundamental I/O bounds that pertain to most of the problems we consider. In Section 4 we show why it is crucial for EM algorithms to exploit locality, and we discuss an automatic load balancing technique called disk striping for using multiple disks in parallel.

In Section 5 we focus on parallel disk algorithms for the canonical batched EM problem of external sorting and the related problems of permuting and fast Fourier transform. In Section 6, we discuss grid and linear algebra batched computations.

For most problems, parallel disks can be utilized effectively by means of disk striping or the parallel disk techniques of Section 5, and hence we restrict ourselves starting in Section 7 to the conceptually simpler single-disk case. In Section 7 we mention several effective paradigms for batched EM problems in computational geometry. In Section 8 we look at EM algorithms for combinatorial problems on graphs. Algorithms for strings are discussed in Chapter ??? and [144]. Experiments and programming environments are discussed in Section 9.

In Section 10, we examine some fundamental lower bounds on I/O performance for the problems examined earlier. In Section 11 we discuss upper and lower bounds for EM algorithms that adapt optimally to dynamically changing internal memory allocations. We conclude with some final remarks and observations in Section 12.

2 Parallel Disk Model (PDM)

EM algorithms explicitly control data placement and movement, and thus it is important for algorithm designers to have a simple but reasonably accurate model of the

Paradigm	Reference
Batched filtering	§7
Batched incremental construction	§7
Disk striping	§4.2
Distribution	§5.1
Distribution Sweeping	§7
Fractional Cascading	§7
Load Balancing	§4
Locality	§4
Marriage before conquest	§7
Merging	§5.2
Parallel simulation	§8
Persistence	§7
Random sampling	§5.1
Scanning (or streaming)	§2.2
Sparsification	§8

Table 1. Paradigms for I/O efficiency discussed in this survey.

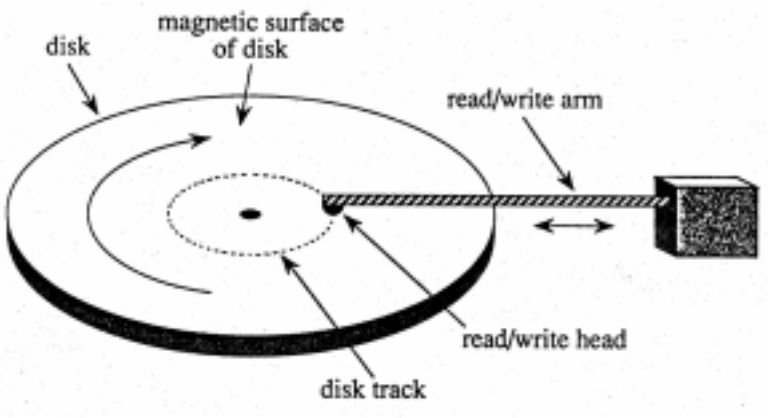

Fig. 2. Platter of a magnetic disk drive.

memory system's characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data are stored on the platters in concentric circles called *tracks*, as shown in Figure 2. To read or write a data item at a certain address on disk, the read/write head must mechanically *seek* to the correct track and then wait for the desired address to pass by. The seek time to move from one random track to another is often on the order of 5 to 10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it pays to transfer a large contiguous group of data items, called a *block*. Similar considerations apply to all levels of the memory hierarchy. Typical block sizes are shown in Figure 1.

Even if an application can structure its pattern of memory accesses to exploit locality and take full advantage of disk block transfer, there is still a substantial *access*

gap between internal and external memory performance. In fact the access gap is growing, since the latency and bandwidth of memory chips are improving more quickly than those of disks. Use of parallel processors further widens the gap. Storage systems such as RAID deploy multiple disks in order to get additional bandwidth [31, 71].

In the next section we describe the high-level parallel disk model (PDM), which we use throughout this survey for the design and analysis of EM algorithms. In Section 2.2 we consider some practical modeling issues dealing with the sizes of blocks and tracks and the corresponding parameter values in PDM. In Section 2.3 we review the historical development of models of I/O and hierarchical memory.

2.1 PDM and Problem Parameters

We can capture the main properties of magnetic disks and multiple disk systems by the commonly used *parallel disk model* (PDM) introduced by Vitter and Shriver [150]:

$$\begin{aligned} N &= \text{problem size (in units of data items);} \\ M &= \text{internal memory size (in units of data items);} \\ B &= \text{block transfer size (in units of data items);} \\ D &= \text{number of independent disk drives;} \\ P &= \text{number of CPUs,} \end{aligned}$$

where $M < N$, and $1 \leq DB \leq M/2$. The data items are assumed to be of fixed length. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items.

If $P \leq D$, each of the P processors can drive about D/P disks; if $D < P$, each disk is shared by about P/D processors. The internal memory size is M/P per processor, and the P processors are connected by an interconnection network. For routing considerations, one desired property for the network is the capability to sort the M data items in the collective main memories of the processors in parallel in optimal $O\big((M/P)\log M\big)$ time.[1] The special cases of PDM for the case of a single processor ($P = 1$) and multiprocessors with one disk per processor ($P = D$) are pictured in Figure 3.

Queries are naturally associated with online computations, but they can also be done in batched mode. For example, in the batched orthogonal 2-D range searching problem discussed in Section 7, we are given a set of N points in the plane and a set of Q queries in the form of rectangles, and the problem is to report the points lying in each of the Q query rectangles. In both the batched and online settings, the number of items reported in response to each query may vary. We thus need to define two more performance parameters:

$$\begin{aligned} Q &= \text{number of input queries (for a batched problem);} \\ Z &= \text{query output size (in units of data items).} \end{aligned}$$

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the resulting formulas are often simplified. We define the lowercase notation

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad q = \frac{Q}{B}, \qquad z = \frac{Z}{B} \tag{1}$$

[1] We use the notation $\log n$ to denote the binary (base 2) logarithm $\log_2 n$. For bases other than 2, the base is specified explicitly.

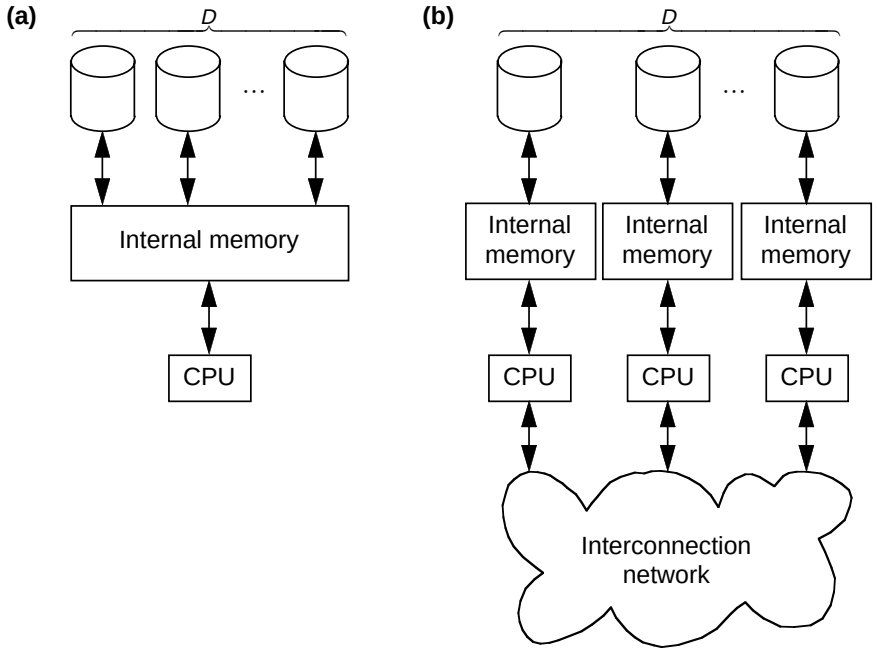

Fig. 3. Parallel disk model: (a) $P = 1$, in which the D disks are connected to a common CPU; (b) $P = D$, in which each of the D disks is connected to a separate processor.

to be the problem input size, internal memory size, query specification size, and query output size, respectively, in units of disk blocks.

We assume that the input data are initially "striped" across the D disks, in units of blocks, as illustrated in Figure 4, and we require the output data to be similarly striped. Striped format allows a file of N data items to be read or written in $O(N/DB) = O(n/D)$ I/Os, which is optimal.

The primary measures of performance in PDM are

1. the number of I/O operations performed,
2. the amount of disk space used, and
3. the internal (sequential or parallel) computation time.

For reasons of brevity in this survey we focus on only the first two measures. Most of the algorithms we mention run in optimal CPU time, at least for the single-processor case. Ideally algorithms should use linear space, which means $O(N/B) = O(n)$ disk blocks of storage.

2.2 Practical Modeling Considerations

Track size is a fixed parameter of the disk hardware; for most disks it is in the range 50–200 KB. In reality, the track size for any given disk depends upon the radius of the track (cf. Figure 2). Sets of adjacent tracks are usually formatted to have the same

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
stripe 0	0 1	2 3	4 5	6 7	8 9
stripe 1	10 11	12 13	14 15	16 17	18 19
stripe 2	20 21	22 23	24 25	26 27	28 29
stripe 3	30 31	32 33	34 35	36 37	38 39

Fig. 4. Initial data layout on the disks, for $D = 5$ disks and block size $B = 2$. The input data items are initially striped block-by-block across the disks. For example, data items 16 and 17 are stored in the second block (i.e., in stripe 1) of disk $\mathcal{D}_3$.

track size, so there are typically only a small number of different track sizes for a given disk. A single disk can have a 3 : 2 variation in track size (and therefore bandwidth) between its outer tracks and the inner tracks.

The minimum block transfer size imposed by hardware is often 512 bytes, but operating systems generally use a larger block size, such as 8 KB, as in Figure 1. It is possible (and preferable in batched applications) to use logical blocks of larger size (sometimes called clusters) and further reduce the relative significance of seek and rotational latency, but the wall clock time per I/O will increase accordingly. For example, if we set PDM parameter B to be five times larger than the track size, so that each logical block corresponds to five contiguous tracks, the time per I/O will correspond to five revolutions of the disk plus the (now relatively less significant) seek time and rotational latency. If the disk is smart enough, rotational latency can even be avoided altogether, since the block spans entire tracks and reading can begin as soon as the read head reaches the desired track. Once the block transfer size becomes larger than the track size, the wall clock time per I/O grows linearly with the block size.

For best results in batched applications, especially when the data are streamed sequentially through internal memory, the block transfer size B in PDM should be considered to be a fixed hardware parameter a little larger than the track size (say, on the order of 100 KB for most disks), and the time per I/O should be adjusted accordingly. For online applications that use pointer-based indexes, a smaller B value such as 8 KB is appropriate, as in Figure 1. The particular block size that optimizes performance may vary somewhat from application to application.

PDM is a good generic programming model that facilitates elegant design of I/O-efficient algorithms, especially when used in conjunction with the programming tools discussed in Section 9. More complex and precise disk models, such as the ones by Ruemmler and Wilkes [110], Ganger [63], Shriver et al. [123], Barve et al. [25], and Farach et al. [56], distinguish between sequential reads and random reads and consider the effects of features such as disk buffer caches and shared buses, which can reduce the time per I/O by eliminating or hiding the seek time. For example, algorithms for spatial join that access preexisting index structures (and thus do random I/O) can often be slower in practice than algorithms that access substantially more data but in a sequential order (as in streaming) [17]. It is thus helpful not only to consider the number of block transfers, but also to distinguish between the I/Os that are random versus those that are sequential. In some applications, automated dynamic block placement can improve disk locality and help reduce I/O time [121].

Another simplification of PDM is that the D block transfers in each I/O are *synchronous*; they are assumed to take the same amount of time. This assumption makes

it easier to design and analyze algorithms for multiple disks. In practice, however, if the disks are used independently, some block transfers will complete more quickly than others. We can often improve overall elapsed time if the I/O is done *asynchronously*, so that disks get utilized as soon as they become available. Buffer space in internal memory can be used to queue the read and write requests for each disk.

2.3 Related Memory Models, Hierarchical Memory, and Caching

The study of problem complexity and algorithm analysis when using EM devices began more than 40 years ago with Demuth's Ph.D. dissertation on sorting [48, 83]. In the early 1970s, Knuth [83] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [60, 83] considered a disk model akin to PDM for $D = 1,\ P = 1,\ B = M/2 = \Theta(N^c)$, for constant $c > 0$, and developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [73] developed a pebbling model of I/O for straightline computations, and Savage and Vitter [118] extended the model to deal with block transfer. Aggarwal and Vitter [8] generalized Floyd's I/O model to allow D simultaneous block transfers, but the model was unrealistic in that the D simultaneous transfers were allowed to take place on a single disk. They developed matching upper and lower I/O bounds for all parameter values for a host of problems. Since the PDM model can be thought of as a more restrictive (and more realistic) version of Aggarwal and Vitter's model, their lower bounds apply as well to PDM. In Section 5.4 we discuss a recent simulation technique due to Sanders et al. [116]; the Aggarwal-Vitter model can be simulated probabilistically by PDM with only a constant factor more I/Os, thus making the two models theoretically equivalent in the randomized sense. Deterministic simulations on the other hand require a factor of $\log(N/D)/\log\log(N/D)$ more I/Os [22].

