Foundations and Trends® in
sample
Vol. xx, No xx (xxxx) 1–160

DOI: xxxxxx

now
the essence of knowledge

Algorithms and Data Structures for External Memory

Jeffrey Scott Vitter[1]

[1] *Department of Computer Science, Purdue University, West Lafayette, IN 47907–2107, U.S.A., 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. In this work we survey the state of the art in the design and analysis of algorithms and data structures for *external memory* (or *EM* for short), where the goal is to exploit locality and parallelism in order to reduce the I/O costs. We consider a variety of EM paradigms for solving batched and online problems efficiently in external memory.

For the batched problem of sorting and related problems like permuting and fast Fourier transform, the key paradigms include distribution and merging. The paradigm of disk striping offers an elegant way to use multiple disks in parallel. For sorting, however, disk striping can be nonoptimal with respect to I/O, so to gain further improvements we discuss distribution and merging techniques for using the disks independently. We also consider useful techniques for batched EM problems involving matrices, geometric data, and graphs.

In the online domain, canonical EM applications include dictionary lookup and range searching. The two important classes of indexed data structures are based upon extendible hashing and B-trees. The paradigms of filtering and bootstrapping provide convenient means in online data structures to make effective use of the data accessed from disk. We also reexamine some of the above EM problems in slightly different settings, such as when the data items are moving, when the data items are variable-length such as character strings, when the data structure is compressed to save space, or when the allocated amount of internal memory can change dynamically.

Programming tools and environments are available for simplifying the EM programming task. We report on some experiments in the domain of spatial databases using the TPIE system (Transparent Parallel I/O programming Environment). The newly developed EM algorithms and data structures that incorporate the paradigms we discuss are significantly faster than other methods used in practice.

1

Introduction

The world is drowning in data! In recent years, we have been deluged by a torrent of data from a variety of increasingly data-intensive applications, including databases, scientific computations, graphics, entertainment, multimedia, web applications, and email. 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 [143]. A petabyte corresponds roughly to the amount of information in a one billion graphically formatted books. The online databases of satellite images used by Microsoft TerraServer (part of MSN Virtual Earth) [315] and Google Earth [173] are multiple terabytes (10^{12} bytes) in size. Wal-Mart's sales data warehouse contains over a half petabyte (500 terabytes) of data. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.

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 typ-

ical. 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 1.1 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 does not do harm), especially when the data sets are not large. The utility 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 of the memory hierarchy, and accessing the

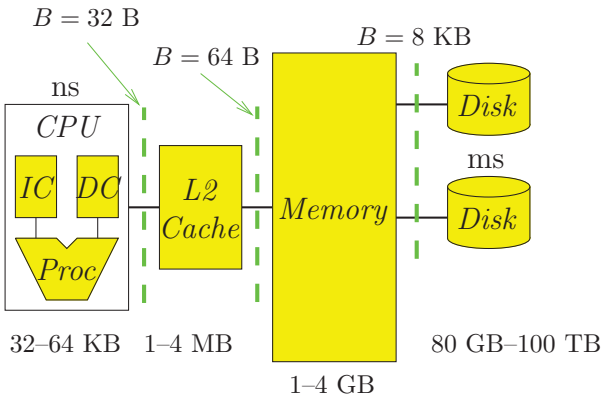

Fig. 1.1 The memory hierarchy of a typical uniprocessor system, including registers, instruction cache, data cache (level 1 cache), level 2 cache, internal memory, and disks. Some systems have in addition a level 3 cache, not shown here. Indicated below each memory level is the range of typical sizes 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 Chapter 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.

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 [133]. 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 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 and data structures that explicitly manage data placement and movement as *external memory* (or *EM*) *algorithms and data structures*. Some authors use the terms *I/O algorithms* or *out-of-core algorithms*. We concentrate in this work 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.

1.1 Overview

In this work we survey several paradigms for exploiting locality and thereby reducing I/O costs when solving problems in external memory. The problems we consider fall into two general categories:

(1) *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.
(2) *Online problems*, in which computation is done in response to a continuous series of query operations. A common technique for online problems is to organize the data items via a hierarchical index, so that only a very small portion of the data needs to be examined in response to each query. The data being queried can be either *static*, which can be preprocessed for efficient query processing, or *dynamic*, where the queries are intermixed with updates such as insertions and deletions.

We base our approach upon the *parallel disk model* (PDM) described in the next section. PDM provides an elegant and reasonably accurate model for analyzing the relative performance of EM algorithms and data structures. The three main performance measures of PDM are *the number of (parallel) I/O operations, the disk space usage,* and *the (parallel) CPU time*. For reasons of brevity, we focus on the first two measures. Most of the algorithms we consider are also efficient in terms of CPU time. In Chapter 3 we list four fundamental I/O bounds that pertain to most of the problems considered in this work. In Chapter 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.

