A Simple and Efficient Parallel Disk Mergesort[1]

Rakesh D. Barve[2] *Jeffrey Scott Vitter*[3]

[1]A shortened version of this paper appeared in *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, St. Malo, France, June 1999, 232–241.

[2]Winphoria Networks, c/o B23 Swagat Mahakali Caves Road, Andheri, Mumbai 400 093, India. Part of this work was done while at Duke University. Supported in part by an IBM Graduate Fellowship, by Army Research Office MURI grant DAAH04–96–1–0013, and by National Science Foundation research grant CCR–9522047. Phone: (919) 660–6507, FAX: (919) 660-6502, Email: rbarve@winphoria.com

[3]Center for Geometric and Biological Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Part of this work was done while the author was visiting I.N.R.I.A. in Sophia Antipolis, France. Supported in part by Army Research Office grants DAAH04–96–1–0013 (MURI) and DAAD19–01–1–0725 and by National Science Foundation research grants CCR–9877133, EIA–9870724, and CCR–0082986. Email: jsv@cs.duke.edu

Abstract

External sorting—the process of sorting a file that is too large to fit into the computer's internal memory and must be stored externally on disks—is a fundamental subroutine in database systems [Gra93, IBM90]. Of prime importance are techniques that use multiple disks in parallel in order to speed up the performance of external sorting. The *simple randomized merging* (*SRM*) mergesort algorithm proposed by Barve et al. [BGV97] is the first parallel disk sorting algorithm that requires a provably optimal number of passes and that is fast in practice. Knuth [Knu98, Section 5.4.9] recently identified SRM (which he calls "randomized striping") as the method of choice for sorting with parallel disks.

In this paper we present an efficient implementation of SRM, based upon novel and elegant data structures. We give a new implementation for SRM's *lookahead forecasting* technique for parallel prefetching and its *forecast and flush* technique for buffer management. Our techniques amount to a significant improvement in the way SRM carries out the *parallel, independent* disk accesses necessary to read blocks of input runs efficiently during external merging. Our implementation is based on *synchronous* parallel I/O primitives provided by the TPIE programming environment [TPI99]; whenever our program issues an I/O read (write) operation, one block of data is synchronously read from (written to) each disk in parallel.

We compare the performance of SRM over a wide range of input sizes with that of *disk-striped mergesort (DSM)*, which is widely used in practice. DSM consists of a standard mergesort in conjunction with striped I/O for parallel disk access. SRM merges together significantly more runs at a time compared with DSM, and thus it requires fewer merge passes. We demonstrate in practical scenarios that even though the streaming speeds for merging with DSM are a little higher than those for SRM (since DSM merges fewer runs at a time), sorting using SRM is often significantly faster than with DSM (since SRM requires fewer passes).

The techniques in this paper can be generalized to meet the load-balancing requirements of other applications using parallel disks, including distribution sort and multiway partitioning of a file into several other files. Since both parallel disk merging and multimedia processing deal with streams that get "consumed" at nonuniform and partially predictable rates, our techniques for lookahead based upon forecasting data may have relevance in video server applications.

1 Introduction and Motivation

The problem of external sorting (i.e., sorting a massive data set that is too large to fit into the computer's internal memory and must be stored externally on disks) is fundamental to database systems. External sorting not only is a common application but it is also a core subroutine in many other database operations [Gra93, IBM90], as well as in external memory graph algorithms [CGG$^+$95] and geometric algorithms [GTVV93]. Modern technology trends indicate that processor speeds are increasing at a faster rate than disk drive performance [GVW96, Vit01], and so the development of external sorting techniques capable of utilizing multiple disks in parallel is of prime importance for database systems.

External mergesort is the most commonly used technique to perform large-scale sorting [ZL98]. In this paper we address problems arising in the development of a simple and efficient parallel disk mergesort. External mergesort consists of a run formation phase, which produces sorted runs, and a merge phase, which merges sorted runs to produce the sorted output. While it is simple to modify run formation techniques developed for single disk systems to work efficiently on parallel disk systems, fundamental difficulties[1] need to be overcome in order to merge together several runs, each one striped across the disks, in a manner that efficiently utilizes all the disks in parallel.

In this paper we present the design, implementation, and performance of an extremely simple and efficient parallel disk merging technique. Our implementation is based on parallel I/O primitives provided by the TPIE [TPI99] programming environment for external memory programming. High-performance external sorting depends crucially on being able to *overlap* I/O and CPU processing fully. To achieve this, it is imperative to use *non-blocking* I/O operations in which the file system cache is bypassed and data is transferred directly to and from buffers provided by the application, thereby avoiding expensive copy operations. The TPIE I/O primitives we employ for the implementation described in this paper are based on non-blocking *memory-mapped* file I/O which allows portions of a file to be mapped directly into the program's address space, avoiding the need for any data copying and allowing file access using simple array references. Moreover, overlapping of I/O and CPU processing is made possible in our implementation by the fact that the algorithms implemented access files almost exclusively in a sequential pattern and the operating system applies sequential read-ahead also to memory mapped files. As a consequence, our performance comparisons also carry over to external sort implementations using more traditional mechanisms for non-buffered, non-blocking I/O based on explicit OS support other than memory-mapped file I/O.

The most significant aspects of our parallel disk merging technique are its elegant and novel data structures and the prefetching and buffer management technique that together constitute a practical implementation of the external mergesort algorithm SRM (simple randomized mergesort) [BGV97], which was shown to have provably efficient parallel I/O performance guarantees for the parallel disk model of Vitter and Shriver [VS94]. SRM was recently identified by Knuth [Knu98][2] in the new edition of his seminal work as the method of choice for optimal sorting on parallel disks. Another interesting aspect of our implementation technique is that it can easily be modified to suit the load balancing needs of other applications in a parallel disk context, including external distribution sort and a multiway partitioning of a file into other files. The technique we develop to implement a lookahead mechanism via forecasting also has potential applications during parallel prefetching in video servers in which many striped files may need to be streamed at nonuniform rates with real

[1]It is easy to carry out an efficient parallel disk merge of several runs when each run resides entirely on a single disk [PSV94], provided the output can be striped across all the disks. However such a merging scheme is fundamentally inefficient for a parallel disk mergesort, since extra transposition passes would be needed.

[2]Knuth refers to the SRM algorithm as "randomized striping".

time constraints.

Although a tremendous amount of research has been conducted on external sorting [ECW94, ZL98, Sal89, ZL96], the focus has largely focussed on single disk systems and on goals such as efficient layout of disk blocks, efficient scheduling of I/O at disks, and techniques to implement read-aheads in course of external sorting. In this paper we focus on the orthogonal approach of minimizing elapsed time by developing techniques to exploit I/O parallelism and minimize *parallel I/O operations* in a system containing several disks. While some of the techniques developed in [ECW94, ZL98, Sal89, ZL96] can potentially be used in conjunction with the ones we develop here, exploring that avenue is beyond the scope of this paper. The parallel disk model (PDM) [VS94] is meant for designing algorithms capable of exploiting I/O parallelism. In the PDM, an input file containing N items[3] is striped in blocks containing B items across D disk drives all of which may be used in parallel as follows: In each I/O operation, an application can transfer at most one block of B items between internal memory and each disk drive; so up to D blocks can be transferred in a single I/O operation. With respect to the problem of external sorting, results in [VS94] and earlier work [AV88] show that given an internal memory capable of holding up to M items, sorting a file of N items requires $\Theta\big(\frac{N}{DB}\log_{M/B}(N/B)\big)$ I/O operations. Sorting requires $\lceil\log_{M/B}(N/B)\rceil$ *passes* over the data; each pass can be done in a linear number of I/O operations (N/DB reads and N/DB writes). The main difficulty in parallel disk sorting is laying out intermediate data blocks, accessing blocks, and designing computation in such a manner that on average each I/O operation transfers $\Theta(D)$ blocks, and additionally, the data blocks residing in memory at a given time are such that the I/O required to get them there can be charged to the amount of "internal memory work" that can be accomplished using that set of memory resident data blocks. Several interesting parallel disk sorting algorithms [VS94, VN93, AP94] performing an optimal number $\Theta\big(\frac{N}{DB}\log_{M/B}(N/B)\big)$ of I/O operations have been proposed, but they are somewhat complicated and difficult to implement in practice.