Surveys of I/O models, algorithms, and challenges appear in [12, 65, 124]. Several versions of PDM have been developed for parallel computation [47, 89, 128]. Models of "active disks" augmented with processing capabilities to reduce data traffic to the host, especially during streaming applications, are given in [2, 108]. Models of micro-electromechanical systems (MEMS) for mass storage appear in [68].

Some authors have studied problems that can be solved efficiently by making only one pass (or a small number of passes) over the data [57, 72]. One approach to reduce the internal memory requirements is to require only an approximate answer to the problem; the more memory available, the better the approximation. A related approach to reducing I/O costs for a given problem is to use random sampling or data compression in order to construct a smaller version of the problem whose solution approximates the original. These approaches are highly problem-dependent and somewhat orthogonal to our focus in this survey.

The same type of bottleneck that occurs between internal memory (DRAM) and external disk storage can also occur at other levels of the memory hierarchy, such as between registers and level 1 cache, between level 1 cache and level 2 cache, between level 2 cache and DRAM, and between disk storage and tertiary devices. The PDM model can be generalized to model the hierarchy of memories ranging from registers at the small end to tertiary storage at the large end. Optimal algorithms for PDM often generalize in a recursive fashion to yield optimal algorithms in the hierarchical memory models [6, 5, 151, 149]. Conversely, the algorithms for hierarchical models can be run in the PDM setting, and in that setting many have the interesting property that they use no explicit knowledge of the PDM parameters M and B.

One of the complications in algorithm design with the PDM and hierarchical models is that the algorithms depend on hardware or software parameters, such as M, D,

and B. Frigo et al. [61] and Bender et al. [28] developed the notion of cache-oblivious algorithms and data structures that require no knowledge of the storage parameters. The goal is for a single algorithm to be optimal no matter what the parameter values. We refer the reader to Chapter ??? for more details.

Unfortunately, the match between theory and practice is harder to establish for hierarchical models and caches than for disks. The simpler hierarchical models are less accurate, and the more practical models are architecture-specific. The relative memory sizes and block sizes of the levels vary from computer to computer. Another issue is how blocks from one memory level are stored in the caches at a lower level. When a disk block is read into internal memory, it can be stored in any specified DRAM location. However, in level 1 and level 2 caches, each item can only be stored in certain cache locations, often determined by a hardware modulus computation on the item's memory address. The number of possible storage locations in the cache for a given item is called the level of associativity. Some caches are direct-mapped (i.e., with associativity 1), and most caches have fairly low associativity (typically at most 4).

Another reason why the hierarchical models tend to be more architecture-specific is that the relative difference in speed between level 1 cache and level 2 cache or between level 2 cache and DRAM is orders of magnitude smaller than the relative difference in latencies between DRAM and the disks. Yet, it is apparent that good EM design principles are useful in developing cache-efficient algorithms. For example, sequential internal memory access is much faster than random access, by about a factor of 10, and the more we can build locality into an algorithm, the faster it will run in practice. By properly engineering the "inner loops", a programmer can often significantly speed up the overall running time. Tools such as simulation environments and system monitoring utilities [84, 109, 129] can provide sophisticated help in the optimization process.

For reasons of focus, we do not consider such hierarchical models and caching issues in this survey. We refer the reader to the following references: Aggarwal et al. [5] define an elegant hierarchical memory model, and Aggarwal et al. [6] augment it with block transfer capability. Alpern et al. [9] model levels of memory in which the memory size, block size, and bandwidth grow at uniform rates. Vitter and Shriver [151] and Vitter and Nodine [149] discuss parallel versions and variants of the hierarchical models. The parallel model of Li et al. [89] also applies to hierarchical memory. Savage [117] gives a hierarchical pebbling version of [118]. Carter and Gatlin [30] define pebbling models of nonassociative direct-mapped caches. Rahman and Raman [105] and Sen and Chatterjee [122] apply EM techniques to models of caches and translation lookaside buffers. Rao and Ross [106, 107] use B-tree techniques to exploit locality for the design of cache-conscious search trees.

3 Fundamental I/O Operations and Bounds

The I/O performance of many batched algorithms can be expressed in terms of the bounds for the following three fundamental operations:

1. *Scanning* (a.k.a. *streaming* or *touching*) a file of N data items, which involves the sequential reading or writing of the items in the file.
2. *Sorting* a file of N data items, which puts the items into sorted order.
3. *Outputting* the Z answers to a query in a blocked "output-sensitive" fashion.

We give the I/O bounds for these four operations in Table 2. We separately list the special case of a single disk ($D = 1$), since the formulas are simpler and many of the discussions in this paper will be restricted to the single-disk case.

The I/O bound $Scan(N) = O(n/D)$, which is clearly required to read or write a file of N items, represents a *linear number of I/Os* in the PDM model. An interesting feature of the PDM model is that almost all nontrivial batched problems require a nonlinear number of I/Os, even those that can be solved easily in linear CPU time in the (internal memory) RAM model. Examples we shall discuss later include permuting, transposing a matrix, list ranking, and several combinatorial graph problems. Many of these problems are equivalent in I/O complexity to permuting or sorting.

The linear I/O bounds for $Scan(N)$ and $Output(Z)$ are trivial. The algorithms and lower bounds for $Sort(N)$ are relatively new and are discussed in later sections. As Table 2 indicates, the multiple-disk I/O bounds for $Scan(N)$, $Sort(N)$, and $Output(Z)$ are D times smaller than the corresponding single-disk I/O bounds; such a speedup is clearly the best improvement possible with D disks.

In practice, the logarithmic term $\log_m n$ in the $Sort(N)$ bound is a small constant. For example, in units of items, we could have $N = 10^{10}$, $M = 10^7$, and $B = 10^4$, and thus we get $n = 10^6$, $m = 10^3$, and $\log_m n = 2$, in which case sorting can be done in a linear number of I/Os. If memory is shared with other processes, the $\log_m n$ term will be somewhat larger, but still bounded by a constant.

It still makes sense to explicitly identify the $\log_m n$ term in the I/O bounds and not hide them within the big-oh or big-theta factors, since the terms can have a significant effect in practice. (Of course, it is equally important to consider any other constants hidden in big-oh and big-theta notations!) The nonlinear I/O bound $\Theta(n \log_m n)$ usually indicates that multiple or extra passes over the data are required. In truly massive problems, the data will reside on tertiary storage. As we suggested in Section 2.3, PDM algorithms can often be generalized in a recursive framework to handle multiple levels of memory. A multilevel algorithm developed from a PDM algorithm that does n I/Os will likely run at least an order of magnitude faster in hierarchical memory than would a multilevel algorithm generated from a PDM algorithm that does $n \log_m n$ I/Os [151].

4 Exploiting Locality and Load Balancing

The key to achieving efficient I/O performance in EM applications is to design the application to access its data with a high degree of locality. Since each read I/O operation transfers a block of B items, we make optimal use of that read operation when all B

Table 2. I/O bounds for the fundamental operations. The PDM parameters are defined in Section 2.1.

Operation	I/O bound, $D = 1$	I/O bound, general $D \geq 1$
$Scan(N)$	$\Theta\left(\frac{N}{B}\right) = \Theta(n)$	$\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$
$Sort(N)$	$\Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) = \Theta(n \log_m n)$	$\Theta\left(\frac{N}{DB} \log_{M/B} \frac{N}{B}\right) = \Theta\left(\frac{n}{D} \log_m n\right)$
$Output(Z)$	$\Theta\left(\max\left\{1, \frac{Z}{B}\right\}\right) = \Theta(\max\{1, z\})$	$\Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right) = \Theta\left(\max\left\{1, \frac{z}{D}\right\}\right)$

items are needed by the application. A similar remark applies to write operations. An orthogonal form of locality more akin to load balancing arises when we use multiple disks, since we can transfer D blocks in a single I/O only if the D blocks reside on distinct disks.

An algorithm that does not exploit locality can be reasonably efficient when run on data sets that fit in internal memory, but it will perform miserably when deployed naively in an EM setting, in which virtual memory is used to handle page management. Examining such performance degradation is a good way to put the I/O bounds of Table 2 into perspective. In Section 4.1 we examine this phenomenon for the single-disk case, when $D = 1$.

In Section 4.2, we look at the multiple-disk case and discuss the important paradigm of *disk striping* [81, 111], for automatically converting a single-disk algorithm into an algorithm for multiple disks. Disk striping can be used to get optimal multiple-disk I/O algorithms for three of the four fundamental operations in Table 2. The only exception is sorting. The optimal multiple-disk algorithms for sorting require more sophisticated load balancing techniques, which we cover in Section 5.

4.1 Locality Issues with a Single Disk

A good way to appreciate the fundamental I/O bounds in Table 2 is to consider what happens when an algorithm does not exploit locality. For simplicity, we restrict ourselves in this section to the single-disk case $D = 1$. For many of the batched problems we look at in this paper, such as sorting, FFT, triangulation, and computing convex hulls, it is well known how to write programs to solve the corresponding internal memory versions of the problems in $O(N \log N)$ CPU time. But if we execute such a program on a data set that does not fit in internal memory, relying upon virtual memory to handle page management, the resulting number of I/Os may be $\Omega(N \log n)$, which represents a severe bottleneck.

We would like instead to incorporate locality *directly* into the algorithm design and achieve the desired I/O bound of $O(n \log_m n)$. At the risk of oversimplifying, we can paraphrase the goal of EM algorithm design for batched problems in the following syntactic way: to derive efficient algorithms so that the N and Z terms in the I/O bounds of the naive algorithms are replaced by n and z, and so that the base of the logarithm terms is not 2 but instead m. The relative speedup in I/O performance can be very significant, both theoretically and in practice. For example, if we go from a naive algorithm that uses $\Theta(N \log n)$ I/Os to one that uses only $\Theta(n \log_m n)$ I/Os, the resulting I/O performance improvement is proportional to $(N \log n)/n \log_m n = B \log m$, which can be extremely large.

4.2 Disk Striping for Multiple Disks

It is conceptually much simpler to program for the single-disk case ($D = 1$) than for the multiple-disk case ($D \geq 1$). *Disk striping* [81, 111] is a practical paradigm that can ease the programming task with multiple disks: I/Os are permitted only on entire stripes, one stripe at a time. For example, in the data layout in Figure 4, data items 20–29 can be accessed in a single I/O step because their blocks are grouped into the same stripe. The net effect of striping is that the D disks behave as a single logical disk, but with a larger logical block size DB.

We can thus apply the paradigm of disk striping to automatically convert an algorithm designed to use a single disk with block size DB into an algorithm for use on D disks each with block size B: In the single-disk algorithm, each I/O step transmits one

block of size DB; in the D-disk algorithm, each I/O step consists of D simultaneous block transfers of size B each. The number of I/O steps in both algorithms is the same; in each I/O step, the DB items transferred by the two algorithms are identical. Of course, in terms of wall clock time, the I/O step in the multiple-disk algorithm will be $\Theta(D)$ times faster than in the single-disk algorithm because of parallelism.

Disk striping can be used to get optimal multiple-disk algorithms for three of the four fundamental operations of Section 3—scanning, online search, and output reporting—but it is nonoptimal for sorting. To see why, consider what happens if we use the technique of disk striping in conjunction with an optimal sorting algorithm for one disk, such as merge sort [83]. The optimal number of I/Os to sort using one disk with block size B is

$$\Theta\left(n \log_m n\right) = \Theta\left(n \frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B} \frac{\log(N/B)}{\log(M/B)}\right). \tag{2}$$

With disk striping, the number of I/O steps is the same as if we use a block size of DB in the single-disk algorithm, which corresponds to replacing each B in (2) by DB, which gives the I/O bound

$$\Theta\left(\frac{N}{DB} \frac{\log(N/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D} \frac{\log(n/D)}{\log(m/D)}\right). \tag{3}$$

On the other hand, the optimal bound for sorting is

$$\Theta\left(\frac{n}{D} \log_m n\right) = \Theta\left(\frac{n}{D} \frac{\log n}{\log m}\right). \tag{4}$$

The striping I/O bound (3) is larger than the optimal sorting bound (4) by a multiplicative factor of

$$\frac{\log(n/D)}{\log n} \frac{\log m}{\log(m/D)} \approx \frac{\log m}{\log(m/D)}. \tag{5}$$

When D is on the order of m, the $\log(m/D)$ term in the denominator is small, and the resulting value of (5) is on the order of $\log m$, which can be significant in practice.

It follows that the only way theoretically to attain the optimal sorting bound (4) is to forsake disk striping and to allow the disks to be controlled *independently*, so that each disk can access a different stripe in the same I/O step. Actually, the only requirement for attaining the optimal bound is that either reading or writing is done independently. It suffices, for example, to do only read operations independently and to use disk striping for write operations. An advantage of using striping for write operations is that it facilitates the writing of parity information for error correction and recovery, which is a big concern in RAID systems. (We refer the reader to [31, 71] for a discussion of RAID and error correction issues.)