Our general goal is to design optimal algorithms and data structures, by which we mean that their performance measures are within

a constant factor of the optimum or best possible.[1] In Chapter 5 we look at the canonical batched EM problem of external sorting and the related problems of permuting and fast Fourier transform. The two important paradigms of distribution and merging—as well as the notion of duality that relates the two—account for all well-known external sorting algorithms. Sorting with a single disk is now well understood, so we concentrate on the more challenging task of using multiple (or parallel) disks, for which disk striping is not optimal. The challenge is to guarantee that the data in each I/O are spread evenly across the disks so that the disks can be used simultaneously. In Chapter 6 we cover the fundamental lower bounds on the number of I/Os needed to perform sorting and related batched problems. In Chapter 7 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 Chapter 5, and hence we restrict ourselves starting in Chapter 8 to the conceptually simpler single-disk case. In Chapter 8 we mention several effective paradigms for batched EM problems in computational geometry. The paradigms include distribution sweep (for spatial join and finding all nearest neighbors), persistent B-trees (for batched point location and visibility), batched filtering (for 3-D convex hulls and batched point location), external fractional cascading (for red-blue line segment intersection), external marriage-before-conquest (for output-sensitive convex hulls), and randomized incremental construction with gradations (for line segment intersections and other geometric problems). In Chapter 9 we look at EM algorithms for combinatorial problems on graphs, such as list ranking, connected components, topological sorting, and finding shortest paths. One technique for constructing I/O-efficient EM algorithms is to simulate parallel algorithms; sorting is used between parallel steps in order to reblock the data for the simulation of the next parallel step.

In Chapters 10–12 we consider data structures in the online setting. The dynamic dictionary operations of insert, delete, and lookup

[1] In this book we generally use the term "optimum" to denote the absolute best possible and the term "optimal" to mean within a constant factor of the optimum.

Paradigm	Reference
Batched dynamic processing	§11.4
Batched filtering	§8
Batched incremental construction	§8
Bootstrapping	§12
Buffer trees	§11.4
B-trees	§11, §12
Compression	§15
Decomposable search	§13.1
Disk striping	§4.2
Distribution	§5.1
Distribution Sweeping	§8
Duality	§5.3
External Hashing	§10
Externalization	§12.3
Fractional Cascading	§8
Filtering	§12
Lazy Updating	§11.4
Load Balancing	§4
Locality	§4.1
Marriage before conquest	§8
Merging	§5.2
Parallel block transfer	§4.2
Parallel simulation	§9
Persistence	§11.1
Random sampling	§5.1
R-trees	§12.2
Scanning (or streaming)	§2.2
Sparsification	§9
Time-forward processing	§11.4

Table 1.1 Paradigms for I/O efficiency discussed in this work.

can be implemented by the well-known method of hashing. In Chapter 10 we examine hashing in external memory, in which extra care must be taken to pack data into blocks and to allow the number of

items to vary dynamically. Lookups can be done generally with only one or two I/Os. Chapter 11 begins with a discussion of B-trees, the most widely used online EM data structure for dictionary operations and one-dimensional range queries. Weight-balanced B-trees provide a uniform mechanism for dynamically rebuilding substructures and are useful for a variety of online data structures. Level-balanced B-trees permit maintenance of parent pointers and support cut and concatenate operations, which are used in reachability queries on monotone subdivisions. The buffer tree is a so-called "batched dynamic" version of the B-tree for efficient implementation of search trees and priority queues in EM sweep line applications. In Chapter 12 we discuss spatial data structures for multidimensional data, especially those that support online range search. Multidimensional extensions of the B-tree, such as the popular R-tree and its variants, use a linear amount of disk space and often perform well in practice, although their worst-case performance is poor. A nonlinear amount of disk space is required to perform 2-D orthogonal range queries efficiently in the worst case, but several important special cases of range searching can be done efficiently using only linear space. A useful paradigm for developing an efficient EM data structure is to "externalize" an efficient data structure designed for internal memory; a key component of how to make the structure I/O-efficient is to "bootstrap" a static EM data structure for small-sized problems into a fully dynamic data structure of arbitrary size. This paradigm provides optimal linear-space EM data structures for several variants of 2-D orthogonal range search.

In Chapter 13 we discuss some additional EM approaches useful for dynamic data structures, and we also investigate kinetic data structures, in which the data items are moving. In Chapter 14 we focus on EM data structures for manipulating and searching text strings. In many applications, especially those that operate on text strings, the data are highly compressible. Chapter 15 discusses ways to develop data structures that are themselves compressed, but still fast to query. In Chapter 16 we discuss EM algorithms that adapt optimally to dynamically changing internal memory allocations.

In Chapter 17 we discuss programming environments and tools that facilitate high-level development of efficient EM algorithms. We focus

on the TPIE system (Transparent Parallel I/O Environment), which we use in the various timing experiments in this work, and STXXL (Standard Template Library for XXL Data Sets). We conclude with some final remarks and observations in the Conclusions.

Table 1.1 lists several of the EM paradigms discussed in this work.

2

Parallel Disk Model (PDM)

External memory (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 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 2.1. 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 1.1.

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

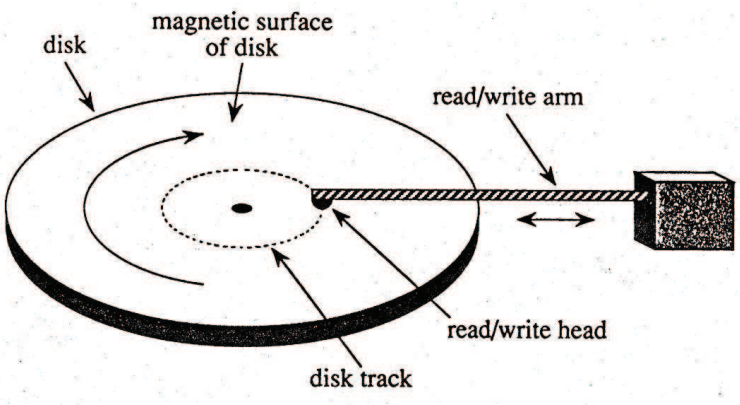

Fig. 2.1 Platter of a magnetic disk drive.

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 [98, 187].