As a consequence, an attractive alternative to implement sorting algorithms for parallel disks is to use the technique of *disk striping* (or *striped I/O*) in conjunction with well known single disk sorting techniques as follows: In each striped-I/O operation, the logical locations of the blocks accessed at each one of the D disks are the same. Logically, the effect of striped I/O is to reduce the number of disks to 1 and increase the block size to DB from the application's point of view. As a result, single disk algorithms such as external mergesort and external radix sort with block size configured to DB can be implemented to utilize D disks on a parallel disk system. Since double-buffered mergesort has been shown to be very efficient [Sal89], its disk-striped version, called disk-striped mergesort (DSM) [Ven94], is considered particularly attractive. Sorting algorithms such as DSM and disk-striped radix sort [CH96] based upon striped I/O are simple and all their I/O operations can achieve full D-disk parallelism. However, because the logical block size blows up from B to DB, the number of runs participating in each merge of DSM goes down by a factor of D, from order M/B to order M/DB, and hence DSM (and for similar reasons, all other striped-I/O sorting algorithms) require a non-optimal number $\lceil\log_{M/DB}(N/DB)\rceil$ of passes over the data. The degree of non-optimality increases with D; most importantly, the non-optimality shows up in practice even for a moderately small number D of disks.

Motivated by the need for a simple and provably efficient parallel disk sorting algorithm, Barve et al. [BGV97] proposed *simple randomized mergesort (SRM)*. The SRM algorithm is the first parallel disk sorting algorithm that requires a provably optimal number $\sim\lceil\log_{M/B}(N/B)\rceil$ of passes over the data and is simple enough to be considered a candidate for implementation. For practical ranges of the parameters B and D, each pass takes a linear optimal number $O(N/DB)$ of

[3]By items, we refer to records or tuples. While discussing I/O complexity bounds, it is convenient to state block sizes and file sizes in terms of items instead of bytes.

I/Os. While merging, SRM uses a generalization of Knuth's *forecasting* technique [Knu98] and a new buffer management technique called *forecast and flush* in order to access the D disks efficiently in a *parallel, independent* manner fundamentally different from striped I/O. The basic prefetching technique in SRM is to use forecasting information to read in the "smallest" block from each one of the D disks in every I/O operation; if there is not enough space in internal memory to read D blocks, SRM simply *flushes* (without any I/O) a sufficient number of "largest" blocks from memory.

DSM inherently *requires* synchronous parallel I/O operations because of the nature of disk-striping. For a direct comparison with DSM and because of the I/O primitives provided by TPIE, our SRM implementation uses synchronous parallel I/O operations. An interesting project, but one beyond the scope of this paper, would be to adapt the basic SRM buffer management and prefetching technique for a system that provides asynchronous parallel I/O operations on multiple disks. Asynchronous I/O should offer speedups in practice by making use of otherwise idle time; its advantages and disadvantages relative to the implementation presented in this paper merit a close look. In contrast to SRM, DSM cannot be adapted to exploit the potential advantages of asynchronous parallel I/O, because it relies fundamentally on disk striping.

1.1 Our Contributions

In this paper, we present novel and elegant data structures and techniques to implement the forecast and flush buffer management scheme of SRM. The latter scheme, as proposed in [BGV97], requires the use of D separate priority queues [CLR90], each corresponding to a unique disk and each involving R forecasting keys at any time, where R is the merge order. Furthermore, the scheme in [BGV97] did not cover details of internal memory management and how to track down efficiently the "largest" blocks in memory at the time of the flush operations. In this paper we present an implementation of the forecasting and flush scheme that requires only a single priority queue comprising R forecasting keys, to be used in conjunction with D ordinary queues implemented as simple arrays. The novelty lies in the way we store, use, and manage forecasting data during the merging process. We also show how to implement memory management and how to perform flush operations. Our design significantly simplifies the implementation of SRM.

The second interesting contribution of our paper is the practical comparison of the merging phase implementation of DSM and SRM, on a state-of-the-art computer system consisting of six disks that can be used independently and in parallel. Our implementation employs non-blocking and non-buffered memory-mapped I/O operations, making it possible for us to overlap I/O and CPU processing, which is crucial to external memory sorting. In each merge operation, DSM merges together approximately $M/2DB$ runs, whereas SRM merges together a significantly larger number of runs (which can go up to approximately $M/2B$). DSM will thus tend to have higher disk locality in each merging operation, compared with SRM. As a result, each merge pass of SRM incurs some overhead relative to each merge of DSM, on account of higher disk latencies (since a larger number of streams is involved) as well as because each merge pass may require *more* than N/DB read I/O operations [BGV97], as we explain later. Hence, even if SRM requires a smaller number of merge passes on a given input, comparing the practical performance of SRM and DSM's merging phases remains an interesting exercise. Our intuition is borne out in practice when we find that the data streaming speed attained by DSM while merging is noticeably better than that of SRM. However, the overhead in our implementation of SRM relative to DSM is small enough that SRM's merging phase easily outperforms DSM's merging phase by a significant margin.

An interesting aspect of our technique is that it can be easily modified to suit the load balancing needs of other applications in a parallel disk context, including external distribution sort and a multiway partitioning of a file into other files which is used in hash-join computations in database systems. The technique we develop to implement a lookahead mechanism via forecasting also has potential applications during parallel prefetching in video servers in which many striped files may need to be streamed at nonuniform rates with real time constraints.

In Section 2 we go through some preliminaries for external mergesort, discuss relevant previous work and describe DSM. In Section 3 we present the SRM merging algorithm and sketch previous results for SRM. In Section 4 we present our implementation of the SRM merging algorithm, including some pseudo-code. In Section 5 we present various aspects of the performance comparison of the merging phases of SRM and DSM. In Section 6 we present various applications of generalizations of the implementation techniques we develop here. Finally, in Section 7, we make some concluding remarks including avenues for related future work.

2 Preliminaries and Previous Work

External sorting has been studied extensively by many researchers. Knuth [Knu98] and Vitter [Vit01] give a comprehensive analysis of methods. External mergesort is the arguably most widely used of the external sorting techniques.

2.1 Run Formation + Merging Passes = Mergesort

External mergesort consists of a *run formation* phase followed by a *merging phase*. Run formation consists of repeatedly reading in a memoryload, sorting it in memory and writing it out to disk(s), thus resulting in the formation of $\Theta(N/M)$ sorted runs each of size $\Theta(M)$, which need to be merged together.

The merging phase consists of repeatedly carrying out *merge operations* until only a single sorted run remains. Each merge operation merges together some number R of runs, where R is called the *merge order*. The merge order R is typically set in accordance with the amount M of internal memory available, the size B of disk blocks, and in the case of striped-I/O mergesort, the number D of disks. The total number of *merge passes* over the data during the merging phase is $\lceil \log_R(N/M) \rceil$.

2.2 Participation Order of Blocks in a Merge

During a merge, data is transferred between internal memory and disks in blocks of B items. The blocks of the runs input to an external merge operation have a natural total order that is useful to define. In the process, we also define the crucial notion of the leading block of a run at any time.

Definition 1 Consider an external merge involving R input runs. A block of items is said to be *depleted* or *consumed* by a merge as soon as the last item in that block is written into the output run. The *leading block* of the rth run, where $0 \leq r < R$, at any time is defined as follows: At the beginning of the merge, the 0th block of the rth run is the leading block of that run. For $i > 0$, the ith block of the rth run becomes that run's leading block as soon as the $(i-1)$st block of that run gets depleted by the merge. A block is said to begin *participating* in the merge as soon as it becomes the leading block of its run. The *participation order*[4] of the blocks of the input runs is defined as follows: The 0th block of the rth run is the rth block in the participation order. The remaining blocks of the input runs follow the 0th blocks of the R runs in the order in which they begin participating in the merge.

In general the leading block of every input run needs to be in internal memory for a merge computation to proceed, unless the input run has been completely depleted by the merge. As a result, the order in which input blocks are read into memory tends to follow approximately the participation order of input blocks.

[4]Sometimes this order is also referred to as the *consumption sequence* [ZL98] of blocks of a merge.

2.3 Previous Work on External (Single Disk) Merging

It is beyond the scope of this paper to carry out a comprehensive survey of the vast body of research on external merging. Salzberg [Sal89] showed that double buffering [Knu98] with reasonably large sized buffers is an efficient approach to implement external merging in general. Zheng and Larson [ZL96] suggested using six to ten floating buffers per input run on average and proposed a planning strategy that utilizes the extra buffer space to read disk blocks in an order different from participation order with a view to optimizing seek time. Estivill-Castro and Wood [ECW94] extended this work, in part, to exploit pre-existing order in input data. Recently, Zhang and Larson [ZL98] suggested further improvements to the planning strategy via extended forecasting and block clustering.