In practice, sorting via disk striping can be more efficient than complicated techniques that utilize independent disks, especially when D is small, since the extra factor $(\log m)/\log(m/D)$ of I/Os due to disk striping may be less than the algorithmic and system overhead of using the disks independently [138]. In the next section we discuss algorithms for sorting with multiple independent disks. The techniques that arise can be applied to many of the batched problems addressed later in the paper. Two such sorting algorithms—distribution sort with randomized cycling and simple randomized merge sort—have relatively low overhead and will outperform disk-striped approaches.

5 External Sorting and Related Problems

The problem of *external sorting* (or sorting in external memory) is a central problem in the field of EM algorithms, partly because sorting and sorting-like operations account for a significant percentage of computer use [83], and also because sorting is an important paradigm in the design of efficient EM algorithms, as we show in Section 8. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as permuting, list ranking, expression tree evaluation, and finding connected components in a sparse graph, require the same number of I/Os in PDM as does sorting.

Theorem 1 ([8, 99]). *The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is*

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right). \tag{6}$$

We saw in Section 4.2 how to construct efficient sorting algorithms for multiple disks by applying the disk striping paradigm to an efficient single-disk algorithm. But in the case of sorting, the resulting multiple-disk algorithm does not meet the optimal $Sort(N)$ bound of Theorem 1. In Sections 5.1 and 5.2 we discuss some recently developed external sorting algorithms that use disks independently. The algorithms are based upon the important *distribution* and *merge* paradigms, which are two generic approaches to sorting. The SRM method and its variants [23, 27, 114], which are based upon a randomized merge technique, outperform disk striping in practice for reasonable values of D. All the algorithms use online load balancing strategies so that the data items accessed in an I/O operation are evenly distributed on the D disks. The same techniques can be applied to many of the batched problems we discuss later in this survey.

All the methods we cover for parallel disks, with the exception of Greed Sort in Section 5.2, provide efficient support for writing redundant parity information onto the disks for purposes of error correction and recovery. For example, some of the methods access the D disks independently during parallel read operations, but in a striped manner during parallel writes. As a result, if we write $D-1$ blocks at a time, the exclusive-or of the $D-1$ blocks can be written onto the Dth disk during the same write operation.

In Section 5.4, we show that if we allow independent reads and writes, we can probabilistically simulate any algorithm written for the Aggarwal-Vitter model discussed in Section 2.3 by use of PDM with the same number of I/Os, up to a constant factor.

In Section 5.5 we consider the situation in which the items in the input file do not have unique keys. In Sections 5.6 and 5.7 we consider problems related to sorting, such as permuting, permutation networks, transposition, and fast Fourier transform. In Section 10, we give lower bounds for sorting and related problems.

5.1 Sorting by Distribution

Distribution sort [83] is a recursive process in which we use a set of $S-1$ partitioning elements to partition the items into S disjoint buckets. All the items in one bucket precede all the items in the next bucket. We complete the sort by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

One requirement is that we choose the $S-1$ partitioning elements so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease

from one level of recursion to the next by a relative factor of $\Theta(S)$, and thus there are $O(\log_S n)$ levels of recursion. During each level of recursion, we scan the data. As the items stream through internal memory, they are partitioned into S buckets in an online manner. When a buffer of size B fills for one of the buckets, its block is written to the disks in the next I/O, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number of buckets (and partitioning elements) is $S = \Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$.

It seems difficult to find $S = \Theta(m)$ partitioning elements using $\Theta(n/D)$ I/Os and guarantee that the bucket sizes are within a constant factor of one another. Efficient deterministic methods exist for choosing $S = \sqrt{m}$ partitioning elements [8, 98, 150], which has the effect of doubling the number of levels of recursion. Probabilistic methods based upon random sampling can be found in [58]. A deterministic algorithm for the related problem of (exact) selection (i.e., given k, find the kth item in the file in sorted order) appears in [127].

In order to meet the sorting bound (6), we must form the buckets at each level of recursion using $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each read step and each write step during the bucket formation must involve on the average $\Theta(D)$ blocks. The file of items being partitioned is itself one of the buckets formed in the previous level of recursion. In order to read that file efficiently, its blocks must be spread uniformly among the disks, so that no one disk is a bottleneck. The challenge in distribution sort is to write the blocks of the buckets to the disks in an online manner and achieve a global load balance by the end of the partitioning, so that the bucket can be read efficiently during the next level of the recursion.

Partial striping is an effective technique for reducing the amount of information that must be stored in internal memory in order to manage the disks. The disks are grouped into clusters of size C and data are written in "logical blocks" of size CB, one per cluster. Choosing $C = \sqrt{D}$ won't change the optimal sorting time by more than a constant factor, but as pointed out in Section 4.2, full striping (in which $C = D$) can be nonoptimal.

Vitter and Shriver [150] develop two randomized online techniques for the partitioning so that with high probability each bucket will be well balanced across the D disks. In addition, they use partial striping in order to fit in internal memory the pointers needed to keep track of the layouts of the buckets on the disks. Their first partitioning technique applies when the size N of the file to partition is sufficiently large or when $M/DB = \Omega(\log D)$, so that the number $\Theta(n/S)$ of blocks in each bucket is $\Omega(D \log D)$. Each parallel write operation writes its D blocks in independent random order to a disk stripe, with all $D!$ orders equally likely. At the end of the partitioning, with high probability each bucket is evenly distributed among the disks. This situation is intuitively analogous to the *classical occupancy problem*, in which b balls are inserted independently and uniformly at random into d bins. It is well-known that if the load factor b/d grows asymptotically faster than $\log d$, the most densely populated bin contains b/d balls asymptotically on the average, which corresponds to an even distribution. However if the load factor b/d is 1, the largest bin contains $(\ln d)/\ln\ln d$ balls, whereas an average bin contains only one ball [147]. Intuitively, the blocks in a bucket act as balls and the disks act as bins. In our case, the parameters correspond to $b = \Omega(d \log d)$, which suggests that the blocks in the bucket should be evenly distributed among the disks.

By further analogy to the occupancy problem, if the number of blocks per bucket is not $\Omega(D \log D)$, then the technique breaks down and the distribution of each bucket

among the disks tends to be uneven, causing a bottleneck for I/O operations. For these smaller values of N, Vitter and Shriver use their second partitioning technique: The file is read in one pass, one memoryload at a time. Each memoryload is independently and randomly permuted and written back to the disks in the new order. In a second pass, the file is accessed one memoryload at a time in a "diagonally striped" manner. Vitter and Shriver show that with very high probability each individual "diagonal stripe" contributes about the same number of items to each bucket, so the blocks of the buckets in each memoryload can be assigned to the disks in a balanced round robin manner using an optimal number of I/Os.

DeWitt et al. [50] present a randomized distribution sort algorithm in a similar model to handle the case when sorting can be done in two passes. They use a sampling technique to find the partitioning elements and route the items in each bucket to a particular processor. The buckets are sorted individually in the second pass.

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [98]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. It maintains an invariant that guarantees good distribution across the disks for each bucket. For each bucket $1 \le b \le S$ and disk $1 \le d \le D$, let num_b be the total number of items in bucket b processed so far during the partitioning and let $num_b(d)$ be the number of those items written to disk d; that is, $num_b = \sum_{1 \le d \le D} num_b(d)$. By application of matching techniques from graph theory, the BalanceSort algorithm is guaranteed to write at least half of any given memoryload to the disks in a blocked manner and still maintain the invariant for each bucket b that the $\lfloor D/2 \rfloor$ largest values among $num_b(1)$, $num_b(2)$, ..., $num_b(D)$ differ by at most 1. As a result, each $num_b(d)$ is at most about twice the ideal value num_b/D, which implies that the number of I/Os needed to read a bucket into memory during the next level of recursion will be within a small constant factor of optimal.

The distribution sort methods that we mentioned above for parallel disks perform write operations in complete stripes, which makes it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. As a result, the write operations must use disks independently, since during each write, multiple buckets will be writing to multiple stripes. Error correction and recovery can still be handled efficiently by devoting to each bucket one block-sized buffer in internal memory. The buffer is continuously updated to contain the exclusive-or (parity) of the blocks written to the current stripe, and after $D-1$ blocks have been written, the parity information in the buffer can be written to the final (Dth) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to read one memoryload at a time and partition the items into S buckets. However, unlike before, the blocks for each individual bucket will reside on the disks in contiguous stripes. Each block therefore has a predefined place where it must be written. If we choose the normal round-robin ordering for the stripes (namely, $\ldots, 1, 2, 3, \ldots, D, 1, 2, 3, \ldots, D, \ldots$), the blocks of different buckets may "collide", meaning that they need to be written to the same disk, and subsequent blocks in those same buckets will also tend to collide. Vitter and Hutchinson [148] solve the

problem by the technique of *randomized cycling*. For each of the S buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1, 2, \ldots, D\}$. The S random permutations are chosen independently. If two blocks (from different buckets) happen to collide during a write to the same disk, one block is written to the disk and the other is kept on a write queue. With high probability, subsequent blocks in those two buckets will be written to different disks and thus will not collide. As long as there is a small pool of available buffer space to temporarily cache the blocks in the write queues, Vitter and Hutchinson show that with high probability the writing proceeds optimally.

We expect that the randomized cycling method or the related merge sort methods discussed at the end of Section 5.2 will be the methods of choice for sorting with parallel disks. Experiments are underway to evaluate their relative performance. Distribution sort algorithms may have an advantage over the merge approaches presented in Section 5.2 in that they typically make better use of lower levels of cache in the memory hierarchy of real systems, based upon analysis of distribution sort and merge sort algorithms on models of hierarchical memory, such as the RUMH model of Vitter and Nodine [149].

5.2 Sorting by Merging

The *merge* paradigm is somewhat orthogonal to the distribution paradigm of the previous section. A typical merge sort algorithm works as follows [83]: In the "run formation" phase, we scan the n blocks of data, one memoryload at a time; we sort each memoryload into a single "run", which we then output onto a series of stripes on the disks. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, we can use the "replacement-selection" technique to get runs of $2M$ data items, on the average, when $M \gg B$ [83].) After the initial runs are formed, the merging phase begins. In each pass of the merging phase, we merge groups of R runs. For each merge, we scan the R runs and merge the items in an online manner as they stream through internal memory. Double buffering is used to overlap I/O and computation. At most $R = \Theta(m)$ runs can be merged at a time, and the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (6), we must perform each merging pass in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

For the binary merging case $R = 2$ we can devise a perfect solution, in which the next D blocks needed for the merge are guaranteed to be on distinct disks, based upon the Gilbreath principle [64, 83]: We stripe the first run into ascending order by disk number, and we stripe the other run into descending order. Regardless of how the items in the two runs interleave during the merge, it is always the case that we can access the next D blocks needed for the output via a single I/O operation, and thus the amount of internal memory buffer space needed for binary merging is minimized. Unfortunately there is no analogue to the Gilbreath principle for $R > 2$, and as we have seen above, we need the value of R to be large in order to get an optimal sorting algorithm.

The Greed Sort method of Nodine and Vitter [99] was the first optimal deterministic EM algorithm for sorting with multiple disks. It handles the case $R > 2$ by relaxing the condition on the merging process. In each step, two blocks from each disk are brought into internal memory: the block b_1 with the smallest data item value and the block b_2 whose largest item value is smallest. If $b_1 = b_2$, only one block is read into memory, and it is added to the next output stripe. Otherwise, the two blocks b_1 and b_2 are merged in memory; the smaller B items are written to the output stripe, and the remaining B items are written back to the disk. The resulting run that is produced is only an "approximately" merged run, but its saving grace is that no two inverted items are too far apart. A final application of Columnsort [87] suffices to restore total order; partial striping is employed to meet the memory constraints. One disadvantage of Greed Sort is that the block writes and block reads involve independent disks and are not done in a striped manner, thus making it difficult to write parity information for error correction and recovery.

Aggarwal and Plaxton [7] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm [44]. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious. Like Greed Sort, the Sharesort algorithm is theoretically optimal (i.e., within a constant factor of optimal), but the constant factor is larger than the distribution sort methods.

One of the most practical methods for sorting is based upon the *simple randomized merge sort* (SRM) algorithm of Barve et al. [23, 27], referred to as "randomized striping" by Knuth [83]. Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space. Barve et al. [23] derive an asymptotic upper bound on the expected I/O performance, with no assumptions on the input distribution. A more precise analysis, which is related to the so-called *cyclic occupancy problem*, is an interesting open problem. The cyclic occupancy problem is similar to the classical occupancy problem we discussed in Section 5.1 in that there are b balls distributed into d bins. However, in the cyclical occupancy problem, the b balls are grouped into c chains of length $b_1, b_2, \ldots b_c$, where $\sum_{1 \le i \le c} b_i = b$. Only the head of each chain is randomly inserted into a bin; the remaining balls of the chain are inserted into the successive bins in a cyclic manner (hence the name "cyclic occupancy"). It is conjectured that the expected maximum bin size in the cyclic occupancy problem is at most that of the classical occupancy problem [23, 83, problem 5.4.9–28]. The bound has been established so far only in an asymptotic sense.