In the next section we describe the high-level parallel disk model (PDM), which we use throughout this work for the design and analysis of EM algorithms and data structures. 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 [335]:

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

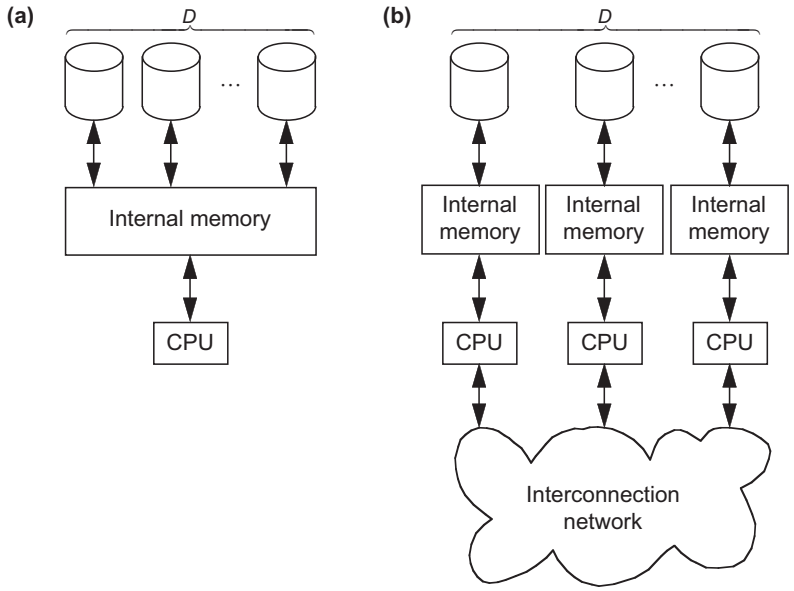

Fig. 2.2 Parallel disk model: (a) $P = 1$, in which the D disks are connected to a common CPU; (b) $P = D$, in which each of the D disks is connected to a separate processor.

where $M < N$, and $1 \leq DB \leq M/2$. The N 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 internal 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 2.2.

Queries are naturally associated with online computations, but they

[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.

can also be done in batched mode. For example, in the batched orthogonal 2-D range searching problem discussed in Chapter 8, 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:

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

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

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad q = \frac{Q}{B}, \qquad z = \frac{Z}{B} \tag{2.1}$$

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

We assume that the data for the problem are initially "striped" across the D disks, in units of blocks, as illustrated in Figure 2.3, and we require the final 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 work 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. There are interesting issues associated with optimizing internal computation time in the presence of multiple disks, in which communication takes place over a particular interconnection network, but they are not the focus of this manuscript. Ideally algorithms and data structures should use linear space, which means $O(N/B) = O(n)$ disk blocks of storage.

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
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. 2.3 Initial data layout on the disks, for $D = 5$ disks and block size $B = 2$. The 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 $\mathcal{D}_3$.

2.2 Practical Modeling Considerations

Track size is a fixed parameter of the disk hardware; for most disks it is in the range 50 KB–2 MB. In reality, the track size for any given disk depends upon the radius of the track (cf. Figure 2.1). Sets of adjacent tracks are usually formatted to have the same 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 1.1. 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 1.1. 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 Chapter 17. More complex and precise disk models, such as the ones by Ruemmler and Wilkes [285], Ganger [165], Shriver et al. [304], Barve et al. [66], Farach-Colton et al. [149], and Khandekar and Pandit [207] 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) [44]. 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 [300].

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 I/O requests for each disk [130].

2.3 Related Models, Hierarchical Memory, and Cache-Oblivious Algorithms

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 [132, 213]. In the early 1970s, Knuth [213] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [159, 213] considered a disk model akin to PDM for $D = 1,\ P = 1,\ B = M/2 = \Theta(N^c)$, for constant $c > 0$, and for those particular parameters, he developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [192] developed a pebbling model of I/O for straightline computations, and Savage and Vitter [296] extended the model to deal with block transfer.

Aggarwal and Vitter [23] 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 simulation technique due to Sanders et al. [294]; 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 [57].

Surveys of I/O models, algorithms, and challenges appear in [3, 30, 169, 247, 305]. Several versions of PDM have been developed for parallel computation [126, 125, 227, 309]. Models of "active disks" augmented with processing capabilities to reduce data traffic to the host, especially during streaming applications, are given in [4, 282]. Models of microelectromechanical systems (MEMS) for mass storage appear in [177].

Some authors have studied problems that can be solved efficiently by making only one pass (or a small number of passes) over the

data [24, 150, 188, 256]. In such *data streaming* applications, one useful 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 problem-dependent and orthogonal to our focus in this work; we refer the reader to the surveys in [24, 256].

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 [20, 21, 334, 336]. Conversely, the algorithms for hierarchical models can be run in the PDM setting.

Frigo et al. [162] introduce the important notion of *cache-oblivious algorithms*, which require no knowledge of the storage parameters, like M and B, nor special programming environments for implementation. It follows that, up to a constant factor, time-optimal and space-optimal algorithms in the cache-oblivious model are similarly optimal in the external memory model. Frigo et al. develop optimal cache-oblivious algorithms for merge sort and distribution sort. Bender et al. [76] and Bender et al. [77] develop cache-oblivious versions of B-trees that offer speed advantages in practice. In recent years, there has been considerable research in the development of efficient cache-oblivious algorithms and data structures for a variety of problems. We refer the reader to [32] for a survey.

The match between theory and practice is harder to establish for hierarchical models and caches than for disks. Generally, the most significant speedups come from optimizing the I/O communication between internal memory and the 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 input 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 [214, 284, 312] can provide sophisticated help in the optimization process.