The abovementioned approaches try to maximize overlapping of I/O and computation and minimize delays on account of disk latency during external merging. While Zheng and Larson do apply their planning strategy in a multiple disk situation, these studies are primarily oriented towards single disk systems. Our interest lies in speeding up external merging using the orthogonal approach of maximizing I/O parallelism. Although some of the abovementioned approaches can be used in conjunction with parallel I/O, we do not pursue that line of work in this paper. In this paper as in the NOWSort [ADADC$^+$97] implementation, we assume that efficient filesystem performance can be obtained as long as the logical block size B is reasonably large (of the order of 256 KB) and B is also the size of an input or output buffer in internal memory. Since all the merging approaches mentioned in the above paragraph use, on average, a constant number of input buffers for each input run, the merge order is $R = O(M/B)$ and the number of merge passes required is roughly $\log_{M/B}(N/M)$, which is optimal for the case when $D = 1$.

2.4 Parallel Disk Sorting using Disk Striping

Disk striping is a simple technique used to transform single disk sorting algorithms to parallel disk sorting algorithms. DSM is a double-buffered mergesort that can easily be implemented with striped I/O. The NOWSort [ADADC$^+$97] implementation uses a disk-striped mergesort to locally perform an external sort at each individual workstation. By the nature of striped I/O, the size of each input buffer in DSM is DB and the order R of external merging in DSM is $R \approx M/2DB$. Each run is striped blockwise across all the D disks in a round-robin manner. Initially the two buffers of each input run are read into internal memory. During the merge, whenever a run's leading buffer gets depleted a parallel read operation to load the next DB items into the free input buffer is issued, while the merge proceeds using the other input buffer of the run. The output of the merge is written in units of DB items and is doubly buffered as well.

Clearly, DSM has the advantage of being simple, enjoying full D-disk parallelism, and an overlap of computation and I/O activity. However, the number $\mathcal{P}_{\text{DSM}}$ of merge passes required in DSM is approximately $\lceil \log_{M/DB}(N/M) \rceil$, which can be larger than the optimal number $\mathcal{P}_{\text{OPT}} = \lceil \log_{M/B}(N/M) \rceil$ of passes by a factor[5] approaching $\Omega(\log D)$ as D approaches $\Omega(M/B)$. More importantly, from a practical point of view, the increase in the number of merge passes shows up even when the number D of disks is only moderately high, thus hindering the performance of DSM in practice.[6]

Another known parallel disk sorting implementation is the disk-striped radix sort [CH96], but it suffers from the same drawbacks as DSM mentioned above.

[5]As $D \longrightarrow M/cB$ where $c > 1$ is a constant, we have $\mathcal{P}_{\text{DSM}}/\mathcal{P}_{\text{OPT}} \longrightarrow 1 + \log_c D$.

[6]In the NOWSort parameters for the Minutesort record, the size of the local internal memory and the size of the file that needs to be sorted locally at each workstation is such that the number of runs merged during the merging phase is very small, so their application of disk-striped mergesort involves only a single merge pass.

2.5 Difficulty of Merging Optimally with Parallel Independent Disks

Optimal sorting on parallel disks requires the ability to access D disks in a *parallel, independent* manner in which different logical blocks may be accessed on each disk; this access mode is therefore fundamentally different from striped I/O. In order to sort optimally using parallel disks, a mergesort needs to merge optimally a large number $R = \Theta(M/B)$ of runs striped across D disks in each merge operation.[7]

Fundamental difficulties arise from the fact that very often during the merge there are times when the set of the R next participating blocks all reside on a small subset of the D disks, thereby causing "hotspots". Such hotspots are caused by the unpredictable, nonuniform rates at which runs get consumed by the merge. When there are hotspots, reading the set of the next R participating blocks can take many more parallel I/Os than the optimal number $\lceil R/D \rceil$ parallel I/Os. We refer the reader to [VS94, BGV97] for more intuition regarding the difficulty merging with parallel independent disks. Nodine and Vitter [VN93] overcame this difficulty by performing external merging by first *approximately* merging the runs followed by additional passes to refine the merge. Aggarwal and Plaxton's Sharesort [AP94] technique does repeated merging and has accompanying overheads. Each of these approaches involves extra overheads and are not ideal for practical implementation.

3 SRM Algorithm

The SRM algorithm of Barve et al. [BGV97] overcomes the difficulties involved in parallel disk merging by using a generalization of the forecasting technique in an elegant prefetching and buffer management scheme. In single disk systems, forecasting refers to the technique of using the last key of an in-memory block of a run to predict when the next block of that run will begin participating in the merge. In SRM, whenever intermediate runs are written to disk (during run formation or as the output of a merge operation), they are striped blockwise in a round-robin manner across the D disks. While writing the blocks of the rth run to disk, SRM implants the following forecasting information in each block of that run: In the ith block of the rth run, it stores the *key value* of the last item in the $(i + D - 1)$st block[8] of that run. *SRM uses the forecasting information in the ith block of a run to predict the time at which the $(i + D)$th block of that run begins participating in the merge.* If the ith block of a run is from disk d, then the $(i + D)$th block of that run is the *next block of that run on disk d.*

The round-robin blockwise striping employed by SRM while writing out runs to disk differs from the usual striping technique in the following sense: The first block of a run is written on a disk d_0 chosen uniformly at random from among the D disks; thereafter, blocks of the run are placed in the usual round-robin fashion on disks $(d_0 + 1) \bmod D$, $(d_0 + 2) \bmod D$, $\ldots$ and so on. This is the only application of randomization in SRM. The randomization helps SRM avoid poor merging performance for any particular ordering of items in the input file. The probabilistic analysis in [BGV97, Knu98] of SRM's I/O performance is with respect to the randomization mentioned here; there are no assumptions whatsoever regarding the input file to be sorted.

3.1 The Forecast and Flush Scheme

We now present the simple prefetching and buffer management scheme employed by SRM. The total number of internal memory blocks used by SRM as presented in [BGV97] is D blocks for blocks actively being read into memory, R blocks to hold the R leading blocks of the R runs, R

[7]It is enough to merge $R = \Theta\big((M/B)^c\big)$ runs, for any constant $0 < c \leq 1$, in each merge operation in order to attain optimality (within constant factors) in the number of passes.

[8]In the original presentation of SRM, the forecasting information in the ith block of a run is the key value of the first item in the $(i + D)$th block of that run. However, the approach here, taken from [Knu98], is a little simpler.

blocks for holding prefetched data, and an additional $2D$ blocks for output run data.[9] The *floating buffers* technique is used to implement internal memory management. The internal merge process works on the R leading blocks corresponding to the R input runs to produce blocks of the output run. As soon as an in-memory block becomes a leading block, it is pinned in internal memory until it gets depleted. The same holds for a block that becomes a leading block while still on disk, as soon as it is read into internal memory. Whenever each one of a set of D blocks of the output run has its forecasting information, that set of blocks is written out with full D-disk parallelism. The forecast and flush scheme for prefetching and buffer management in order to "feed" the internal computation works as follows:

1. Until there are no more input blocks to be read into internal memory:
    (a) Find out, for each disk, the smallest block (with respect to the participation order) among all blocks on that disk. Suppose that of these D smallest blocks (one per disk), ℓ of them are leading blocks of their respective runs, where $0 \leq \ell \leq D$. (Invariants ensure [BGV97] that the number of free blocks in memory is at least ℓ.)
    (b) If the number of free blocks available to hold prefetched data is $D - \ell - f$, for some $f > 0$, then *flush* out f of the largest blocks (with respect to the participation order) among all the prefetched blocks. This merely involves tracking down the f largest blocks among the prefetched blocks in internal memory and then simply marking them as free blocks; so there is no I/O involved in flushing. If at least one block is flushed, update the information about the smallest blocks on the D disks (since we now pretend as though the flushed blocks are on disk.)
    (c) In parallel, read in the smallest block from each one of the D disks.

3.2 Provable Performance Guarantees

As stated above, SRM merges approximately $M/2B$ runs in each merge operation and so the number of merge passes it requires is $\sim \mathcal{P}_{\text{OPT}} = \lceil \log_{M/B}(N/M) \rceil$, which is optimal. Write operations during SRM proceed at full disk parallelism as indicated above. A rigorous analysis of the expected number of parallel reads (Step 1c) required in SRM is presented in [BGV97]; here, the expectation is with respect to the randomization used by SRM to choose the starting disk for each intermediate output run, and the analysis is worst-case and so holds for any arbitrary inputs.

The flushing[10] of blocks in Step 1b above may lead to extra parallel read operations; so the expected number of parallel read operations required in an SRM merging pass can, in general, exceed N/DB. This brings us to the following definition.

Definition 2 The *parallel I/O overhead* $\nu \geq 1$ of an SRM merging pass is defined as the ratio between the number of parallel read operations incurred during that merging pass and the optimal quantity N/DB.