The expected performance of SRM is not optimal for some parameter values, but it significantly outperforms the use of disk striping for reasonable values of the parameters, as shown in Table 3. Experimental confirmation of the speedup was obtained on a 500 megahertz CPU with six fast disk drives, as reported by Barve and Vitter [27].

We can get further improvements in merge sort by a more careful prefetching schedule for the runs. Barve et al. [24] and Kallahalla and Varman [77, 78] have developed competitive and optimal methods for prefetching blocks in parallel I/O systems.

5.3 Duality between Distribution and Merging

Hutchinson et al. [75, 76] have demonstrated a powerful duality between the process of writing a sequence of blocks to parallel disks and the process of reading a sequence of blocks from parallel disks. The writing process is straightforward: In order to write

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	0.56	0.47	0.37
$k = 10$	0.61	0.52	0.40
$k = 50$	0.71	0.63	0.51

Table 3. The ratio of the number of I/Os used by simple randomized merge sort (SRM) to the number of I/Os used by merge sort with disk striping, during a merge of kD runs, where $kD \approx M/2B$. The figures were obtained by simulation.

sequence Σ of distinct blocks, each with a specified disk location, each block, when accessed, is placed into a buffer. At each I/O step, a block is written to each disk that has at least one outstanding write request. The duality principle tells us that the I/O schedule for Σ, when run in reverse, gives a valid I/O schedule for reading the reverse sequence Σ' of blocks. How to construct an optimal prefetching (read) schedule for a sequence Σ' had long been an open problem [77]. By duality, however, it is easily solvable by computing the optimal schedule for the write problem on the reverse sequence Σ and then reversing the I/O schedule. A related duality exists as well for the problem of reading a sequence of blocks with repetitions, in which caching comes into play [75, 76].

Duality can be extended to the two sorting paradigms of distribution and merging. In particular, we can reduce the merge process to one of reading a sequence of blocks. The difficulty is that the input order Σ' for the blocks, namely, the order in which the blocks need to be accessed during the merge, is highly data-dependent and not known in advance. The key to duality is to characterize Σ' in a simple and easily implementable way. If we examine the process of merging, as illustrated in Figure 5 from the bottom to top, we see that the merging buffer contains a partially filled block from each run (not yet expired). When the block empties all its items into the merged output stream, the next block from that run can be inserted into the merging buffer. The merging buffer is pictured in the upper rectangle in Figure 5), which is distinct from the space reserved for the prefetch buffers (lower rectangle in Figure 5).

The first moment, therefore, when a block absolutely needs to be present in memory is when the smallest key value of the block is merged into the output stream. We therefore define the *trigger* of a block to be the key value of the smallest key in the block. We say that a block is accessed (or read) when it is moved from the prefetch buffer to the merging buffer, where it stays until its items are exhausted by the merging process. Thus, the access (or read) order Σ' of the blocks is given by the sorted order of the triggers.

We have now reduced the merging problem to that of prefetching for the input sequence Σ'. By duality, we can compute an optimal I/O schedule for Σ' by computing the optimal I/O schedule for writing the reverse sequence Σ, which is straightforward. By reversing the I/O schedule, we get an optimal I/O schedule for merging.

The problem remains, however, to determine a striping discipline for laying out the buckets on the disks. When the placement of the blocks of each run is done by randomized cycling [148], as described in Section 5.1, the expected number of I/O steps in the schedule computed via duality meets the optimal sorting bound $Sort(N)$. The resulting merge sort algorithm is optimal up to second-order terms and is the method

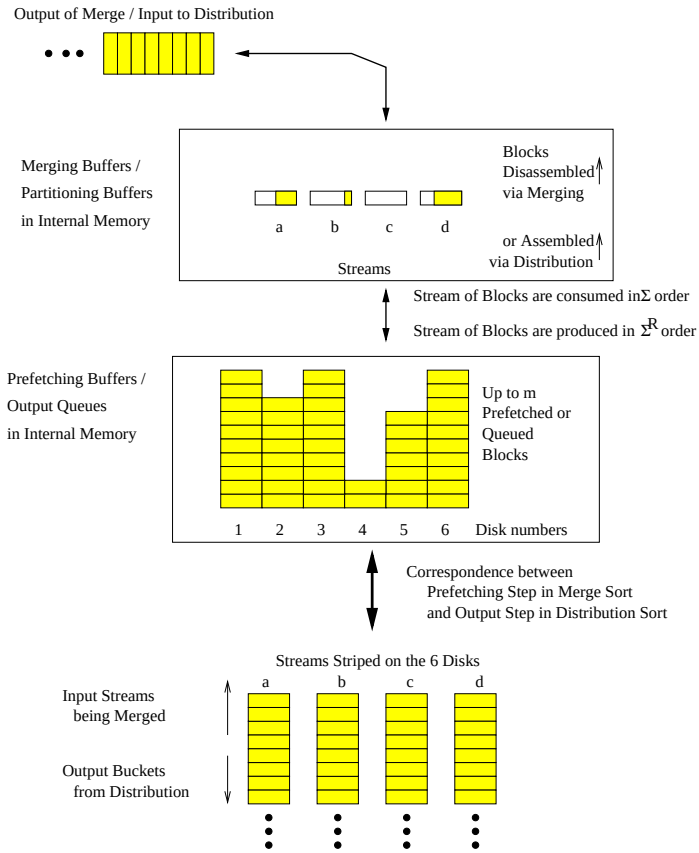

Fig. 5. The duality between merging and distribution. Buffer space is required both "privately" within the application (for storing the lead block of each run in merging and for storing the next block being formed for each bucket in distribution) and for the I/O buffers. For merging, blocks are read in the order Σ' given by the triggers, the first key values from each block.

of choice in practice, along with the distribution sort method based on randomized cycling discussed at the end of Section 5.1.

5.4 A General Simulation

Sanders et al. [116] and Sanders [115] give an elegant randomized technique to simulate the Aggarwal-Vitter model of Section 2.3, in which D simultaneous block transfers are allowed regardless of where the blocks are located on the disks. On the average, the simulation realizes each I/O in the Aggarwal-Vitter model by only a constant number of I/Os in PDM. One property of the technique is that the read and write steps use the disks independently. Armen [22] had earlier shown that deterministic simulations resulted in an increase in the number of I/Os by a multiplicative factor of $\log(N/D)/\log\log(N/D)$.

The technique of Sanders et al. consists of duplicating each disk block and storing the two copies on two independently and uniformly chosen disks (chosen by a hash function). In terms of the occupancy model, each ball (block) is duplicated and stored in two random bins (disks). Let us consider the problem of retrieving a specific set of D blocks from the disks. For each block, there is a choice of two disks from which it can be read. Regardless of which D blocks are requested, Sanders et al. show how to choose the copies that permit the D blocks to be retrieved with high probability in only two parallel I/Os. A natural application of this technique is to the layout of data on multimedia servers in order to support multiple stream requests, as in video-on-demand.

When writing blocks of data to the disks, each block must be written to both the disks where a copy is stored. Sanders et al. prove that with high probability D blocks can be written with a constant number of I/O steps on the average, assuming that there are $O(D)$ blocks of internal buffer space to serve as write queues. Blocks are queued until the disk is available for writing. The key part of the analysis is showing that the internal buffer space does not overflow, except with exponentially small probability. The read and write bounds can be improved with a corresponding tradeoff in redundancy and internal memory space.

5.5 Handling Duplicates

Arge et al. [15] describe a single-disk merge sort algorithm for the problem of *duplicate removal*, in which there are a total of K distinct items among the N items. It runs in $O\bigl(n \max\{1, \log_m(K/B)\}\bigr)$ I/Os, which is optimal in the comparison model. The algorithm can be used to sort the file, assuming that a group of equal items can be represented by a single item and a count.

A harder instance of sorting called *bundle sorting* arises when we have K distinct key values among the N items, but all the items have different secondary information. Abello et al. [1] and Matias et al. [92] develop optimal distribution sort algorithms for bundle sorting using $\mathit{BundleSort}(N, K) = O\bigl(n \max\{1, \log_m \min\{K, n\}\}\bigr)$ I/Os, and Matias et al. [92] prove the matching lower bound. Matias et al. [92] also show how to do bundle sorting (and sorting in general) *in place* (i.e., without extra disk space). In distribution sort, for example, the blocks for the subfiles can be allocated from the blocks freed up from the file being partitioned; the disadvantage is that the blocks in the individual subfiles are no longer consecutive on the disk. The algorithms can be adapted to run on D disks with a speedup of $O(D)$ using the techniques described in Sections 5.1 and 5.2.

5.6 Permuting and Transposition

Permuting is the special case of sorting in which the key values of the N data items form a permutation of $\{1, 2, \ldots, N\}$.

Theorem 2 ([8]). *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, \mathit{Sort}(N)\right\}\right). \tag{7}$$

The I/O bound (7) for permuting can be realized by using one of the sorting algorithms from Section 5 except in the extreme case $B \log m = o(\log n)$, in which case it is faster to move the data items one by one in a nonblocked way. The one-by-one method is trivial if $D = 1$, but with multiple disks there may be bottlenecks on

individual disks; one solution for doing the permuting in $O(N/D)$ I/Os is to apply the randomized balancing strategies of [150].

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

Theorem 3 ([8]). *With D disks, the number of I/Os required to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right), \tag{8}$$

where $N = pq$ and $n = N/B$.

When B is relatively large (say, $\frac{1}{2}M$) and N is $O(M^2)$, matrix transposition can be as hard as general sorting, but for smaller B, the special structure of the transposition permutation makes transposition easier. In particular, the matrix can be broken up into square submatrices of B^2 elements such that each submatrix contains B blocks of the matrix in row-major order and also B blocks of the matrix in column-major order. Thus, if $B^2 < M$, the transpositions can be done in a simple one-pass operation by transposing the submatrices one at a time in internal memory.

Matrix transposition is a special case of a more general class of permutations called *bit-permute/complement* (BPC) permutations, which in turn is a subset of the class of *bit-matrix-multiply/complement* (BMMC) permutations. BMMC permutations are defined by a $\log N \times \log N$ nonsingular 0-1 matrix A and a $(\log N)$-length 0-1 vector c. An item with binary address x is mapped by the permutation to the binary address given by $Ax \oplus c$, where $\oplus$ denotes bitwise exclusive-or. BPC permutations are the special case of BMMC permutations in which A is a permutation matrix, that is, each row and each column of A contain a single 1. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen et al. [40] characterize the optimal number of I/Os needed to perform any given BMMC permutation solely as a function of the associated matrix A, and they give an optimal algorithm for implementing it.

Theorem 4 ([40]). *With D disks, the number of I/Os required to perform the BMMC permutation defined by matrix A and vector c is*

$$\Theta\left(\frac{n}{D}\left(1 + \frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \tag{9}$$

where γ is the lower-left $\log n \times \log B$ submatrix of A.

An interesting theoretical question is to determine the I/O cost for each individual permutation, as a function of some simple characterization of the permutation, such as number of inversions.

5.7 Fast Fourier Transform and Permutation Networks

Computing the fast Fourier transform (FFT) in external memory consists of a series of I/Os that permit each computation implied by the FFT directed graph (or butterfly) to be done while its arguments are in internal memory. A permutation network computation consists of an oblivious (fixed) pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs [158].

Theorem 5. *With D disks, the number of I/Os required for computing the N-input FFT digraph or an N-input permutation network is $Sort(N)$.*

Cormen and Nicol [39] give some practical implementations for one-dimensional FFTs based upon the optimal PDM algorithm of [150]. The algorithms for FFT are faster and simpler than for sorting because the computation is nonadaptive in nature, and thus the communication pattern is fixed in advance.

6 Matrix and Grid Computations

Dense matrices are generally represented in memory in row-major or column-major order. Matrix transposition, which is the special case of sorting that involves conversion of a matrix from one representation to the other, was discussed in Section 5.6. For certain operations such as matrix addition, both representations work well. However, for standard matrix multiplication (using only semiring operations) and LU decomposition, a better representation is to block the matrix into square $\sqrt{B} \times \sqrt{B}$ submatrices, which gives the upper bound of the following theorem:

Theorem 6 ([73, 118, 150, 157]). *The number of I/Os required for standard matrix multiplication of two $K \times K$ matrices or to compute the LU factorization of a $K \times K$ matrix is $\Theta\left(K^3/\min\{K, \sqrt{M}\}DB\right)$.*

Hong and Kung [73] and Nodine et al. [97] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [88] reduce the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [70] show how to derive efficient EM algorithms automatically for computations expressed in tensor form.

If a $K \times K$ matrix A is sparse, that is, if the number N_z of nonzero elements in A is much smaller than K^2, then it may be more efficient to store only the nonzero elements. Each nonzero element $A_{i,j}$ is represented by the triple $(i, j, A_{i,j})$. Unlike the dense case, in which transposition can be easier than sorting (e.g., see Theorem 3 when $B^2 \leq M$), transposition of sparse matrices is as hard as sorting:

Theorem 7. *For a matrix stored in sparse format and containing N_z nonzero elements, the number of I/Os required to convert the matrix from row-major order to column-major order, and vice-versa, is $\Theta\left(Sort(N_z)\right)$.*

The lower bound follows by reduction from sorting. If the ith item in the input of the sorting instance has key value $x \neq 0$, there is a nonzero element in matrix position (i, x).