For reasons of focus, we do not consider hierarchical and cache models in this work. We refer the reader to the previous references on cache-oblivious algorithms as well as the following references: Aggarwal et al. [20] define an elegant hierarchical memory model, and Aggarwal et al. [21] augment it with block transfer capability. Alpern et al. [28] model levels of memory in which the memory size, block size, and bandwidth grow at uniform rates. Vitter and Shriver [336] and Vitter and Nodine [334] discuss parallel versions and variants of the hierarchical models. The parallel model of Li et al. [227] also applies to hierarchical memory. Savage [295] gives a hierarchical pebbling version of [296]. Carter and Gatlin [93] define pebbling models of nonassociative direct-mapped caches. Rahman and Raman [278] and Sen, Chatterjee, and

Dumir [301] apply EM techniques to models of caches and translation lookaside buffers.

3

Fundamental I/O Operations and Bounds

The I/O performance of many algorithms and data structures can be expressed in terms of the bounds for these 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) *Searching* online through N sorted data items.
(4) *Outputting* the Z items of an answer to a query in a blocked "output-sensitive" fashion.

We give the I/O bounds for these four operations in Table 3.1. We single out the special case of a single disk ($D = 1$), since the formulas are simpler and many of the discussions in this work will be restricted to the single-disk case.

We discuss the algorithms and lower bounds for $Sort(N)$ and $Search(N)$ in Chapters 5, 6, 10, and 11. The lower bounds for searching assume the comparison model of computation; searching via hashing can be done in $\Theta(1)$ I/Os on the average.

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

Table 3.1 I/O bounds for the four fundamental operations. The PDM parameters are defined in Section 2.1.

The first two of these I/O bounds—$Scan(N)$ and $Sort(N)$—apply to batched problems. The last two I/O bounds—$Search(N)$ and $Output(Z)$—apply to online problems and are typically combined together into the form $Search(N) + Output(Z)$. As mentioned in Section 2.1, some batched problems also involve queries, in which case the I/O bound $Output(Z)$ may be relevant to them as well. In some pipelined contexts, the Z items in an answer to a query do not need to be output to the disks but rather can be "piped" to another process, in which case there is no I/O cost for output. Relational database queries are often processed in such a pipeline fashion. For simplicity, in this article we explicitly consider the output cost for queries.

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 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.

As Table 3.1 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. For $Search(N)$, the speedup is less significant: The I/O bound $\Theta(\log_B N)$ for $D = 1$ becomes $\Theta(\log_{DB} N)$ for $D \geq 1$; the resulting speedup is only $\Theta\big((\log_B N)/\log_{DB} N\big) = \Theta\big((\log DB)/\log B\big) = \Theta\big(1 + (\log D)/\log B\big)$, which is typically less than 2.

In practice, the logarithmic terms $\log_m n$ in the $Sort(N)$ bound and $\log_{DB} N$ in the $Search(N)$ bound are small constants. 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. In online applications, a smaller B value, such as $B = 10^2$, is more appropriate, as explained in Section 2.2. The corresponding value of $\log_B N$ for the example is 5, so even with a single disk, online search can be done in a relatively small constant number of I/Os.

It still makes sense to explicitly identify terms such as $\log_m n$ and $\log_B N$ 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 problem 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 [336].

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 input I/O operation transfers a block of B items, we make optimal use of that input operation when all B items are needed by the application. A similar remark applies to output I/O 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 it is run on data sets that fit in internal memory, but it will perform miserably when deployed naively in an EM setting and virtual memory is used to handle page management. Examining such performance degradation is a good way to put the I/O bounds of Table 3.1 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* [209, 286], 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 3.1. The only exception is sorting. The optimal multiple-disk algorithms for sorting require more sophisticated load balancing techniques, which we cover in Chapter 5.

4.1 Locality Issues with a Single Disk

A good way to appreciate the fundamental I/O bounds in Table 3.1 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 work, 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. Similarly, in the online setting, many types of search queries, such as range search queries and stabbing queries, can be done using binary trees in $O(\log N + Z)$ query CPU time when the tree fits into internal memory, but the same data structure in an external memory setting may require $\Omega(\log N + Z)$ I/Os per query.

We would like instead to incorporate locality *directly* into the algorithm design and achieve the desired I/O bounds of $O(n \log_m n)$ for the batched problems and $O(\log_B N + z)$ for online search, in line with the fundamental bounds listed in Table 3.1. 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. For online problems, we want the base of the logarithm to be B and to replace Z by z. The relative speedup in I/O performance can be very significant, both theoretically and in practice. For example, for batched problems, the I/O performance improvement can be a factor of $(N \log n)/n \log_m n = B \log m$, which is extremely large. For online problems, the performance improvement

can be a factor of $(\log N + Z)/(\log_B N + z)$; this value is always at least $(\log N)/\log_B N = \log B$, which is significant in practice, and can be as much as $Z/z = B$ for large Z.