The analysis of SRM in [BGV97] implies that for most values of M, D, and B which together determine in SRM, the expected value of ν is 1 or a small constant greater than 1. Although the upper bound analysis indicates that there are some R, D pairs for which the expected number of parallel read operations in an SRM merging pass is non-optimal by small factors, simulations

[9]Knuth [Knu98] points out that SRM can be configured to work with any number $R + m'$ of blocks for prefetched data as long as $m' \geq D - 1$; so there is some flexibility here.

[10]The algorithm, as stated, may flush out some blocks that will be read in immediately in the next parallel read operation. Such blocks need not be flushed and in practice a check can easily be enforced to prevent such blocks being flushed and then ensuring that no block is read from the disk(s) corresponding to these blocks in the ensuing parallel read operation. For reasons of brevity, we do not delve into this detail in the rest of the paper.

suggest that the upper bound analysis is pessimistic [BGV97], and SRM's performance in practice is optimal, with ν close to 1. Knuth [Knu98] recently identified SRM as the method of choice for sorting on parallel disks.

3.3 Data Structures Required in the Straightforward Implementation

In a direct implementation of SRM as stated, one *forecasting heap* [BGV97] would be required for each one of the D disks in order to keep track of the smallest block (with respect to the participation order) on each disk at any time. Each forecasting heap is basically a priority queue [CLR90]. In general, at any time, the forecasting heap for disk d would contain R elements, each one corresponding to the smallest block of a unique input run on disk d at that time. Whenever a parallel read operation is completed, all the D forecasting heaps would have to be updated. Flush operations also require updating one or more forecasting heaps.

Additionally, SRM needs to maintain order among the prefetched blocks, since from time to time, the algorithm may need to flush out of internal memory up to $D-1$ of the largest prefetched blocks. There is also a need to ensure that I/O and computation are overlapped as far as possible.

4 Implementation Techniques and Data Structures for SRM

In this section we present an implementation of the forecast and flush scheme using novel data structures and techniques. Our approach greatly simplifies the task of implementing SRM. We require only one priority queue in conjunction with D ordinary queues, as opposed to the D priority queues required by a naive implementation. Maintenance of order among prefetched blocks falls out as a natural consequence of our technique. We also propose a simple technique to overlap I/O and computation.

For the moment, we assume that we have access to an appropriate high-level interface to specify I/O. In the next section, we show how such an interface was implemented in the TPIE [TPI99] programming environment for external memory programming. In Section 4.1 we describe a novel approach based on a *forecasting heap* data structure which provides a method to process forecasting data to track the smallest block on each disk in our merge implementation: Section 4.1.1 considers the practical situation in which the forecasting heap can be implemented completely in main memory using only a small fraction of available main memory whereas Section 4.1.2 considers exceptional situations in which the forecasting heap implementation involves external memory operations. In Section 4.2 we describe how the forecasting heap is used to maintain the core data structures in our implementation: The *lookahead queue*, which helps maintain blocks in memory in the participation order, and the D *occupancy queues* (one per disk), which track the participation order of blocks on each disk. The f blocks to be flushed (Step 1b in Section 3.1) correspond to the f trailing in-memory blocks in the lookahead queue. The D blocks read in a parallel read operation (Step 1c in Section 3.1) are the disk blocks at the head of the D occupancy queues. Section 4.2 also describes certain basic operations at the core of our implementation. Section 4.3 illustrates the use of the above data structures and the basic operations, and finally Section 4.4 describes the implementation in detail.

4.1 Managing Forecasting Data

In single disk merging, the key of the last item of a block in a run is the forecaster for the next block of that run, so there is no inherent need to store forecasting information explicitly in blocks. However, in SRM, given the ith block of a run, we need to be able to forecast when the $(i+D)$th block of that run will begin participating, so forecasting information has to be stored explicitly. While the original implementation of SRM proposed implanting one forecasting key (the key of the last item in the block $D-1$ blocks farther in the run) in each block of a run, here we propose that forecasting information be managed altogether separately.

Our proposal is motivated by the observation that in most applications in practice (and in particular in database applications), the size of the forecasting data involved in a merge operation is *much smaller* than the size of the runs participating in the merge. If the size of an item is I bytes and the size of its key is K bytes, then while the size (in bytes) of the runs input to a merging pass is $N \cdot I$ bytes, the size of the corresponding forecasting data is only NK/B bytes, so the forecasting data is smaller by a factor of BI/K, which is large in practice. For instance, database benchmarks typically have $I = 100$ bytes and $K = 10$ bytes. If $B = 1000$ items,[11] then the forecasting data is 10000 times smaller than the files being merged. Thus while merging files totally involving 1 GB of data, the total amount of forecasting data is only $\approx$ 100 KB, which is typically a very small fraction of the main memory.

Given the importance of the role of forecasting in SRM and the size of internal memory likely to be used, *in most situations all the forecasting data relevant to a merge operation can be kept resident in internal memory.* One way to implement this would be first to read in the small file(s) containing all the forecasting data related to the input runs at the beginning of the merge operation. The forecasting data corresponding to the output run can either be written out at once at the end of the merge operation or written out in a blocked manner from time to time during the merge. The forecasting data corresponding to the input runs can be disposed of after the merge operation. Using this approach the number of parallel I/O operations for reading and writing forecasting data over a merging pass would be $\lceil 2NK/DB^2I \rceil$, which is a small fraction of the minimum number $\lceil 2N/DB \rceil$ of parallel I/O operations required for transferring the run data blocks during the merging pass. An alternative implementation that is much easier and likely to be feasible most of the time, is never have forecasting data go to disk at any time throughout the entire mergesort. Forecasting data is intermediate data generated during SRM, so there is no fear of it being lost on account of system failures. In internal memory, two forecasting buffers can be used over all merging passes; the two buffers flip-flop in their role as buffers for input run forecasting data and output run forecasting data from each merge pass to the next.

However, as we point out in Section 4.1.3, there may be exceptional situations in which having the forecasting data resident in internal memory may not reasonable. For instance, this can happen while merging terabytes of data; in such cases, forecasting data may consume either a significant fraction of available internal memory or (in really extreme situations) may even exceed internal memory. For such cases, we propose yet another technique to process forecasting data that will consume only a small fraction of internal memory (only a small portion of the forecasting data corresponding to a merge operation will be in memory at a time) and will incur only a small I/O overhead relative to the I/O required for transferring the run data blocks.

In the remainder of this section we describe the the precise operations on forecasting data in our implementation, followed by a discussion of how to handle the forecasting data when it cannot be kept in internal memory.

4.1.1 The Forecasting Heap

Consider an SRM merge operation involving R input runs. Since we propose to manage the forecasting data separately from the runs themselves, for each run there is a *forecasting data run* that contains the forecasting keys of that run in sorted order. In all situations other than the exceptional situation discussed in Section 4.1.2, the R forecasting data runs remain resident in internal memory during the course of the merge. The only operation that needs to be supported on the forecasting data runs in order to facilitate our implementation of SRM's forecast and flush scheme is *an incremental merge of the R forecasting data runs that outputs one forecasting key at*

[11] In our implementation and in [ADADC+97], B is even larger.

a time. Such an incremental merge can be implemented by having a priority queue that contains, at any time, the leading forecasting key of each of the R forecasting runs. The smallest key in the priority queue is output at each step; if that key belongs to the rth forecasting data run, then the next key from that run is inserted into the priority queue. The process can thus continue one step at a time. We use the term *forecasting heap* to refer to the priority queue on the forecasting data runs.

The forecasting key output at each step predicts the time at which some block from some run begins participating in the merge. The order of the forecasting keys output by the forecasting heap is precisely the participation order of their corresponding run blocks. By maintaining R counters, one per run, each initialized to 0 at the start of the merge, we can keep track of the index of the first block in the rth run whose participation time has not yet been predicted; we simply increment the rth counter whenever the forecasting heap outputs a key from the rth run. The forecasting heap can thus be used to perform a "lookahead" as we discuss below.

4.1.2 When Forecasting Data is Too Large

In the exceptional situations when the forecasting data involved in a merge operation is so large that it would consume a significant portion of or may even exceed available internal memory, we propose the following implementation for the forecasting heap. In such situations, we choose a special block size B_f (*in bytes*) for processing files containing forecasting data runs. The size of B_f is chosen such that $B_f = BI/b_f$, where b_f is a small constant reasonably greater than 1 that is chosen for a desired performance. The idea is that a portion of size $R \cdot B_f$ bytes of internal memory is dedicated to the forecasting heap; a buffer of B_f bytes is used to buffer each forecasting data run. Whenever the buffer of a forecasting data run gets depleted, we read in (using a sequential I/O operation from a single disk) the next B_f bytes from the forecasting data run. In this manner the forecasting heap can easily be implemented.