For further discussion of numerical EM algorithms we refer the reader to the survey by Toledo [134]. Some issues regarding programming environments are covered in [36] and Section 9.

7 Batched Problems in Computational Geometry

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [86, 112, 113], geographic information systems (GIS) [86, 112, 137], constraint logic programming [79, 80], object-oriented databases [159], statistics, virtual reality systems, and computer graphics [62]. NASA's Earth Observing System project, the core part of the Earth Science Enterprise (formerly Mission to Planet Earth), produces petabytes (10^{15} bytes) of raster data per year [54]! Microsoft's TerraServer online

database of satellite images is over one terabyte in size [131]. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.[2]

For systems of this size to be efficient, we need fast EM algorithms for basic problems in computational geometry. Luckily, many problems on geometric objects can be reduced to a small core of problems, such as computing intersections, convex hulls, or nearest neighbors. Useful paradigms have been developed for solving these problems in external memory.

Theorem 8. *Certain batched problems involving $N = nB$ input items, $Q = qB$ queries, and $Z = zB$ output items can be solved using*

$$O\big((n+q)\log_m n + z\big) \tag{10}$$

I/Os with a single disk:

1. *Computing the pairwise intersections of N segments in the plane and their trapezoidal decomposition;*
2. *Finding all intersections between N nonintersecting red line segments and N nonintersecting blue line segments in the plane;*
3. *Answering Q orthogonal 2-D range queries on N points in the plane (i.e., finding all the points within the Q query rectangles);*
4. *Constructing the 2-D and 3-D convex hull of N points;*
5. *Voronoi diagram and Triangulation of N points in the plane;*
6. *Performing Q point location queries in a planar subdivision of size N;*
7. *Finding all nearest neighbors for a set of N points in the plane;*
8. *Finding the pairwise intersections of N orthogonal rectangles in the plane;*
9. *Computing the measure of the union of N orthogonal rectangles in the plane;*
10. *Computing the visibility of N segments in the plane from a point; and*
11. *Performing Q ray-shooting queries in 2-D Constructive Solid Geometry (CSG) models of size N.*

The parameters Q and Z are set to 0 if they are not relevant for the particular problem.

Goodrich et al. [67], Zhu [161], Arge et al. [21], Arge et al. [19], and Crauser et al. [41, 42] develop EM algorithms for those problems using these EM paradigms for batched problems:

Distribution sweeping, a generalization of the distribution paradigm of Section 5 for "externalizing" plane sweep algorithms.

Persistent B-trees, an offline method for constructing an optimal-space persistent version of the B-tree data structure (see Chapter ???), yielding a factor of B improvement over the generic persistence techniques of Driscoll et al. [52].

Batched filtering, a general method for performing simultaneous EM searches in data structures that can be modeled as planar layered directed acyclic graphs; it is useful for 3-D convex hulls and batched point location. Multisearch on parallel computers is considered in [51].

[2] For brevity, in the remainder of this survey we deal only with the single-disk case $D = 1$. The single-disk I/O bounds for the batched problems can often be cut by a factor of $\Theta(D)$ for the case $D \geq 1$ by using the load balancing techniques of Section 5. In practice, disk striping (cf. Section 4.2) may be sufficient.

External fractional cascading, an EM analogue to fractional cascading on a segment tree, in which the degree of the segment tree is $O(m^\alpha)$ for some constant $0 < \alpha \leq 1$. Batched queries can be performed efficiently using batched filtering.

External marriage-before-conquest, an EM analogue to the technique of Kirkpatrick and Seidel [82] for performing output-sensitive convex hull constructions.

Batched incremental construction, a localized version of the randomized incremental construction paradigm of Clarkson and Shor [34], in which the updates to a simple dynamic data structure are done in a random order, with the goal of fast overall performance on the average. The data structure itself may have bad worst-case performance, but the randomization of the update order makes worst-case behavior unlikely. The key for the EM version so as to gain the factor of B I/O speedup is to batch together the incremental modifications.

We focus in the remainder of this section primarily on the distribution sweep paradigm [67], which is a combination of the distribution paradigm of Section 5.1 and the well-known sweeping paradigm from computational geometry [104, 45]. As an example, let us consider computing the pairwise intersections of N orthogonal segments in the plane by the following recursive distribution sweep: At each level of recursion, the region under consideration is partitioned into $\Theta(m)$ vertical *slabs*, each containing $\Theta(N/m)$ of the segments' endpoints. We sweep a horizontal line from top to bottom to process the N segments. When the sweep line encounters a vertical segment, we insert the segment into the appropriate slab. When the sweep line encounters a horizontal segment h, as pictured in Figure 6, we report h's intersections with all the "active" vertical segments in the slabs that are spanned *completely* by h. (A vertical segment is "active" if it intersects the current sweep line; vertical segments that are found to be no longer active are deleted from the slabs.) The remaining two end portions of h (which "stick out" past a slab boundary) are passed recursively to the next level, along with the vertical segments. The downward sweep then proceeds. After the initial sorting (to get the segments with respect to the y-dimension), the sweep at each of the $O(\log_m n)$ levels of recursion requires $O(n)$ I/Os, yielding the desired bound (10). Some timing experiments on distribution sweeping appear in [32]. Arge et al. [19] develop a unified approach to distribution sweep in higher dimensions.

A central operation in spatial databases is spatial join. A common preprocessing step is to find the pairwise intersections of the bounding boxes of the objects involved in the spatial join. The problem of intersecting orthogonal rectangles can be solved by combining the previous sweep line algorithm for orthogonal segments with one for range searching. Arge et al. [19] take a more unified approach using distribution sweep, which is extendible to higher dimensions: The active objects that are stored in the data structure in this case are rectangles, not vertical segments. The authors choose the branching factor to be $\Theta(\sqrt{m})$. Each rectangle is associated with the largest contiguous range of vertical slabs that it spans. Each of the possible $\Theta\left(\binom{\sqrt{m}}{2}\right) = \Theta(m)$ contiguous ranges of slabs is called a *multislab*. The reason why the authors choose the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ is so that the number of multislabs is $\Theta(m)$, and thus there is room in internal memory for a buffer for each multislab. The height of the tree remains $O(\log_m n)$.

The algorithm proceeds by sweeping a horizontal line from top to bottom to process the N rectangles. When the sweep line first encounters a rectangle R, we consider the multislab lists for all the multislabs that R intersects. We report all the active rectangles in those multislab lists, since they are guaranteed to intersect R. (Rectangles no longer active are discarded from the lists.) We then extract the left and right end portions

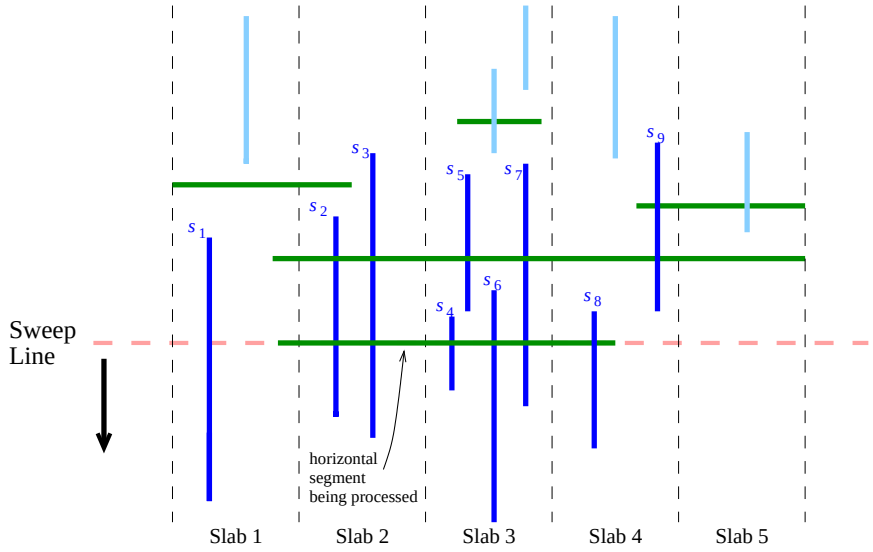

Fig. 6. Distribution sweep used for finding intersections among N orthogonal segments. The vertical segments currently stored in the slabs are indicated in bold (namely, s_1, $s_2, \ldots, s_9$). Segments s_5 and s_8 are not active, but have not yet been deleted from the slabs. The sweep line has just advanced to a new horizontal segment that completely spans slabs 2 and 3, so slabs 2 and 3 are scanned and all the active vertical segments in slabs 2 and 3 (namely, s_2, s_3, s_4, s_6, s_7) are reported as intersecting the horizontal segment. In the process of scanning slab 3, segment s_5 is discovered to be no longer active and can be deleted from slab 3. The end portions of the horizontal segment that "stick out" into slabs 1 and 4 are handled by the lower levels of recursion, where the intersection with s_8 is eventually discovered.

of R that partially "stick out" past slab boundaries, and we pass them down to process in the next lower level of recursion. We insert the remaining portion of R, which spans complete slabs, into the list for the appropriate multislab. The downward sweep then continues. After the initial sorting preprocessing, each of the $O(\log_m n)$ sweeps (one per level of recursion) takes $O(n)$ I/Os, yielding the desired bound (10).

The resulting algorithm, called Scalable Sweeping-Based Spatial Join (SSSJ) [18, 19], outperforms other techniques for rectangle intersection. It was tested against two other sweep line algorithms: the Partition-Based Spatial-Merge (QPBSM) used in Paradise [103] and a faster version called MPBSM that uses an improved dynamic data structure for intervals [18]. The TPIE system described in Section 9 served as the common implementation platform. The algorithms were tested on several data sets. The timing results for the two data sets in Figures 7(a) and 7(b) are given in Figures 7(c) and 7(d), respectively. The first data set is the worst case for sweep line algorithms; a large fraction of the line segments in the file are active (i.e., they intersect the current sweep line). The second data set is a best case for sweep line algorithms, but the two PBSM algorithms have the disadvantage of making extra copies of the rectangles. In both cases, SSSJ shows considerable improvement over the PBSM-based methods. In other experiments done on more typical data, such as TIGER/line road data sets [133], SSSJ and MPBSM perform about 30% faster than does QPBSM. The conclusion we

draw is that SSSJ is as fast as other known methods on typical data, but unlike other methods, it scales well even for worst-case data. If the rectangles are already stored in an index structure, such as the R-tree index structure, hybrid methods that combine distribution sweep with inorder traversal often perform best [17].

For the problem of finding all intersections among N line segments, Arge et al. [21] give an efficient algorithm based upon distribution sort, but the output component of the I/O bound is slightly nonoptimal: $z \log_m n$ rather than z. Crauser et al. [41, 42] attain the optimal I/O bound (10) by constructing the trapezoidal decomposition for the intersecting segments using an incremental randomized construction. For I/O efficiency, they do the incremental updates in a series of batches, in which the batch size is geometrically increasing by a factor of m.

8 Batched Problems on Graphs

The first work on EM graph algorithms was by Ullman and Yannakakis [136] for the problem of transitive closure. Chiang et al. [33] consider a variety of graph problems, several of which have upper and lower I/O bounds related to sorting and permuting. Abello et al. [1] formalize a functional approach to EM graph problems, in which computation proceeds in a series of scan operations over the data; the scanning avoids side effects and thus permits checkpointing to increase reliability. Kumar and Schwabe [85], followed by Buchsbaum et al. [29], develop graph algorithms based upon amortized data structures for binary heaps and tournament trees. Munagala and Ranade [95] give improved graph algorithms for connectivity and undirected breadth-first search, and Arge et al. [13] extend the approach to compute the minimum spanning forest (MSF). Meyer [93] provides some improvements for graphs of bounded degree. Arge [11] gives efficient algorithms for constructing ordered binary decision diagrams. Grossi and Italiano [69] apply their multidimensional data structure to get dynamic EM algorithms for MSF and two-dimensional priority queues (in which the *delete_min* operation is replaced by $delete\_min_x$ and $delete\_min_y$). Techniques for storing graphs on disks for efficient traversal and shortest path queries are discussed in [4, 66, 74, 96]. Computing wavelet decompositions and histograms [152, 153, 155] is an EM graph problem related to transposition that arises in On-Line Analytical Processing (OLAP). Wang et al. [154] give an I/O-efficient algorithm for constructing classification trees for data mining.