4.2 Disk Striping and Parallelism with 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* [209, 286] 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 2.3, 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 automatically to 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 Chapter 3—streaming, online search, and answer 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 [213]. The optimal number of I/Os to sort using one disk with block size B is

$$\Theta\left(n \log_m n\right) = \Theta\left(n \frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B} \frac{\log(N/B)}{\log(M/B)}\right). \tag{4.1}$$

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 (4.1) by DB, which gives the I/O bound

$$\Theta\left(\frac{N}{DB}\frac{\log(N/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D}\frac{\log(n/D)}{\log(m/D)}\right). \tag{4.2}$$

On the other hand, the optimal bound for sorting is

$$\Theta\left(\frac{n}{D}\log_m n\right) = \Theta\left(\frac{n}{D}\frac{\log n}{\log m}\right). \tag{4.3}$$

The striping I/O bound (4.2) is larger than the optimal sorting bound (4.3) by a multiplicative factor of

$$\frac{\log(n/D)}{\log n}\frac{\log m}{\log(m/D)} \approx \frac{\log m}{\log(m/D)}. \tag{4.4}$$

When D is on the order of m, the $\log(m/D)$ term in the denominator is small, and the resulting value of (4.4) 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 (4.3) 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 input or output is done independently. It suffices, for example, to do only input operations independently and to use disk striping for output operations. An advantage of using striping for output operations is that it facilitates the maintenance of parity information for error correction and recovery, which is a big concern in RAID systems. (We refer the reader to [98, 187] 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 [327]. 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 this work. Three such sorting algorithms we introduce in the next chapter—distribution sort and merge sort with randomized cycling (RCD and RCM) and simple randomized merge sort (SRM)—have relatively low overhead and 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 [213], and also because sorting is an important paradigm in the design of efficient EM algorithms, as we show in Chapter 9. With some technical qualifications, many problems that can be solved easily in linear time in the (internal memory) RAM model, 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.

In this chapter we discuss optimal EM algorithms for sorting. The following bound is the most fundamental one that arises in the study of EM algorithms:

Theorem 5.1 ([23, 264]). The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right). \tag{5.1}$$

The constant of proportionality in the lower bound for sorting is 2, as we shall see in Chapter 6, and we can come very close to that constant factor by some of the recently developed algorithms we discuss in this chapter.

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 (5.1) of Theorem 5.1.

In Sections 5.1–5.3 we discuss some recently developed external sorting algorithms that use disks independently and achieve bound (5.1). The algorithms are based upon the important *distribution* and *merge* paradigms, which are two generic approaches to sorting. They 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 work.

The distribution sort and merge sort methods using randomized cycling (RCD and RCM) [130, 195] from Sections 5.1 and 5.3 and the simple randomized merge sort (SRM) [64, 68] of Section 5.2 are the methods of choice for external sorting. For reasonable values of M and D, they outperform disk striping in practice and achieve the I/O lower bound bound (5.1) with the lowest known constant of proportionality.

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 input operations, but in a striped manner during parallel output operations. As a result, if we output $D-1$ blocks at a time in an I/O, the exclusive-or of the $D-1$ blocks can be output onto the Dth disk during the same I/O operation.

In Section 5.3 we develop a powerful notion of duality that leads to improved new algorithms for prefetching, caching, and sorting. In Section 5.4 we show that if we allow independent inputs and output operations, 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 Chapter 6 we give lower bounds for sorting and related problems.

5.1 Sorting by Distribution

Distribution sort [213] is a recursive process in which we use a set of $S-1$ partitioning elements $e_1, e_2, \ldots, e_{S-1}$ to partition the current set of items into S disjoint subfiles (or *buckets*), as shown in Figure 5.1 for the case $D = 1$. The ith bucket, for $1 \leq i \leq S$, consists of all items with key value in the interval $[e_{i-1}, e_i)$, where by convention we let $e_0 = -\infty$, $e_S = +\infty$. The important property of the partitioning is that all the items in one bucket precede all the items in the next bucket. Therefore, we can complete the sort by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

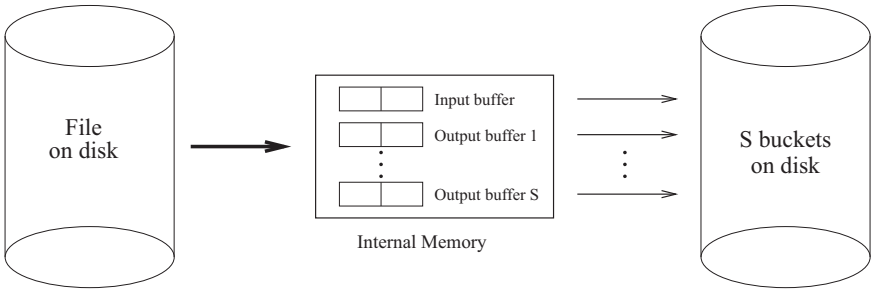

Fig. 5.1 Schematic illustration of a level of recursion of distribution sort for a single disk ($D = 1$). (For simplicity, the input and output operations use separate disks.) The file on the left represents the original unsorted file (in the case of the top level of recursion) or one of the buckets formed during the previous level of recursion. The algorithm streams the items from the file through internal memory and partitions them in an online fashion into S buckets based upon the key values of the $S-1$ partitioning elements. Each bucket has double buffers of total size at least $2B$ to allow the the input from the disk on the left to be overlapped with the output of the buckets to the disk on the right.

5.1.1 Finding the Partitioning Elements

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 can be output to disk, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number S of buckets (and partitioning elements) is $\Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$. In the last level of recursion, there is no point in having buckets of fewer than $\Theta(M)$ items, so we can limit S to be $O(N/M) = O(n/m)$. These two constraints suggest that the desired number S of partitioning elements is $\Theta\bigl(\min\{m, n/m\}\bigr)$.

It seems difficult to find $S = \Theta\bigl(\min\{m, n/m\}\bigr)$ partitioning elements deterministically 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 $S = \Theta\bigl(\min\{\sqrt{m}, n/m\}\bigr)$ partitioning elements [23, 263, 335], which has the effect of doubling the number of levels of recursion. 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 [308].