The $R \cdot B_f$ bytes occupied by the forecasting heap can be made small by choosing a large value of b_f, but since the total number of extra I/O operations required over a merging pass on account of the forecasting heap would be $\lceil NKb_f/B^2I \rceil$, the parameter b_f should not be made too large. As a fraction of the minimum number $\lceil N/DB \rceil$ of (parallel) I/O operations required during a merge, the I/O overhead of the forecasting heap is DKb_f/BI, which is extremely small in most practical situations. For example, with $K = 10$ bytes, $I = 100$ bytes, $B = 1000$, and $b_f = 8$, the fractional I/O overhead is $D/1250$ which is small in practice. The memory usage for the forecasting heap in this case would be less than 1/16 of the internal memory usage of SRM, assuming that at least $2R + D$ blocks of internal memory are used by SRM to process the main merge. The memory and I/O overhead of the forecasting data corresponding to the output run of a merge operation can be somewhat smaller, but are in the same ballpark.

4.1.3 Determining When Forecasting Data Can Be Kept In Memory

The main advantage of SRM [BGV97] is the fact that it requires no more passes than the optimal number required in external sorting, and the overhead in the execution of each pass is very small. The SRM implementation proposed in this paper differs from [BGV97] in terms of the special treatment proposed here in order to handle forecasting data in practical situations. In order to determine when forecasting data can be kept entirely in memory, we consider how to determine the best merge order for the SRM implementation proposed in this paper: We find the smallest positive integer R' such that $\lceil \log_{R'} n \rceil = \log_{M/B} n$ and then decrement the value of R' until $M \geq M(R')$, where

$$M(R') = R'B + 2DB + (R' + 1) \cdot B_f.$$

The right hand side of the above inequality is the sum of the memory requirements (including input, output, and prefetch buffers) for an SRM merge of order R', including the buffers required by the external memory forecasting merge described in Section 4.1.2. Next, we find the smallest positive integer R'' such that $\lceil \log_{R'} n \rceil = \lceil \log_{R''} n \rceil$ and assign the value so obtained to R, the merge order we can use in our implementation.

Forecasting data can be kept entirely in memory if and only if

$$M \geq RB + DB + 2NK/B.$$

4.2 Other Data Structures and Primitive Operations

Let m denote the total number of internal memory blocks used for input run blocks, including leading blocks, prefetched blocks, and active blocks into which read operations are currently reading data. We maintain pointers to starting blocks of all the runs produced in a pass in a standard I/O-optimal external memory queue and at the start of each merge operation of order R, we load the R pointers corresponding to the runs being merged into memory. Our implementation maintains the set of R internal memory pointers, denoted $\mathit{Leading}_0$, $\mathit{Leading}_1$, ..., $\mathit{Leading}_{R-1}$, pointing to the leading blocks of the R runs. The implementation also maintains a *main merge heap* that is continuously merging leading blocks to produce items of the output run. All of R runs' data blocks in internal memory other than the R leading blocks are maintained in a queue of *placeholders*, called the *lookahead queue* LQ for reasons that will soon be clear. Each placeholder is a structure with a field *block_ptr* to store a pointer to a block in internal memory, a field *run_id* to store the identity of a run, and a field *block_num* to store the index of a block within a run. We use $LQ.\mathit{head}$ and $LQ.\mathit{tail}$ to denote the placeholders at the head and the tail of the queue LQ. For each disk $0 \leq d < D$, we maintain an *occupancy queue* OQ_d of elements. Each element is a pointer to some placeholder stored in the lookahead queue LQ. Elements in OQ_d always point to placeholders of blocks on disk d that are not yet in internal memory. We use $OQ_d.\mathit{head}$ and $OQ_d.\mathit{tail}$ to denote the elements at the head and tail of the queue OQ_d. By appending (resp., prepending) an element to OQ_d, we refer to the act of adding an element behind (resp., in front of) the element $OQ_d.\mathit{tail}$ (resp., $OQ_d.\mathit{head}$.) The lookahead queue as well as all the occupancy queues must be traversable in both directions. We use s_0, s_1, ..., s_{R-1} to denote the starting disks of the R input runs. Each starting disk is chosen randomly when the run is begun and is known to the merging algorithm.

The purpose of the lookahead queue LQ is to maintain prefetched input run blocks in the participation order. The purpose of each occupancy queue OQ_d, where $0 \leq d < D$, is to maintain (pointers to) placeholders corresponding to blocks from disk d in their participation order, so that blocks from disk d can be read by *Parallel_Read* operations in proper sequence.

Our implementation and SRM's properties ensure that the number[12] of elements in the lookahead queue LQ is never more than $\max\{m, RD\} + R + D$, and the number of elements in any occupancy queue cannot exceed $R + D$. Since elements of the LQ or any of the OQ_d's are $O(1)$ bytes in size, we can very simply implement LQ and all the OQ_d queues using statically allocated circular arrays in the obvious manner, with insignificant space overhead in practice.

Next we define some primitive operations in order to facilitate the presentation of our implementation:

1. *Lookahead*( ). The operation *Lookahead*( ) gets the next forecasting key from the forecasting heap and updates the forecasting heap appropriately, as discussed earlier. If the key so obtained predicts the participation time of the ith block in the rth run, then a new placeholder p with $p.\mathit{block\_num} := i$, $p.\mathit{run\_id} := r$, and $p.\mathit{block\_ptr} := \mathit{nil}$, is appended to LQ which makes $LQ.\mathit{tail} = p$. Finally, an element pointing to placefolder p is appended to OQ_d, where $d = (s_r + i) \bmod D$ is the disk on which the ith block of run r resides.

[12]Although the lookahead queue LQ can have R entries for each of the D disks, the number of blocks of internal memory actually in use can never exceed m.

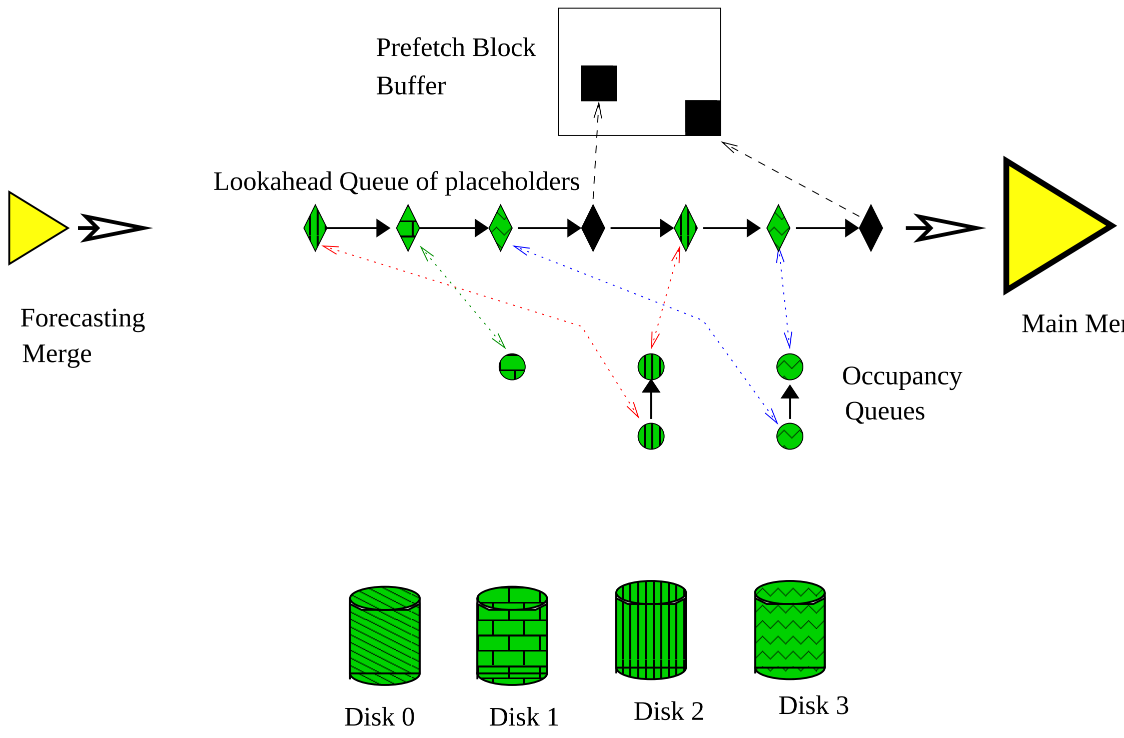

Figure 1: Implementation scheme for SRM: the lookahead queue contains placeholders for blocks in the participation order. A placeholder corresponding to an already prefetched blocks has a pointer to its block (shown in solid black are the two such placeholders); a placeholder corresponding to an unprefetched block of disk d appears in occupancy queue OQ_d. Placeholders in the OQs are in participation order. Each *Parallel_Read* operation reads in the block corresponding to placeholder $OQ_d.head$ for each d such that $0 \le d \le D-1$. Before a *Parallel_Read* operation, if any OQ_d is empty (e.g., OQ_0 here), we execute as many *Lookahead*( ) operations as are necessary, generating more placeholders, until OQ_d is nonempty.