Table 4 gives the best known I/O bounds for several graph problems, as a function of the number $V = vB$ of vertices and the number $E = eB$ of edges. The best known I/O lower bound for these problems is $\Omega\big((E/V)Sort(V) = e \log_m v\big)$, as mentioned in Section 10. A sparsification technique [55] can often be applied to convert I/O bounds of the form $O\big(Sort(E)\big)$ to the improved form $O\big((E/V)Sort(V)\big)$. For example, the actual I/O bounds for connectivity and MSF derived by Munagala and Ranade [95] and Arge et al. [13] are $O\big(\max\{1, \log\log(V/e)\}Sort(E)\big)$. For the MSF problem, we can partition the edges of the graph into E/V sparse subgraphs on V vertices, and then apply the algorithm of [13] to each subproblem to create E/V spanning forests in a total of $O\big(\max\{1, \log\log(V/e)\}(E/V)Sort(V)\big)$ I/Os. We can then merge the E/V spanning forests, two at a time, in a balanced binary merging procedure by repeatedly applying the algorithm of [13]. After the first level of binary merging, the spanning forests collectively have at most $E/2$ edges; after two levels, they have at most $E/4$ edges, and so on in a geometrically decreasing manner. The total cost for the final spanning forest is thus $O\big(\max\{1, \log\log(V/e)\}(E/V)Sort(V)\big)$ I/Os. The reason why sparsification works is that the spanning forest output by each binary merge is only

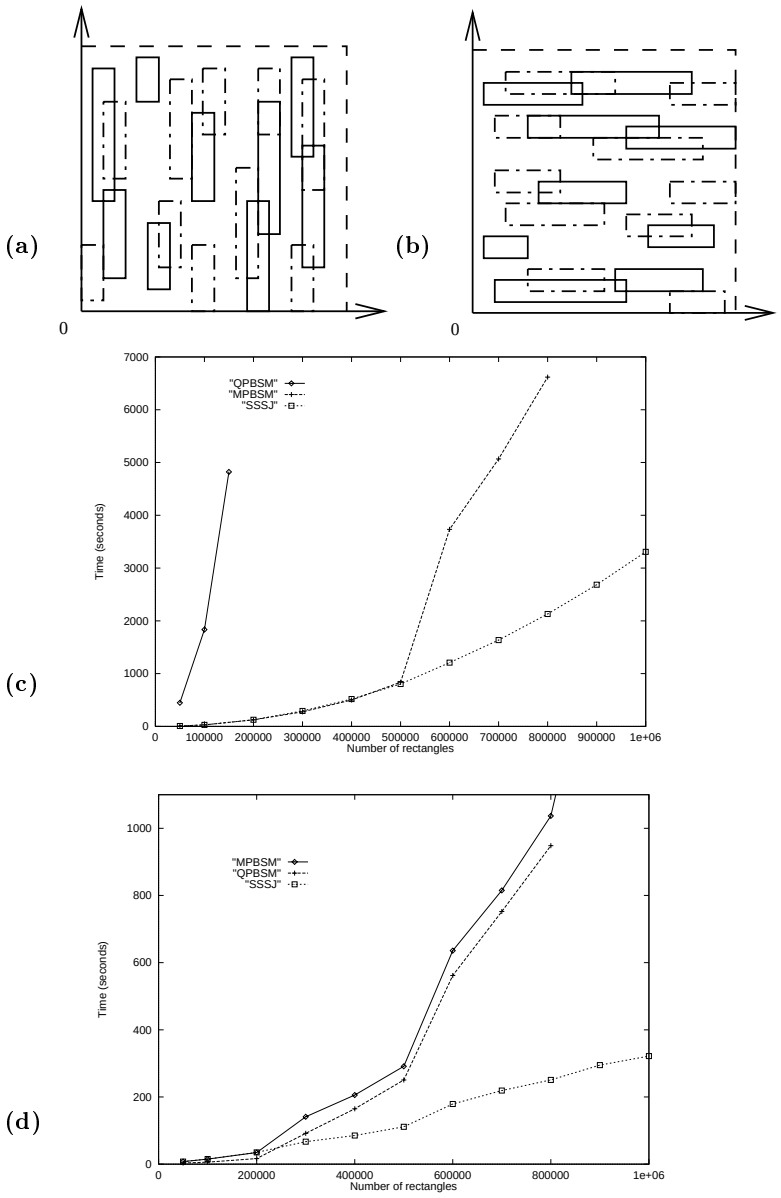

Fig. 7. Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM) (a) Data set 1 consists of tall and skinny (vertically aligned) rectangles. (b) Data set 2 consists of short and wide (horizontally aligned) rectangles. (c) Running times on data set 1. (d) Running times on data set 2.

$\Theta(V)$ in size, yet it preserves the necessary information needed for the next merge step. The same approach works for connectivity.

Graph Problem	I/O Bound, $D = 1$
List ranking, Euler tour of a tree, Centroid decomposition, Expression tree evaluation	$\Theta\big(Sort(V)\big)$ [33]
Connected components, Minimum spanning forest (MSF)	$O\left(\max\left\{1,\ \log\log\frac{V}{e}\right\}\frac{E}{V}Sort(V)\right)$ [13, 55, 95] (deterministic) $\Theta\left(\frac{E}{V}Sort(V)\right)$ [33] (randomized)
Bottleneck MSF, Biconnected components	$O\left(\min\left\{V^2,\ \max\left\{1,\ \log\frac{V}{M}\right\}\frac{E}{V}Sort(V),\ (\log B)\frac{E}{V}Sort(V) + e\log V\right\}\right)$ [1, 33, 55, 85] (deterministic) $\Theta\left(\frac{E}{V}Sort(V)\right)$ [33, 55] (randomized)
Ear decomposition, Maximal matching	$O\left(\min\left\{V^2,\ \max\left\{1,\ \log\frac{V}{M}\right\}Sort(E),\ (\log B)Sort(E) + e\log V\right\}\right)$ [1, 33, 85] (deterministic) $O\big(Sort(E)\big)$ [33] (randomized)
Undirected breadth-first search	$O\big(BundleSort(E, V) + V\big)$ [95]
Undirected single-source shortest paths	$O(e\log e + V)$ [85]
Directed and undirected depth-first search, Topological sorting, Directed breadth-first search, Directed single-source shortest paths	$O\left(\min\left\{\frac{ve}{m} + V,\ (V + e)\log v\right\}\right)$ [29, 33, 85]
Transitive closure	$O\left(Vv\sqrt{\frac{e}{m}}\right)$ [33]

Table 4. Best known I/O bounds for batched graph problems for the single-disk case $D = 1$. The number of vertices is denoted by $V = vB$ and the number of edges by $E = eB$. The terms $Sort(N)$ and $BundleSort(N, K)$ are defined in Sections 3 and 5.5. Lower bounds are discussed in Section 10.

In the case of *semi-external graph problems* [1], in which the vertices fit in internal memory but not the edges (i.e., $V \leq M < E$), several of the problems in Table 4

can be solved optimally in external memory. For example, finding connected components, biconnected components, and minimum spanning forests can be done in $O(e)$ I/Os when $V \leq M$. The I/O complexities of several problems in the general case remain open, including connected components, biconnected components, and minimum spanning forests in the deterministic case, as well as breadth-first search, topological sorting, shortest paths, depth-first search, and transitive closure. It may be that the I/O complexity for several of these problems is $\Theta\big((E/V)\mathit{Sort}(V) + V\big)$. For special cases, such as trees, planar graphs, outerplanar graphs, and graphs of bounded tree width, several of these problems can be solved substantially faster in $O\big(\mathit{Sort}(E)\big)$ I/Os [4, 33, 90, 91].

Chiang et al. [33] exploit the key idea that efficient EM algorithms can often be developed by a sequential simulation of a parallel algorithm for the same problem. The intuition is that each step of a parallel algorithm specifies several operations and the data upon which they act. If we bring together the data arguments for each operation, which we can do by two applications of sorting, then the operations can be performed by a single linear scan through the data. After each simulation step, we sort again in order to reblock the data into the linear order required for the next simulation step. In list ranking, which is used as a subroutine in the solution of several other graph problems, the number of working processors in the parallel algorithm decreases geometrically with time, so the number of I/Os for the entire simulation is proportional to the number of I/Os used in the first phase of the simulation, which is $\mathit{Sort}(N) = \Theta(n \log_m n)$. The optimality of the EM algorithm given in [33] for list ranking assumes that $\sqrt{m} \log m = \Omega(\log n)$, which is usually true in practice. That assumption can be removed by use of the buffer tree data structure [10] (see Chapter ???). A practical randomized implementation of list ranking appears in [126].

Dehne et al. [46, 47] and Sibeyn and Kaufmann [128] use a related approach and get efficient I/O bounds by simulating coarse-grained parallel algorithms in the BSP parallel model. Coarse-grained parallel algorithms may exhibit more locality than the fine-grained algorithms considered in [33], and as a result the simulation may require fewer sorting steps. Dehne et al. make certain assumptions, most notably that $\log_m n \leq c$ for some small constant c (or equivalently that $M^c < NB$), so that the periodic sortings can each be done in a linear number of I/Os. Since the BSP literature is well developed, their simulation technique provides efficient single-processor and multiprocessor EM algorithms for a wide variety of problems.

In order for the simulation techniques to be reasonably efficient, the parallel algorithm being simulated must run in $O\big((\log N)^c\big)$ time using N processors. Unfortunately, the best known polylog-time algorithms for problems such as depth-first search and shortest paths use a polynomial number of processors, not a linear number. P-complete problems such as lexicographically-first depth-first search are unlikely to have polylogarithmic time algorithms even with a polynomial number of processors. The interesting connection between the parallel domain and the EM domain suggests that there may be relationships between computational complexity classes related to parallel computing (such as P-complete problems) and those related to I/O efficiency. It may thus be possible to show by reduction that certain groups of problems are "equally hard" to solve efficiently in terms of I/O and are thus unlikely to have solutions as fast as sorting.

9 The TPIE External Memory Programming Environment

In this section we describe TPIE (Transparent Parallel I/O Environment)[3] [14, 135, 138], which serves as the implementation platform for the experiments described in Section 7 as well as in several of the referenced papers. TPIE is a comprehensive set of C++ templates for EM paradigms and a run-time library. Its goal is to help programmers develop high-level, portable, and efficient implementations of EM algorithms.

There are three basic approaches to supporting development of I/O-efficient code, which we call *access-oriented*, *array-oriented*, and *framework-oriented*. TPIE falls primarily into the third category with some elements of the first category. Access-oriented systems preserve the programmer abstraction of explicitly requesting data transfer. They often extend the read-write interface to include data type specifications and collective specification of multiple transfers, sometimes involving the memories of multiple processing nodes. Examples of access-oriented systems include the UNIX file system at the lowest level, higher-level parallel file systems such as Whiptail [125], Vesta [37], PIOUS [94], and the High Performance Storage System [156], and I/O libraries MPI-IO [36] and LEDA-SM [43].

Array-oriented systems access data stored in external memory primarily by means of compiler-recognized data types (typically arrays) and operations on those data types. The external computation is directly specified via iterative loops or explicitly data-parallel operations, and the system manages the explicit I/O transfers. Array-oriented systems are effective for scientific computations that make regular strides through arrays of data and can deliver high-performance parallel I/O in applications such as computational fluid dynamics, molecular dynamics, and weapon system design and simulation. Array-oriented systems are generally ill-suited to irregular or combinatorial computations. Examples of array-oriented systems include PASSION [132], Panda [120] (which also has aspects of access orientation), PI/OT [102], and ViC* [35].

TPIE [14, 135, 138] provides a framework-oriented interface for batched computation as well as an access-oriented interface for online computation. Instead of viewing batched computation as an enterprise in which code reads data, operates on it, and writes results, a framework-oriented system views computation as a continuous process during which a program is fed streams of data from an outside source and leaves trails of results behind. TPIE programmers do not need to worry about making explicit calls to I/O routines. Instead, they merely specify the functional details of the desired computation, and TPIE automatically choreographs a sequence of data movements to feed the computation.

TPIE is written in C++ as a set of templated classes and functions. It consists of three main components: a block transfer engine (BTE), a memory manager (MM), and an access method interface (AMI). The BTE is responsible for moving blocks of data to and from the disk. It is also responsible for scheduling asynchronous read-ahead and write-behind when necessary to allow computation and I/O to overlap. The MM is responsible for managing main memory in coordination with the AMI and BTE. The AMI provides the high-level uniform interface for application programs. The AMI is the only component that programmers normally need to interact with directly. Applications that use the AMI are portable across hardware platforms, since they do not have to deal with the underlying details of how I/O is performed on a particular machine.

[3] The TPIE software distribution is available free of charge at `http://www.cs.duke.edu/TPIE/` on the World Wide Web.

We have seen in the previous sections that many batched problems in spatial databases, GIS, scientific computing, and graphs can be solved optimally using a relatively small number of basic paradigms such as scanning (or streaming), multiway distribution, and merging, which TPIE supports as access mechanisms. Batched programs in TPIE thus consist primarily of a call to one or more of these standard access mechanisms. For example, a distribution sort can be programmed by using the access mechanism for multiway distribution. The programmer has to specify the details as to how the partitioning elements are formed and how the buckets are defined. Then the multiway distribution is invoked, during which TPIE automatically forms the buckets and writes them to disk using double buffering. For online data structures such as hashing, B-trees, and R-trees, TPIE supports more traditional block access like the access-oriented systems.

10 Lower Bounds on I/O

In this section we prove the lower bounds from Theorems 1–5 and mention some related I/O lower bounds for the batched problems in computational geometry and graphs covered in Sections 7 and 8.