Probabilistic methods for choosing partitioning elements based upon random sampling [151] are simpler and allow us to choose $S = O\bigl(\min\{m, n/m\}\bigr)$ partitioning elements in $o(n/D)$ I/Os: Let $d = O(\log S)$. We take a random sample of dS items, sort the sampled items, and then choose every dth item in the sorted sample to be a partitioning element. Each of the resulting buckets has the desired size of $O(N/S)$ items. The resulting number of I/Os needed to choose the partitioning elements is thus $O\bigl(dS + \mathit{Sort}(dS)\bigr)$. Since $S = O\bigl(\min\{m, n/m\}\bigr) = O(\sqrt{n}\,)$, the I/O bound is $O(\sqrt{n}\log^2 n) = o(n)$ and therefore negligible.

5.1.2 Load Balancing Across the Disks

In order to meet the sorting I/O bound (5.1), 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. The challenge is in the more general multiple-disk case: Each input I/O step and each output I/O 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. In summary, the challenge in distribution sort is to output 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 input 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 output in "logical blocks" of size CB, one per cluster. Choosing $C = \sqrt{D}$ will not change the 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 [335] 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 output operation sends 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 on the average, whereas any individual bin contains an average of only one ball [331].[1] 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 streamed through internal memory in one pass, one memoryload at a time. Each memoryload is independently and randomly permuted and output back to the disks in the new order. In a second pass, the file is input 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. [134] 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 [263]. 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

[1] We use the notation $\ln d$ to denote the natural (base e) logarithm $\log_e d$.

output 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 output 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 bring a bucket into memory during the next level of recursion will be within a small constant factor of the optimum.

5.1.3 Randomized Cycling Distribution Sort

The distribution sort methods that we mentioned above for parallel disks perform output operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks that belong to a given stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during the input operations in the next level of recursion. In the output phase, each bucket must therefore keep track of the last block output 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 input operations can be done in a striped manner. As a result, the output I/O operations must use disks independently, since during each output step, multiple buckets will be transmitting 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 stream the data items through internal memory and partition them 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, 1, 2, 3, ..., D, 1, 2, 3, ..., D, ...), 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 [332] solve this 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. That is, each bucket has its own random permutation ordering, chosen independently from those of the other $S-1$ buckets, and the blocks of each bucket are output to the disks in a round-robin manner using its permutation ordering. If two blocks (from different buckets) happen to collide during an output to the same disk, one block is output to the disk and the other is kept in an output buffer in internal memory. With high probability, subsequent blocks in those two buckets will be output to different disks and thus will not collide.

As long as there is a small pool of D/ϵ block-sized output buffers to temporarily cache the blocks, Vitter and Hutchinson [332] show analytically that with high probability the output proceeds optimally in $(1+\epsilon)n$ I/Os. We also need $3D$ blocks to buffer blocks waiting to enter the distribution process [213, problem §5.4.9–26]. There may be some blocks left in internal memory at the end of a distribution pass. In the pathological case, they may all belong to the same bucket. This situation can be used as an advantage by choosing the bucket to recursively process next to be the one with the most blocks in memory.

The resulting sorting algorithm, called *randomized cycling distribution sort* (RCD), provably achieves the optimal sorting I/O bound (5.1) on the average with extremely small constant factors. In particular, for any parameters $\epsilon,\ \delta > 0$, assuming that $m \geq D(\ln 2 + \delta)/\epsilon + 3D$, the average number of I/Os performed by RCD is

$$\left(2 + \epsilon + O(e^{-\delta D})\right) \frac{n}{D} \left\lceil \log_{m-3D-D(\ln 2+\delta)/\epsilon} \frac{n}{m} \right\rceil + 2\frac{n}{D} + o\left(\frac{n}{D}\right). \quad (5.2)$$

When $D = o(m)$, for any desired constant $0 < \alpha < 1$, we can choose ϵ and δ appropriately to bound (5.2) as follows with a constant of

proportionality of 2:

$$\sim 2\frac{n}{D}\lceil \log_{\alpha m} n \rceil. \tag{5.3}$$

The only differences between (5.3) and the leading term of the lower bound we derive in Chapter 6 are the presence of the ceiling around the logarithm term and the fact that the base of the logarithm is arbitrarily close to m but not exactly m.

RCD operates very fast in practice. Figure 5.2 shows a typical simulation [332] that indicates that RCD operates with small buffer memory requirements; the layout discipline associated with the SRM method discussed in Section 5.2.1 performs similarly.

Randomized cycling distribution sort and the related merge sort algorithms discussed in Sections 5.2.1 and 5.3.4 are the methods of choice for sorting with parallel disks. Distribution sort algorithms may