2. *Parallel_Read*. The operation *Parallel_Read* issues read requests[13] for a set of at most D blocks, one per disk. The reads are carried out in *parallel*. *Parallel_Read* is *non-blocking* in the sense that it returns control immediately without waiting for the reads to complete. The precise block of disk d for which a read is issued is determined as follows: If the occupancy queue OQ_d is empty, no block is read from disk d. Otherwise, if $OQ_d.head$ points to placeholder p in LQ, the following is done:
   (a) Block $p.block\_num$ of run $p.run\_id$ is read from disk d into internal memory.
   (b) The field $p.block\_ptr$ is set to point to the newly read block in internal memory.
   (c) Element $OQ_d.head$ is removed from OQ_d.

4.3 Basic Ideas of Our Implementation

Figure 1 shows a schematic of our SRM implementation. Placeholders in LQ appear in the participation order of their respective blocks. In general, LQ may contain placeholders whose corresponding blocks remain to be fetched into memory; every such placeholder p has $p.block\_ptr = nil$. Blocks are "fed" to the main merge using LQ. Whenever the Bth item of leading block $Leading_r$, for some r such that $0 \le r \le R-1$, gets consumed by the main merge process, by definition the

[13]Our implementation uses memory-mapped I/O. The memory-mapped I/O calls we use are an enhanced version [ACG$^+$98] of the original memory-mapped calls provided by Digital Unix. The enhanced version sends off an asynchronous I/O request under the hood as soon as the call is made. We implement a D-disk parallel I/O operation by issuing D such calls for blocks on different disks.

block corresponding to placeholder $p' = LQ.head$ becomes the new leading block, and we remove p' from LQ. Our implementation guarantees the following *prefetching invariant*: Before the time t at which any block b begins participating in the main merge, a *Parallel_Read* operation prefetching (among $D-1$ other blocks) block b will already have been issued by our algorithm, and moreover, at time t the field $p'.block\_ptr$ of placeholder $p' = LQ.head$ points precisely to block b in memory. To accomplish this invariant, each *Parallel_Read* operation needs to read in, from each disk, the smallest unread block from that disk. The above invariant corresponds to Step 1c of the forecast and flush scheme of Section 3.1.

We use the D occupancy queues and *Lookahead*( ) operations (and hence the auxiliary merge being carried incrementally by the forecasting heap) in this context. Entries in occupancy queue OQ_d, for each d such that $0 \le d \le D-1$, correspond to unread blocks on disk d and are ordered according to the participation order of their blocks. Each *Parallel_Read* operation reads in the block corresponding to $OQ_d.head$, for each $0 \le d \le D-1$, thus reading in the smallest block from each disk. To guarantee that one block is indeed read from each disk, we must ensure that each OQ is non-empty before the *Parallel_Read* operation: This is accomplished by simply performing as many *Lookahead*( ) operations as necessary[14] until each OQ_d has at least one entry.

In order to maintain the prefetching invariant, we maintain a special variable, *marked* that remembers the placeholder of the smallest block for which a read was issued in the most recent *Parallel_Read* operation. Whenever the placeholder *marked* begins participating in the merge, we temporarily interrupt the main merge computation and employ the mechanism of the previous paragraph to ensure full parallelism for the *Parallel_Read* operation. Then we execute the *Parallel_Read* operation, update the value of *marked*, and resume the merging computation. Since *Parallel_Read* is non-blocking, our implementation performs I/O overlapped with computation. Sometimes, in the above process, a few blocks may need to be flushed but this too is easy to implement using our data structures: To flush f blocks, we simply traverse LQ from tail to head, find the first f placeholders $p_0, p_1, \ldots, p_{f-1}$ so found with $p_j.block\_ptr \neq nil$, and among other things set $p_j.block\_ptr := nil$.

4.4 Algorithmic Description

We are now in a position to give an algorithmic description of our implementation. For simplicity, we do not mention the I/O operations in the context of the forecasting heap, with the understanding that the implementor will choose an appropriate technique and perform the corresponding operations related to the auxiliary merge based upon the discussion in 4.1.

We use two variables *marked* and *next_marked* to record appropriate placeholders in the lookahead queue LQ. As mentioned earlier, the total number of blocks available at the start of the merge operation for input run blocks (including leading blocks, blocks currently being read into internal memory, and prefetched blocks) is assumed to be a number $m \ge 2R + D$. The output run has two DB item sized buffers that are used in the usual double-buffered fashion, with writes at full D-disk parallelism. Whenever an input run block gets depleted by the merge or when an input run block gets flushed, the number of free blocks increases by 1. If p is a placeholder in LQ, then we follow the convention that $p.block\_ptr$ is *nil* whenever the block $p.block\_num$ of run $p.run\_id$ is not in internal memory and is still on disk.

1. For each run $0 \le r < R$, read into internal memory its first block (on starting disk s_r). The pointers $Leading_0$, $Leading_1$, ..., $Leading_{R-1}$ are made to point to the corresponding first blocks. The total number of parallel I/O operations required to implement this step is equal to the maximum number of first blocks on any one disk.

[14]The precise number of *Lookahead*( ) operations needed to do so is unpredictable and dependent on the disk distribution of the relevant blocks and their participation order.

2. Insert the first key from each one of the R forecasting data runs into the forecasting heap. Insert the first item from each run into the main merge heap.

3. Initialize the lookahead queue LQ and the D occupancy queues OQ_d, for $0 \leq d < D$, to be empty queues.

4. While there exists at least one empty occupancy queue OQ_d and the forecasting heap is not empty, execute a *Lookahead*( ) operation.

5. Set $\mathit{marked} := LQ.\mathit{head}$.

6. [*Get ready for the next parallel read in the merge by flushing blocks if necessary.*] If there are at least D free blocks in internal memory, then proceed to Step 7. Otherwise let the number of free blocks in internal memory be $D - f$, where $f \geq 1$. Beginning with the tail of LQ, traverse LQ towards its head until f placeholders p_0, p_1, …, p_{f-1} are found such that $p_j.\mathit{block\_ptr} \neq \mathit{nil}$, for $0 \leq j < f$. Suppose that disk d_j is the disk from which the block $p_j.\mathit{block\_num}$ of run $p_j.\mathit{run\_id}$ originates. Then, for each $0 \leq j < f$,

   (a) Set $p_j.\mathit{block\_ptr} := \mathit{nil}$.

   (b) If the occupancy queue OQ_{d_j} is empty, insert an element pointing to the placeholder p_j into OQ_{d_j}; otherwise prepend an element pointing to placeholder p_j to the head of queue OQ_{d_j}.

   (c) If placeholder p_j is ahead of the placeholder *marked* in queue LQ, then set $\mathit{marked} := p_j$.

7. Execute a *Parallel_Read* operation[15].

8. While there exists at least one empty occupancy queue OQ_d and the forecasting heap is not empty, execute a *Lookahead*( ) operation.

9. Set $\mathit{next\_marked} := p'$, where p' is the placeholder in LQ closest to $LQ.\mathit{head}$ among the placeholders pointed to by elements $OQ_d.\mathit{head}$, for $0 \leq d < D$.

10. $\mathit{flag} := 0$.

11. While ($\mathit{flag} = 0$)

    (a) Generate the next item x from the main merge heap. Let r be the run containing x.

    (b) If run r has no more items to be merged, free the leading block $\mathit{Leading}_r$ and proceed to Step 11d. Otherwise, if the leading block $\mathit{Leading}_r$ of run r has just been depleted, free that block, set $\mathit{Leading}_r$ to point to the block $p.\mathit{block\_ptr}$, for placeholder $p = LQ.\mathit{head}$, and remove placeholder p from the lookahead queue LQ. If $p = \mathit{marked}$, set $\mathit{flag} := 1$.

    (c) Insert the next item from run r into the main merge priority queue. (If the leading block $\mathit{Leading}_r$ just changed in Step 11b above, the next item from run r is the first item of block $\mathit{Leading}_r$.)

    (d) Add item x to the output run buffer.

    (e) If adding x completes the block of an output run, add the key of x to the forecasting data run of the output run.

    (f) If the current output run buffer now has DB items, then switch output buffers and issue a non-blocking request to write out DB items to disk with full D-disk parallelism.