10.1 Scanning

The most trivial batched problem is that of scanning (a.k.a. streaming or touching) a file of N data items, which can be done in a linear number $O(N/DB) = O(n/D)$ of I/Os.

10.2 Permuting

Permuting is one of several simple problems that can be done in linear CPU time in the (internal memory) RAM model, but require a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter B. The following proof of the permutation lower bound (7) of Theorem 2 appears in [75].

Theorem 9. *Assuming that M/B is an increasing function as $N \to \infty$, the number of I/Os required to sort or permute n items, up to lower-order terms, is at least*

$$\frac{2N}{D} \frac{\log(N/B)}{B \log(M/B) + 2 \log N} \sim \begin{cases} \frac{2N}{DB} \frac{\log(N/B)}{\log(M/B)} & \text{if } B \log \frac{M}{B} = \omega(\log N); \\ \frac{N}{D} & \text{if } B \log \frac{M}{B} = o(\log N). \end{cases}$$

The second case in the theorem is the pathological case in which the block size B and internal memory size M are so small that the optimal way to permute the items is to move them one at a time in the naive manner, not making use of blocking.

For the lower bound calculation, we can assume without loss of generality that there is only one disk, namely, $D = 1$. The I/O lower bound for general D follows by dividing the lower bound for one disk by a factor of D. We can also assume at any given time that there is only one copy of each block, either on disk or in memory.

The lower bound proof is an elaboration of the one by Aggarwal and Vitter [8], in which they bound the maximum number of permutations that can be produced by at most t I/Os. If we take the value of t for which the bound first reaches $N!$, we get a lower bound on the worst-case number of I/Os. Our elaboration is more careful to provide a precise bound on both inputs and output operations, so that constant factors are provided.

Initially, the number of producible permutations is 1. Let us consider the effect of an output. There can be at most $N/B + o - 1$ nonempty blocks before the oth output operation, and thus the items in the oth output can go into one of $N/B + o$ places relative to the other blocks. Hence, the oth output boosts the number of producible permutations by a factor of at most $N/B + o$, which can be bounded trivially by

$$N(1 + \log N) \tag{11}$$

For the case of an input operation, we first consider a read I/O from a specific block on disk. If the b items involved in the read I/O were together in internal memory at some previous time (e.g., if the block was created by an earlier output operation), then the items could have been arranged in an arbitrary order by the algorithm while in internal memory. Thus, the $b!$ possible ordering of the b inputed items relative to themselves could already have been produced before the input operation. This implies in a subtle way that rearranging the newly inputed items among the other $M - b$ items in internal memory can boost the number of producible permutations by a factor of at most $\binom{M}{b}$, which is the number of ways to intersperse b indistinguishable items within a group of size M.

The above analysis applies to input from a specific block. If the input was preceded by a total of o output operations, there are at most $N/B + o \leq N(1 + \log N)$ blocks to choose from for the I/O, so the number of producible permutations is boosted further by at most $N(1 + \log N)$. Therefore, assuming that at some point the b inputed items were previously together in internal memory, an input operation can boost the number of producible permutations by at most

$$N(1 + \log N)\binom{M}{b}. \tag{12}$$

Now let's consider an input operation in which some of the inputed items were not together previously in internal memory (e.g., the first time a block is read). By rearranging the relative order of the items in internal memory, we can increase the number of producible permutations. Given that there are N/B full blocks initially, we get the maximum increase when the N/B blocks are read in full, which boosts the number of producible permutations by a factor of

$$(B!)^{N/B}. \tag{13}$$

Let I be the total number of input operations. In the ith input operation, let b_i be the number of items brought into internal memory. By the simplicity property, some of the items in the block being accessed may not be brought into internal memory, but rather may be left on disk. In this case, b_i counts only the number of items that are removed from disk and left in internal memory. In particular, we have $0 \leq b_i \leq B$.

By the simplicity property, we need to make room in internal memory for the new items arriving, and in the end all items are stored back on disk. Therefore we get the following lower bound on the number O of output operations:

$$O \geq \frac{1}{B}\Big(\sum_{1 \leq i \leq I} b_i\Big). \tag{14}$$

Combining (11), (12), and (13), we find that

$$\big(N(1 + \log N)\big)^{I+O} \prod_{1 \leq i \leq I} \binom{M}{b_i} \geq \frac{N!}{(B!)^{N/B}}, \tag{15}$$

where O satisfies (14).

Let $\bar{B}$ be the average number of items read during the I input operations. By a convexity argument, the left-hand side of (15) is maximized when each b_i has the same value, namely, $\bar{B}$. From (15) and (14), we get

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{\bar{B}}^{I} \geq \frac{N!}{(B!)^{N/B}}; \tag{16}$$

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{\bar{B}}^{(I+O)/(1+\bar{B}/B)} \geq \frac{N!}{(B!)^{N/B}}. \tag{17}$$

The left-hand side of (17) maximized when $\bar{B} = B$, so we get

$$\big(N(1+\log N)\big)^{I+O}\binom{M}{B}^{(I+O)/2} \geq \frac{N!}{(B!)^{N/B}}. \tag{18}$$

Theorem 9 follows by taking logarithms of both sides of (18) and using Stirling's formula and the fact that M/B is an increasing function of N.

10.3 Sorting

Permuting is a special case of sorting, and hence, the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and we must resort to the comparison model to get the full lower bound (6) of Theorem 1 [8]. In the typical case in which $B \log m = \Omega(\log n)$, the comparison model is not needed to prove the sorting lower bound; the difficulty of sorting in that case arises not from determining the order of the data but from permuting (or routing) the data.

10.4 Related Problems

The proof used above for permuting also works for permutation networks, in which the communication pattern is oblivious (fixed). Since the choice of disk block is fixed for each I/O, there is no $\big(N(1+\log N)\big)^{I+O}$ term as there is in (15). Hence, when we solve for $I + O$, we get the lower bound (6) rather than (7). The lower bound follows directly from the counting argument; unlike the sorting derivation, it does not require the comparison model for the case $B \log m = o(\log n)$. The lower bound also applies directly to FFT, since permutation networks can be formed from three FFTs in sequence. The transposition lower bound involves a potential argument based upon a togetherness relation [8].

Arge et al. [15] show for the comparison model that any problem with an $\Omega(N \log N)$ lower bound in the (internal memory) RAM model requires $\Omega(n \log_m n)$ I/Os in PDM for a single disk. Their argument leads to a matching lower bound of $\Omega\big(n \max\{1, \log_m(K/B)\}\big)$ I/Os in the comparison model for duplicate removal with one disk.

10.5 Bundle Sorting

For the problem of bundle sorting, in which the N items have a total of K distinct key values (but the secondary information of each item is different), Matias et al. [92] derive the matching lower bound $\mathit{BundleSort}(N, K) = \Omega\big(n \max\{1, \log_m \min\{K, n\}\}\big)$. The proof consists of the following parts. The first part is a simple proof of the

same lower bound as for duplicate removal, but without resorting to the comparison model (except for the pathological case $B \log m = o(\log n)$). It suffices to replace $N!$ on the right-hand side of (15) by $N!/\big((N/K)!\big)^K$, which is the maximum number of permutations of N numbers having K distinct values. Solving for $I + O$ gives the lower bound $\Omega\big(n \max\{1, \log_m(K/B)\}\big)$, which is equal to the desired lower bound for $BundleSort(N, K)$ when $K = B^{1+\Omega(1)}$ or $M = B^{1+\Omega(1)}$. Matias et al. [92] derive the remaining case of the lower bound for $BundleSort(N, K)$ by a potential argument based upon the transposition lower bound. Dividing by D gives the lower bound for D disks.

10.6 Geometrical and Graph Problems

Chiang et al. [33], Arge [11], Arge and Miltersen [16], and Munagala and Ranade [95] give models and lower bound reductions for several computational geometry and graph problems. The geometry problems discussed in Section 7 are equivalent to sorting in both the internal memory and PDM models. Problems such as list ranking and expression tree evaluation have the same nonlinear I/O lower bound as permuting. Other problems such as connected components, biconnected components, and minimum spanning forest of sparse graphs with E edges and V vertices require as many I/Os as E/V instances of permuting V items. This situation is in contrast with the (internal memory) RAM model, in which the same problems can all be done in linear CPU time. (The known linear-time RAM algorithm for finding a minimum spanning tree is randomized.) In some cases there is a gap between the best known upper and lower bounds, which we examine further in Section 8.

10.7 Issue of Indivisibility

The lower bounds mentioned above assume that the data items are in some sense "indivisible", in that they are not split up and reassembled in some magic way to get the desired output. It is conjectured that the sorting lower bound (6) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, Adler [3] showed that removing the indivisibility assumption can lead to faster algorithms. A similar result is shown by Arge and Miltersen [16] for the decision problem of determining if N data item values are distinct. Whether the conjecture is true is a challenging theoretical open problem.

11 Dynamic Memory Allocation

The amount of internal memory allocated to a program may fluctuate during the course of execution because of demands placed on the system by other users and processes. EM algorithms must be able to adapt dynamically to whatever resources are available so as to preserve good performance [100]. The algorithms in the previous sections assume a fixed memory allocation; they must resort to virtual memory if the memory allocation is reduced, often causing a severe degradation in performance.

Barve and Vitter [26] discuss the design and analysis of EM algorithms that adapt gracefully to changing memory allocations. In their model, without loss of generality, an algorithm (or program) $\mathcal{P}$ is allocated internal memory in phases: During the ith phase, $\mathcal{P}$ is allocated m_i blocks of internal memory, and this memory remains allocated to $\mathcal{P}$ until $\mathcal{P}$ completes $2m_i$ I/O operations, at which point the next phase begins. The process continues until $\mathcal{P}$ finishes execution. The model makes the reasonable assumption that the duration for each memory allocation phase is long enough to allow all the memory in that phase to be used by the algorithm.

For sorting, the lower bound approach of Section 10.2 implies that

$$\sum_i 2m_i \log m_i = \Omega(n \log n).$$

We say that $\mathcal{P}$ is *dynamically optimal* for sorting if

$$\sum_i 2m_i \log m_i = O(n \log n)$$

for all possible sequences m_1, m_2, ... of memory allocation. Intuitively, if $\mathcal{P}$ is dynamically optimal, no other algorithm can perform more than a constant number of sorts in the worst-case for the same sequence of memory allocations.

Barve and Vitter [26] define the model in generality and give dynamically optimal strategies for sorting, matrix multiplication, and buffer tree operations. Their work represents the first theoretical model of dynamic allocation and the first algorithms that can be considered dynamically optimal. Previous work was done on memory-adaptive algorithms for merge sort [100, 160] and hash join [101], but the algorithms handle only special cases and can be made to perform nonoptimally for certain patterns of memory allocation.

12 Conclusions

In this survey we have described several useful paradigms for the design and implementation of efficient external memory (EM) algorithms. The problem domains we have considered include sorting, permuting, FFT, scientific computing, computational geometry, graphs, databases, and geographic information systems. Interesting challenges remain in virtually all these problem domains. One difficult problem is to prove lower bounds for permuting and sorting without the indivisibility assumption. Another promising area is the design and analysis of EM algorithms for efficient use of multiple disks. Optimal bounds have not yet been determined for several basic EM graph problems such as topological sorting, shortest paths, breadth- and depth-first search, and connected components. There is an intriguing connection between problems that have good I/O speedups and problems that have fast and work-efficient parallel algorithms.

A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice. For some of the problems that can be solved optimally up to a constant factor, the constant overhead is too large for the algorithm to be of practical use, and simpler approaches are needed. In practice, algorithms cannot assume a static internal memory allocation; they must adapt in a robust way when the memory allocation changes.

Many interesting challenges and opportunities in algorithm design and analysis arise from new architectures being developed, such as networks of workstations, hierarchical storage devices, disk drives with processing capabilities, and storage devices based upon microelectromechanical systems (MEMS). Active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have recently been proposed to further reduce the I/O bottleneck, especially in large database applications [2, 108]. MEMS-based nonvolatile storage has the potential to serve as an intermediate level in the memory hierarchy between DRAM and disks. It could ultimately provide better latency and bandwidth than disks, at less cost per bit than DRAM [119, 139].

Acknowledgments. The author wishes to thank Pankaj Agarwal, Lars Arge, Ricardo Baeza-Yates, Adam Buchsbaum, Jeff Chase, David Hutchinson, Vasilis Samoladas, Amin Vahdat, the members of the Center for Geometric Computing at Duke University, and the referees for several helpful comments and suggestions. Figure 1 is a modified version of a figure by Darren Vengroff, Figure 2 comes from [38], and Figure 7 is a modified version of a figure in [18].