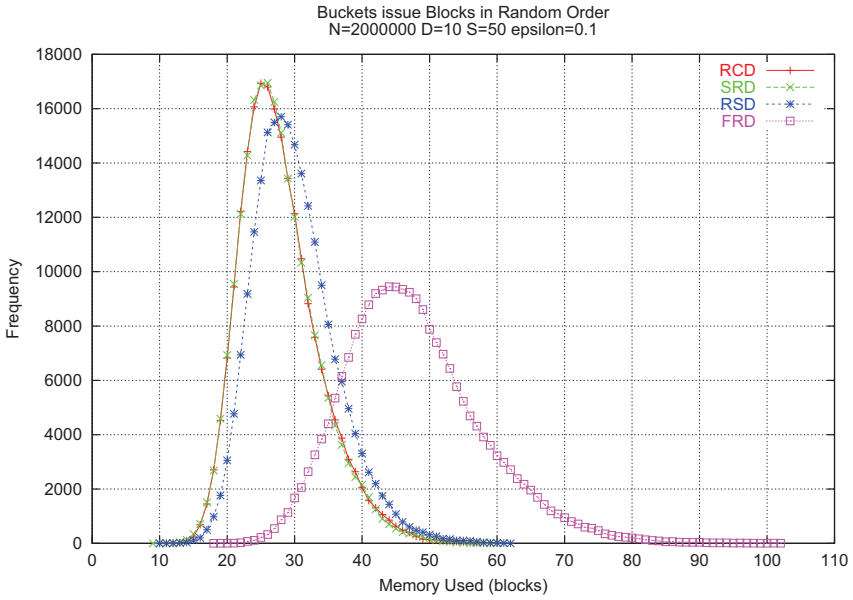

Fig. 5.2 Simulated distribution of memory usage during a distribution pass with $n = 2\times10^6$, $D = 10$, $S = 50$, $\epsilon = 0.1$ for four methods: RCD (randomized cycling distribution), SRD (simple randomized distribution—striping with a random starting disk), RSD (randomized striping distribution—striping with a random starting disk for each stripe), and FRD (fully randomized distribution—each bucket is independently and randomly assigned to a disk). For these parameters, the performance of RCD and SRD are virtually identical.

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 [334]. On the other hand, the merge approaches can take advantage of replacement selection to start off with larger run sizes.

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 [213]: 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 $M \gg B$ [130, 213].

After the initial runs are formed, the merging phase begins, as shown in Figure 5.3 for the case $D = 1$. 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 (5.1), 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 input operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed. The challenge is to ensure that those blocks reside on different disks so that they can be input in a single parallel 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 output to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

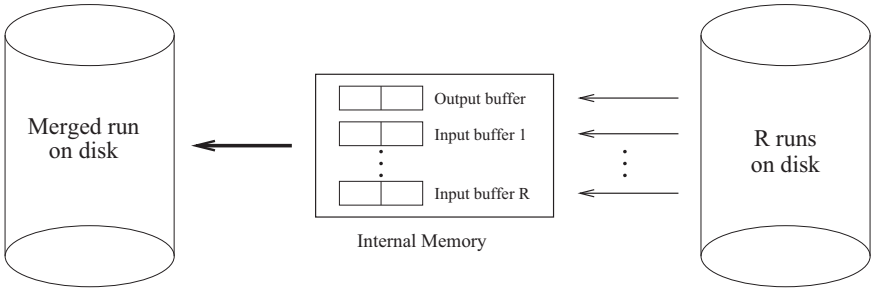

Fig. 5.3 Schematic illustration of a merge during the course of a single-disk ($D = 1$) merge sort. (For simplicity, the input and output use separate disks.) Each of R sorted runs on the disk on the right are streamed through internal memory and merged into a single sorted run that is output to the disk on the left. Each run has double buffers of total size at least $2B$ to allow the the input from the runs to be overlapped with the output of the merged run.

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 [166, 213]: 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 [264] 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 input 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 added to the output stripe, and the remaining B items are output back to the disks. The resulting run that is produced is only an "approximately" merged run, but its saving grace is that no two in-

verted items are very far apart. A final application of Columnsort [225] suffices to restore total order; partial striping is employed to meet the memory constraints. One disadvantage of Greed Sort is that the input and output I/O operations involve independent disks and are not done in a striped manner, thus making it difficult to write parity information for error correction and recovery.

Chaudhry and Cormen [94] show experimentally that oblivious algorithms such as Columnsort work well in the context of cluster-based sorting.

Aggarwal and Plaxton [22] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm [122]. 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 the optimum), but the constant factor is larger than for the distribution sort methods.

5.2.1 Simple Randomized Merge Sort

One approach to merge sort is to stripe each run across the disks and use the disk striping technique of Section 4.2. However, disk striping devotes too much internal memory (namely, $2RD$ blocks) to cache blocks not yet merged, and thus the effective order of the merge is reduced to $R = \Theta(m/D)$ (cf. (4.2)), which gives a nonoptimal result.

A better approach is the *simple randomized merge sort* (SRM) algorithm of Barve et al. [64, 68], referred to as "randomized striping" by Knuth [213]. It uses much less space in internal memory for caching blocks and thus allows R to be much larger. 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 input into memory, and if there is not enough room, the least needed blocks are "flushed" back to disk (without any I/Os required) to free up space.

Barve et al. [64] derive an asymptotic upper bound on the expected I/O performance, with no assumptions about the original distribution of items in the file. A more precise analysis, which is related to the so-

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

Table 5.1 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.

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"). We conjecture that the expected maximum bin size in the cyclic occupancy problem is at most that of the classical occupancy problem [64, 213, problem 5.4.9–28]. The bound has been established so far only in an asymptotic sense [64].

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. Barve and Vitter [68] give experimental confirmation of the speedup with six fast disk drives and a 500 megahertz CPU, as shown in Table 5.1.

When the internal memory is large enough to provide $\Theta(D \log D)$ blocks of cache space and $3D$ blocks for output buffers, SRM provably achieves the optimal I/O bound (5.1). For any parameter $\epsilon \to 0$, assuming that $m \ge D(\log D)/\epsilon^2 + 3D$, the average number of I/Os performed by SRM is

$$(2 + \epsilon)\frac{n}{D}\left\lceil \log_{m-3D-D(\log D)/\epsilon^2} \frac{n}{m} \right\rceil + 2\frac{n}{D} + o\left(\frac{n}{D}\right). \tag{5.4}$$

When $D = o(m/\log m)$, for any desired constant $0 < \alpha < 1$, we can choose ϵ to bound (5.4) with a constant of proportionality of 2:

$$\sim 2\frac{n}{D}\lceil \log_{\alpha m} n \rceil. \tag{5.5}$$

6

Lower Bounds on I/O

In this chapter we prove the lower bounds from Theorems 5.1–5.4, including a careful derivation of the constants of proportionality in the permuting and sorting lower bounds. We also mention some related I/O lower bounds for the batched problems in computational geometry and graphs that we cover later in Chapters 8 and 9.