    (g) If the main merge heap is empty, set $\mathit{flag} := 2$.

12. If $\mathit{flag} = 1$, set $\mathit{marked} := \mathit{next\_marked}$ and loop back to Step 6.

[15]This operation may read into memory one or more blocks flushed in Step 6.

13. If *flag* $= 2$, write to disk the remaining items from the output run buffer and the merge is completed.

5 Performance Results

In this section we present the performance of SRM in practical scenarios and, in particular, compare its performance with that of an efficient implementation of DSM and demonstrate that SRM significantly outperforms DSM. We begin with a description of the computer system and programming environment of our implementations.

5.1 Computer System and Environment

Our experiments were carried out on a Digital Personal Workstation with a 500Mhz EV5.6 (21164A) CPU. We used six (i.e., $D = 6$) ST34501W Cheetah [Sea] disks for our experiments, two disks on each one of three Ultra-Wide SCSI buses attached to the system. The operating system was Digital Unix Version 4.0. In our experiments inputs varied in size from 100 MB to 1 GB and we use main memory in the range 15 MB to 24 MB. While the main memory we used in our experiments is small compared with main memory available on some computers today, our experiments are representative of the relative performances of the SRM and DSM techniques for larger inputs and main memories: In general since the SRM technique enables efficient merging of more runs at a time (with some caveats) thereby incurring a smaller number of passes than DSM, the advantage of SRM relative to DSM increases with increasing values of N/M and D.

Both the algorithms were implemented using the *Transparent Parallel I/O Environment (TPIE)* [TPI99] programming environment, which was originally developed by Darren Vengroff [Ven94] for his PhD and is currently being extended as part of an ongoing project at Duke University's Center for Geometric and Biological Computing. TPIE is a stream-oriented environment written in C++ designed to enable the implementation of efficient external memory algorithms on single and multiple disk systems. It provides basic building blocks for programmers to use while writing external memory programs. TPIE has built-in features such as a memory manager that manages buffers; it also keeps track of the amount of internal memory used by a program, which is very useful to control memory utilization during experiments as well as in memory management in general.

We implemented an interface for parallel disk streams striped in the usual roun-robin manner in units of logical blocks across the six disks. Each striped stream consists of one Unix file on each disk; each disk is a separately mounted filesystem. In order to facilitate randomized striping, our interface allows an application to begin striping on any disk of its choice. The I/O operations of TPIE used in our experiments were implemented using an enhanced version [ACG$^+$98] of memory-mapped I/O calls. A parallel I/O operation is simulated by six memory-mapped I/O calls, one to each disk. Each memory-mapped I/O call is a non-blocking call that instantaneously dispatches off an asynchronous I/O operation under the hood. In all our experiments, the size of the unified buffer cache was small enough (relative to the amount of data involved while sorting) so that effects from the buffering in the unified buffer cache were negligible.

5.1.1 Block Size

In our experiments, we used a logical block size of 256 KB; thus, all the memory-mapped I/O calls mapped regions of size 256 KB. It would be interesting to explore use of a smaller logical block size if we had control over disk block allocation, disk scheduling and so on, because we could use techniques as in [ZL96, ZL98] to achieve good I/O performance. Since we use filesystems and do not have such control, we can still ensure that the disk block allocation, readahead, and disk scheduling will be done efficiently by use of large block sizes in memory-mapped calls (hence our choice of the 256 KB size) and by making sure that each Unix file is accessed sequentially (so that filesystem readahead is triggered wherever possible). We set the block size to 256 KB for both DSM and SRM to allow proper comparisons in performance.

5.2 Input Characteristics

For all our experiments, we considered items of size $I = 104$ bytes, with keys of size $K = 8$ bytes and a block size $B = 2520$ items. The block size in bytes is therefore $104 \times 2520 = 262,080$ bytes, which is for all practical purposes 256 KB $= 262,144$ bytes. The unsorted input stream for each run of the two sorting algorithms was always a uniformly randomly generated sequence of items. *Both SRM and DSM, in our implementation, use the same run formation algorithm and so for a given internal memory size, both algorithms have to merge an identical number of runs during their merging phase.*

In general, the merging phase of an external mergesort takes time given by

$$\text{Number of Merge Passes} \times \text{Time per merge pass.}$$

While the first quantity above is known to be $\approx \lceil \log_R(N/M) \rceil$, the second quantity is a complicated function of various parameters and components of the I/O system (including block size) and the merge order R. The greater the merge order R, greater is the number of logically distinct streams accessed on each disk at any time, and greater is the time per merge pass. In our experiments, SRM outperforms DSM although it requires more time per merge pass.

Because of the randomization used by SRM, it is hard to construct a particular sequence of input records that brings out bad I/O performance in SRM's merging phase; indeed, the whole point of randomization is to ensure that no pathologically ordered input file can hinder the performance of SRM. Moreover the very nature of SRM ensures that its performance cannot degrade if some run suddenly gets consumed at higher rates relative to others; this is because all runs are striped and because each *Parallel_Read* operation always reads into internal memory the smallest block from each disk. We believe that skewed and nonuniformly randomly generated inputs cannot significantly change the performance characteristics of SRM (because of randomization) and DSM (because of striped reads.)

5.3 DSM and SRM Configurations

Each buffer of a striped-I/O parallel disk stream contains DB items. Since each striped-I/O stream uses double buffering, the amount of internal memory used by each striped-I/O parallel disk stream is 3.2 MB, which slightly larger than $2DBI$ bytes owing to some other implementation related overheads.

5.3.1 Run Formation and the Number of Runs Formed

The run formation stage of both SRM and DSM involves at most two striped-I/O parallel disk streams active at any time. Thus, the number of runs generated during run formation is determined by the amount of internal memory the algorithm is allowed to use, the amount of internal memory consumed by the buffers of a striped-I/O stream, and the amount of internal memory that TPIE reserves for program variables. In the rest of this section, we use the symbol U to denote the number of runs formed during the run formation stage of any given experiment.

5.3.2 Determining the Merge Order

During the merging pass, SRM and DSM use internal memory in very different ways. Given the same amounts of memory, the maximum merge order of an SRM merge operation is significantly larger than the maximum merge order of a DSM merge operation. During a merge of order R, DSM requires enough internal memory to have 3.2 MB sized buffers for each one of $R + 1$ streams. Thus the merge order for DSM is determined in a straightforward manner.

On the other hand, in order to carry out an R-way merge, SRM requires only one buffer of size 3.2 MB (corresponding to two buffers of size DB for its output run), $2R+D$ buffers of size 256 KB (corresponding to B), space for forecasting data and some other small per-run memory overheads. In all our experiments, all the forecasting data consumed a very small fraction of the total amount of internal memory available: It was always smaller than 100 KB, whereas the internal memory available in our two sets of experiments were $\approx$ 15 MB and $\approx$ 24 MB respectively. It is very often the case in SRM that there is a wide range of feasible values for R in which the resulting number $\lceil \log_R U \rceil$ of merging passes required to merge the initial U runs is the same optimal value. In such cases *we set SRM's merge order R to be equal to the smallest possible feasible value resulting in an optimal number $\lceil \log_R U \rceil$ of merging passes, but for practical reasons*[16] *we never set R higher than 19.* The advantage of using the smallest possible merge order is that the number of files involved in the merge is the smallest possible, which tends to keep the amount of disk latency incurred while merging as small as possible. Another advantage is that the average amount of internal memory space available per input run during a merge operation is increased, which has the effect of minimizing the I/O overhead ν (defined in Section 3) incurred in every merging pass.

Even though we try to keep the merge order of SRM as small as is possible, *the merge order of SRM merging operations is significantly greater than DSM merging operations.* Hence one expects that DSM merging passes will have higher disk locality and that SRM merging passes will incur more overhead relative to DSM merging passes on account of disk latency.

5.4 Performance Numbers and Graphs

In this section we report on two sets of experiments to compare the performance of SRM and DSM. In both cases, the input file size was varied in units of 1 million items ($\approx$ 100 MB) in the range from 1 million items ($\approx$ 100 MB) to 10 million items ($\approx$ 1 GB). In the first set of experiments, the amount of internal memory available to the sorting algorithm was 15 MB whereas in the second set of experiments it was 24 MB.

In Figure 2 and Tables 1 and 2, we present the performance numbers for the merging phases of both algorithms for the two sets of experiments. Table 1 is the table corresponding to experiments for internal memory size 15 MB, and Table 2 is the table corresponding to experiments for internal memory size 24 MB. Each data point in Tables 1 and 2 is based upon the average value obtained by conducting the same experiment five times with a different random input on each run. The graph in Figure 2 plots the average time in seconds required to complete the merging phase of SRM or DSM at a given data point. The tables provide additional insightful information. Of particular interest is the *average data streaming rate* during a merging phase, which is defined as the total amount of I/O (reads as well as writes) in bytes during the merging phase, divided by the time required to complete the merging phase.