References

1. J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. In *Proceedings of the European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, 332–343, Venice, Italy, August 1998. Springer-Verlag.
2. A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11), 81–91, November 1998.
3. M. Adler. New coding techniques for improved bandwidth utilization. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 37, 173–182, Burlington, VT, October 1996.
4. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, 117–126, 1998.
5. A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 19, 305–314, New York, 1987.
6. A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 28, 204–216, Los Angeles, 1987.
7. A. Aggarwal and C. G. Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 5, 659–668, 1994.
8. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1116–1127, 1988.
9. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 72–109, 1994.
10. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, 334–345. Springer-Verlag, 1995. A complete version appears as BRICS technical report RS–96–28, University of Aarhus.
11. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 1004 of *Lecture Notes in Computer Science*, 82–91. Springer-Verlag, 1995.
12. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*, 213–254. Springer-Verlag, 1997.
13. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, volume 1851 of *Lecture Notes in Computer Science*, July 2000.
14. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Workshop on Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, 328–348, Baltimore, January 1999. Springer-Verlag.
15. L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, 83–94. Springer-Verlag, 1993.
16. L. Arge and P. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 139–159. American Mathematical Society Press, Providence, RI, 1999.
17. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proceedings of the International Conference on Extending Database Technology*, volume 7, Konstanz, Germany, March 2000.
18. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the International Conference on Very Large Databases*, volume 24, 570–581, New York, August 1998.
19. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, 685–694, 1998.
20. L. Arge, V. Samoladas, and J. S. Vitter. Two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 18, 346–357, Philadelphia, PA, May–June 1999.

21. L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear. Special issue on cartography and geographic information systems. An earlier version appeared in *Proceedings of the Third European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, 295–310, Springer-Verlag, September 1995.
22. C. Armen. Bounds on the separation of two parallel disk models. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, volume 4, 122–127, Philadelphia, May 1996.
23. R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 601–631, 1997.
24. R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive analysis of buffer management algorithms. *Journal of Algorithms*, 36, August 2000.
25. R. D. Barve, E. A. M. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. In *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 83–92, Atlanta, GA, May 1999.
26. R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 40, 273–284, New York, October 1999.
27. R. D. Barve and J. S. Vitter. A simple and efficient parallel disk mergesort. *ACM Trans. Comput. Syst.*, 35(2), 189–215, March/April 2002.
28. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 41, Redondo Beach, California, November 12–14 2000.
29. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, January 2000.
30. L. Carter and K. S. Gatlin. Towards an optimal bit-reversal permutation program. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 39, 544–553, Palo Alto, November 1998.
31. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, June 1994.
32. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 8(4), 211–236, 1998.
33. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 6, 139–149, January 1995.
34. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4, 387–421, 1989.
35. A. Colvin and T. H. Cormen. ViC*: A compiler for virtual-memory c*. In *Proceedings of the International Workshop on High-Level Programming Models and Supportive Environments*, volume 3, 1998.
36. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 5, 127–146. Kluwer Academic Publishers, 1996.
37. P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3), 225–264, August 1996.
38. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.
39. T. H. Cormen and D. M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1), 5–20, January 1998.
40. T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1), 105–136, 1999.
41. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for geometric problems. In *Proceedings of the ACM Symposium on Computational Geometry*, volume 14, 259–268, June 1998.
42. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. I/O-optimal computation of segment intersections. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 131–138. American Mathematical Society Press, Providence, RI, 1999.
43. A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In J. S. Vitter and C. Zaroliagis, editors, *Proceedings of the European Symposium on Algorithms*, volume 1643 of *Lecture Notes in Computer Science*, 228–242, London, July 1999. Springer-Verlag.
44. R. Cypher and G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47(3), 501–548, 1993.

45. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
46. F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 9, 106–115, June 1997.
47. F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the International Parallel Processing Symmposium*, volume 13, 14–20, April 1999.
48. H. B. Demuth. *Electronic Data Sorting*. Ph.d., Stanford University, 1956. A shortened version appears in *IEEE Transactions on Computing*, C-34(4), 296–310, April 1985, special issue on sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, editors.
49. P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6, 64–84, 1980.
50. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, volume 1, 280–291, December 1991.
51. W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 10, 98–107, 1998.
52. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38, 86–124, 1989.
53. R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2), 85–113, June 1988.
54. NASA's Earth Observing System (EOS) web page, NASA Goddard Space Flight Center, `http://eospso.gsfc.nasa.gov/`.
55. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—A technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5), 669–96, 1997.
56. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 39, 174–183, Palo Alto, November 1998.
57. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 40, 501–511, New York, October 1999.
58. W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, 3rd edition, 1968.
59. P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2), 236–280, March 1999.
60. R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, 105–109. Plenum, 1972.
61. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 40, 1999.
62. T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 11–20, Boston, March 1992.
63. G. R. Ganger. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the Computer Measurement Group Conference*, volume 21, 1263–1269, December 1995.
64. M. Gardner. *Magic Show*, chapter 7. Knopf, New York, 1977.
65. G. A. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4), 779–793, December 1996.
66. R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proceedings of the International Conference on Very Large Databases*, volume 24, 26–37, August 1998.
67. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 34, 714–723, Palo Alto, November 1993.
68. J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. In *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.
69. R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 87–106. American Mathematical Society Press, Providence, RI, 1999.
70. S. K. S. Gupta, Z. Li, and J. H. Reif. Generating efficient programs for two-level memories from tensor-products. In *Proceedings of the IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, volume 7, 510–513, Washington, D.C., October 1995.
71. L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2–3), 182–208, 1994.

72. M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 107–118. American Mathematical Society Press, Providence, RI, 1999.
73. J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 13, 326–333, May 1981.
74. D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proceedings of the International Conference on Computing and Combinatorics*, volume 1627 of *Lecture Notes in Computer Science*, 51–60. Springer-Verlag, July 1999.
75. D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. submitted to journal. An earlier version appeared in *Proceedings of the European Symposium on Algorithms*, Springer-Verlag, Lecture Notes in Computer Science, *2161*, August 2001.
76. D. A. Hutchinson, P. Sanders, and J. S. Vitter. The power of duality for prefetching and sorting with parallel disks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 2076 of *Lecture Notes in Computer Science*, Crete, Greece, July 2001. Springer-Verlag.
77. M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, volume 6, 68–77, Atlanta, GA, May 1999. ACM Press.
78. M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel i/o systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 13, Crete, Greece, July 2001.
79. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the ACM Conference on Principles of Database Systems*, volume 9, 299–313, 1990.
80. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3), 589–612, 1996.
81. M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, 35(11), 978–988, November 1986.
82. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15, 287–299, 1986.
83. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1998.
84. D. E. Knuth. *MMIXware*. Springer, Berlin, 1999.
85. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, volume 8, 169–176, October 1996.
86. R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.
87. F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4), 344–354, April 1985. Special issue on sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, editors.
88. C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 34, 704–713, 1993.
89. Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 8, 35–59, 1996.
90. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the International Conference on Computing and Combinatorics*, volume 1627 of *Lecture Notes in Computer Science*, 51–60. Springer-Verlag, July 1999.
91. A. Maheshwari and N. Zeh. I/O-efficient algorithms for bounded treewidth graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, Washington, D.C., January 2001.
92. Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, 839–848, San Francisco, January 2000.
93. U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, Washington, D.C., January 2001.
94. S. A. Moyer and V. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. *Journal of Parallel and Distributed Computing*, 38(1), 81–91, October 1996.
95. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 10, 687–694, Baltimore, MD, January 1999.
96. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2), 181–214, August 1996.
97. M. H. Nodine, D. P. Lopresti, and J. S. Vitter. I/O overhead and parallel VLSI architectures for lattice computations. *IEEE Transactions on Communications*, 40(7), 843–852, July 1991.

98. M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 5, 120–129, Velen, Germany, June–July 1993.
99. M. H. Nodine and J. S. Vitter. Greed Sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4), 919–933, July 1995.
100. H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorts. In *Proceedings of the International Conference on Very Large Databases*, volume 19, 618–629, Dublin, 1993.
101. H. Pang, M. J. Carey, and M. Livny. Partially preemptive hash joins. In P. Buneman and S. Jajodia, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 59–68, Washington, D.C., May 1993.
102. I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron. PI/OT: Parallel I/O templates. *Parallel Computing*, 23(4), 543–570, June 1997.
103. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 259–270, June 1996.
104. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, Berlin, 1985.
105. N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. In *Workshop on Algorithm Engineering and Experimentation*, Lecture Notes in Computer Science. Springer-Verlag, January 2000.
106. J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In M. Atkinson et al., editors, *Proceedings of the International Conference on Very Large Databases*, volume 25, 78–89, Los Altos, CA, 1999. Morgan Kaufmann Publishers.
107. J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 475–486, Dallas, Texas, 2000.
108. E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the International Conference on Very Large Databases*, volume 22, 62–73, August 1998.
109. M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1), 78–103, January 1997.
110. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 17–28, March 1994.
111. K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of IEEE International Conference on Data Engineering*, volume 2, 336–242, Los Angeles, 1986.
112. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1989.
113. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.
114. P. Sanders. Personal communication, 2000.
115. P. Sanders. Reconciling simplicity and realism in parallel disk models. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, Washington, January 2001.
116. P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, 849–858, San Francisco, January 2000.
117. J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In *Proceedings of the International Conference on Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, 270–281. Springer-Verlag, August 1995.
118. J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, 117–146. JAI Press, 1987.
119. S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 9, November 2000.
120. K. E. Seamons and M. Winslett. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2), 191–211, 1996.
121. M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the Annual USENIX Technical Conference*, 249–264, New Orleans, 1995.
122. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, San Francisco, January 2000.
123. E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 182–191, June 1998.
124. E. A. M. Shriver and M. H. Nodine. An introduction to parallel I/O models and algorithms. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 2, 31–68. Kluwer Academic Publishers, 1996.
125. E. A. M. Shriver and L. F. Wisniewski. An API for choreographing data accesses. Technical Report PCS-TR95-267, Dept. of Computer Science, Dartmouth College, November 1995.
126. J. F. Sibeyn. From parallel to external list ranking. Technical Report MPI–I–97–1–021, Max-Planck-Institut, September 1997.

127. J. F. Sibeyn. External selection. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, 291–301. Springer-Verlag, 1999.
128. J. F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proceedings of the Italian Conference on Algorithms and Complexity*, volume 3, 229–240, 1997.
129. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6), 196–205, June 1994.
130. R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. *Algorithmica*, 15(2), 154–171, February 1996.
131. Microsoft's TerraServer online database of satellite images, available on the World-Wide Web at `http://terraserver.microsoft.com/`.
132. R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6), 70–78, June 1996.
133. T. (tm). 1992 technical documentation. Technical report, U. S. Bureau of the Census, 1992.
134. S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 161–179. American Mathematical Society Press, Providence, RI, 1999.
135. TPIE user manual and reference, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.
136. J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3, 331–360, 1991.
137. M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors. *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
138. D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of NASA Goddard Conference on Mass Storage Systems*, volume 5, II, 553–570, September 1996.
139. P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The "Millipede"—More than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, 44(3), 323–340, 2000.
140. J. S. Vitter. External memory algorithms. In *Proceedings of the European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, Venice, Italy, August 1998. Springer-Verlag. Invited paper.
141. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.
142. J. S. Vitter. Online data structures in external memory. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, 119–133, Prague, August 1999. Springer-Verlag. Invited paper.
143. J. S. Vitter. Online data structures in external memory. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 1668 of *Lecture Notes in Computer Science*, Vancouver, August 1999. Springer-Verlag. Invited paper.
144. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2), 209–271, June 2001.
145. J. S. Vitter. Algorithms and data structures for external memory. In *The Computer Engineering Handbook*, chapter 32, 32–1–32–33. CRC Press and IEEE Press, 2002.
146. J. S. Vitter. External memory algorithms. In *Handbook of Massive Data Sets*, 359–418. Kluwer Academic Publishers, 2002.
147. J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 9, 431–524. North-Holland, 1990.
148. J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, Washington, January 2001.
149. J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17, 107–114, 1993.
150. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3), 110–147, 1994.
151. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3), 148–169, 1994.
152. J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 193–204, Philadelphia, PA, June 1999.
153. J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the International ACM Conference on Information and Knowledge Management*, volume 7, 96–104, Washington, November 1998.
154. M. Wang, B. Iyer, and J. S. Vitter. Scalable mining for classification rules in relational databases. In *Proceedings of the International Database Engineering & Application Symposium*, 58–67, Cardiff, Wales, July 1998.

155. M. Wang, J. S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proceedings of the International Symposium on Spatial and Temporal Databases*, volume 7, Redondo Beach, CA, July 2001.
156. R. W. Watson and R. A. Coyne. The parallel I/O architecture of the high-performance storage system(HPSS). In *Proceedings of the IEEE Symposium on Mass Storage Systems*, volume 14, 27–44, September 1995.
157. D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the DAGS Symposium on Parallel Computation*, volume 2, 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.
158. C. Wu and T. Feng. The universality of the shuffle-exchange network. *IEEE Transactions on Computers*, C-30, 324–332, May 1981.
159. S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kauffman, 1990.
160. W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. In *Proceedings of the International Conference on Very Large Databases*, volume 23, 376–385, Athens, Greece, 1997.
161. B. Zhu. Further computational geometry in secondary memory. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 834 of *Lecture Notes in Computer Science*, 514 ff. Springer-Verlag, 1994.