6.1 Permuting

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. Permuting is one of several simple problems that can be done in linear CPU time in the (internal memory) RAM model. But if we assume that the N items are indivisible and must be transferred as individual entities, permuting requires a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter B.

Our main result for parallel disk sorting is that we close the gap between the upper and lower bounds up to lower order terms. The lower bound from [23] left open the nature of the constant factor of

proportionality of the leading term; in particular, it was not clear what happens if the number of output steps and input steps differ.

Theorem 6.1 ([195]). Assuming that $m = M/B$ is an increasing function, the number of I/Os required to sort or permute n indivisible items, up to lower-order terms, is at least

$$\frac{2N}{D}\frac{\log n}{B\log m + 2\log N} \sim \begin{cases} \dfrac{2n}{D}\log_m n & \text{if } B\log m = \omega(\log N); \\ \dfrac{N}{D} & \text{if } B\log m = o(\log N). \end{cases} \tag{6.1}$$

The main case in Theorem 6.1 is the first one, and this theorem shows that the constant of proportionality in the $Sort(N)$ bound (5.1) of Theorem 5.1 is at least 2.

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 optimum way to permute the items is to move them one at a time in the naive manner, not making use of blocking.

We devote the rest of this section to a proof of Theorem 6.1. 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 D.

We call an input operation *simple* if each item that is transferred from the disk gets removed from the disk and deposited into an empty location in internal memory. Similarly, an output is *simple* if the transferred items are removed from internal memory and deposited into empty locations on disk.

Lemma 6.2 ([23]). For each computation that implements a permutation of the N items, there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.

The lemma can be demonstrated easily by starting with a valid permutation computation and working backwards. At each I/O step,

in backwards order, we cancel the transfer of an item if its transfer is not needed for the final result; if it is needed, we make the transfer simple. The resulting I/O strategy is simple.

For the lower bound, we use the basic approach of Aggarwal and Vitter [23] and 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. In a similar way, we can get a lower bound on the average case by computing the value of t for which the bound first reaches $N!/2$.

In particular, we say that a permutation $\langle p_1, p_2, \ldots, p_N \rangle$ of the N items can be *produced after t_I input operations and t_O output operations* if there is some intermixed sequence of t_I input operations and t_O output operations so that the items end up in the permuted order $\langle p_1, p_2, \ldots, p_N \rangle$ in extended memory. (By extended memory we mean the memory locations of internal memory followed by the memory locations on disk, in sequential order.) The items do not have to be in contiguous positions in internal memory or on disk; there can be arbitrarily many empty locations between adjacent items.

As mentioned above, we can assume that I/Os are simple. Each I/O causes the transfer of exactly B items, although some of the items may be **nil**. In addition, the I/Os obey block boundaries, in that all the non-**nil** items in a given I/O come from or go to the same block on disk.

Initially, before any I/Os are performed and the items reside on disk, 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

$$N(1 + \log N). \tag{6.2}$$

For the case of an input operation, we first consider an input I/O from a specific block on disk. If the b items involved in the input 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 they were in internal memory. Thus, the $b!$ possible orderings of the b input items relative to themselves could already have been produced before the input operation. This implies in a subtle way that rearranging the newly input 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 prior time the b input items were together in internal memory, an input operation can boost the number of producible permutations by at most

$$N(1 + \log N)\binom{M}{b}. \tag{6.3}$$

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

$$(B!)^{N/B}. \tag{6.4}$$

Let I be the total number of input I/O 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 put into 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 that arrive, 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:

$$O \geq \frac{1}{B}\Big(\sum_{1\leq i\leq I} b_i\Big). \tag{6.5}$$

Combining (6.2), (6.3), and (6.4), we find that

$$\big(N(1+\log N)\big)^{I+O} \prod_{1\leq i\leq I} \binom{M}{b_i} \geq \frac{N!}{(B!)^{N/B}}, \tag{6.6}$$

where O satisfies (6.5).

Let $\tilde{B} \leq B$ be the average number of items input during the I input operations. By a convexity argument, the left-hand side of (6.6) is maximized when each b_i has the same value, namely, $\tilde{B}$. We can rewrite (6.5) as $O \geq I\tilde{B}/B$, and thus we get $I \leq (I+O)/(1+\tilde{B}/B)$. Combining these facts with (6.6), we get

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{\tilde{B}}^{I} \geq \frac{N!}{(B!)^{N/B}}; \tag{6.7}$$

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{\tilde{B}}^{(I+O)/(1+\tilde{B}/B)} \geq \frac{N!}{(B!)^{N/B}}. \tag{6.8}$$

By assumption that M/B is an increasing function, the left-hand side of (6.8) maximized when $\tilde{B} = B$, so we get

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{B}^{(I+O)/2} \geq \frac{N!}{(B!)^{N/B}}. \tag{6.9}$$

The lower bound on $I+O$ for $D=1$ follows by taking logarithms of both sides of (6.9) and solving for $I+O$ using Stirling's formula. We get the general lower bound of Theorem 6.1 for D disks by dividing the result by D.

6.2 Lower Bounds for Sorting and Other Problems

Permuting is a special case of sorting, and hence, the permuting lower bound of Theorem 6.1 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