Each table lists the total number U of runs formed during run formation, which is identical for both SRM and DSM. For DSM, the table lists the merge order R_{DSM} (each merge operation except possibly the last merge operation of a DSM merging phase has this merge order), the number $Passes_{\mathrm{DSM}}$ of passes, the time $Time_{\mathrm{DSM}}$ required to complete the merging phase, and the data streaming rate $Rate_{\mathrm{DSM}}$ attained by DSM during its merging phase. For SRM, for each data point, the table lists the number $Passes_{\mathrm{SRM}}$ of passes, the merge order R_{SRM} (of each SRM merging operation in the merging phase except possibly the last one), the time $Time_{\mathrm{SRM}}$ required to complete the merging phase, SRM's data streaming rate $Rate_{\mathrm{SRM}}$, and the I/O overhead ν corresponding to extra parallel read operations.

[16]We observed a system-specific threshold that causes a noticeable discontinuity in I/O performance characteristics of the file system when R exceeded 19.

	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$	$N=7$	$N=8$	$N=9$	$N=10$
U	12	24	36	47	59	71	82	94	106	117
$Passes_{\mathrm{DSM}}$	3	3	4	4	4	4	5	5	5	5
R_{DSM}	3	3	3	3	3	3	3	3	3	3
$Time_{\mathrm{DSM}}$(s)	18.3	35.7	70.8	95.8	122	141.2	198.4	226.2	254.8	284.2
$Rate_{\mathrm{DSM}}$(MB/s)	32.5	33.3	33.6	33.1	32.5	33.7	35.0	35.1	35.0	34.9
$Passes_{\mathrm{SRM}}$	1	2	2	2	2	2	2	2	2	2
R_{SRM}	12	5	6	7	8	9	10	10	11	11
$Time_{\mathrm{SRM}}$(s)	6.3	26.6	39.2	52.0	65.8	79.2	91.0	104.8	119.2	138.4
$Rate_{\mathrm{SRM}}$(MB/s)	31.5	29.8	30.4	30.5	30.2	30.1	30.5	30.3	30.0	28.7
ν	1.04	1.03	1.04	1.03	1.03	1.04	1.03	1.03	1.03	1.03

Table 1: Comparing SRM and DSM when internal memory is 15 MB and there are $D = 6$ disks. The input size N is in units of 1 million items, each of size 104 bytes.

	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$	$N=7$	$N=8$	$N=9$	$N=10$
U	7	14	20	27	33	40	46	53	60	66
$Passes_{\mathrm{DSM}}$	2	2	2	3	3	3	3	3	3	3
R_{DSM}	5	5	5	5	5	5	5	5	5	5
$Time_{\mathrm{DSM}}$(s)	11.5	21.2	32.4	63.0	79.6	95.0	110.6	127.2	143.2	158.2
$Rate_{\mathrm{DSM}}$(MB/s)	34.5	37.4	36.7	37.8	37.4	37.6	37.7	37.4	37.4	37.6
$Passes_{\mathrm{SRM}}$	1	1	2	2	2	2	2	2	2	2
R_{SRM}	7	14	5	6	6	7	7	8	8	9
$Time_{\mathrm{SRM}}$(s)	6.0	20.0	36.0	48.8	62.4	75.4	89.2	100.4	114.6	126.0
$Rate_{\mathrm{SRM}}$(MB/s)	33.1	19.8	33.1	32.5	31.8	31.6	31.1	31.6	31.2	31.5
ν	1.00	1.00	1.00	1.01	1.01	1.01	1.01	1.01	1.01	1.01

Table 2: Comparing SRM and DSM when memory is 24 MB and there are $D = 6$ disks. The input size N is in units of 1 millions items, each of size 104 bytes.

5.5 Relative Performance Comparisons

SRM significantly outperforms DSM in the experiments. For the experiments with internal memory size 15 MB, SRM's performance is better by a margin of almost 50%. SRM's margin of improvement is less impressive with internal memory size 24 MB; but the improvement is still in the 25% ballpark for input sizes larger than $N = 4$ million items. There is one data point ($N = 3$ million items, with memory size of 24 MB) at which DSM is actually marginally better than SRM; this happens to be the only point in all our experiments in which DSM and SRM require the same number of passes. The main advantage of SRM over DSM is that SRM allows efficient external merging of $\Omega(m)$ runs irrespective of the number of disks in the I/O system. Our observations are consistent with theory in that they indicate that when the number of disks is fixed, as m increases, the relative performance gain of SRM over DSM diminishes.

The average data streaming rate of DSM is consistently better than that of SRM, as anticipated, but SRM outperforms DSM because of its smaller number of passes. When we compare SRM's streaming rate for 15 MB with its streaming rate for 24 MB, we see an overall improvement in streaming rate for the larger internal memory size, since the number of runs is reduced and SRM's merge order tends to be smaller. We are not able to explain the improvement in the streaming rate of DSM's performance in the experiments with 24 MB relative to its performance in the experiments with 15 MB; the improvement is somewhat surprising because DSM's merge order increases from 3

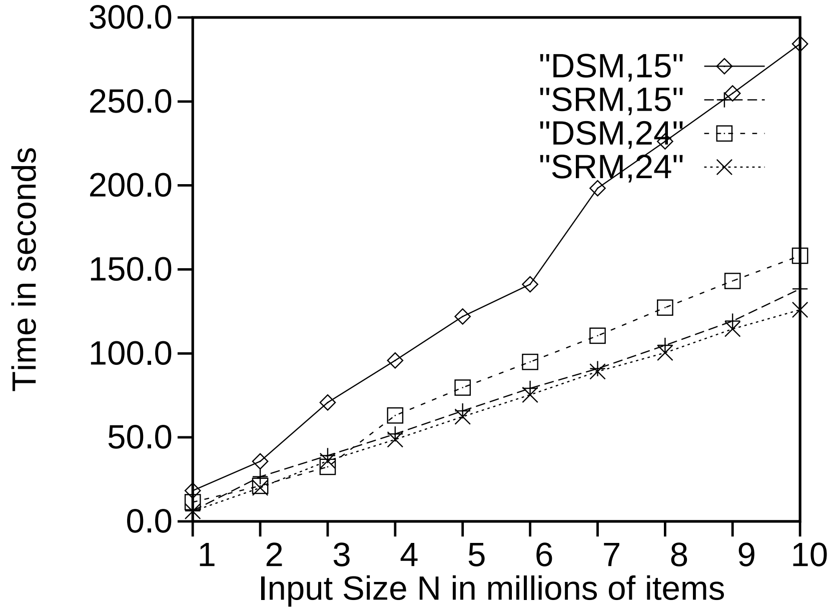

Figure 2: Merging phase timings of SRM and DSM.

to 5 when the internal memory is increased.

The I/O overhead ν in the total number of *Parallel_Read* operations required by SRM is small, very close to 1, as expected on the basis of previous analysis [BGV97]. This suggests to us that the elapsed time performance is not hindered by the flush operations incurred by SRM; if the implementation of the *Parallel_Read* can be improved, which we think possible, we can get further improvement of SRM's elapsed time performance. We briefly mention possible approaches to improve the performance of each parallel I/O operation in Section 5.6.

One interesting observation regarding our data points is the sudden drop in streaming rate when the merge order becomes 14, as in the case with $N = 2$ million items and an internal memory size of 24 MB. In this case, the streaming rate is 22 MB/sec, although a streaming rate of almost 30 MB/sec is possible for $R = 11$, $N = 9$ million items and $N = 10$ million items, and internal memory size of 15 MB. We were not able to account for the sudden drop in streaming rate; some preliminary experiments indicate that the most important reason for the sudden drop in streaming rate may not be the increased seeking but some other effects perhaps related to the number of files opened by the application at any time.

5.6 Improving I/O Performance

SRM can be made to perform even better if the implementation of the parallel I/O is improved. The hardware being used (the CPU and the I/O system) and our experience with parallel I/O systems [BSG$^+$99] suggest that there is scope for improving the performance of parallel I/O operations in our implementation, thereby improving the streaming data rates. Possible techniques to improve performance when we have control over disk block allocation, disk scheduling etc. (as discussed in Section 5.1.1) include controlling the layout of blocks of runs and carefully planning the sequence in which blocks from each disk can be read into memory [ZL96, ZL98]. Another approach that may help in improving I/O performance is to use techniques such as the one in [BSG$^+$99] which exploit the readahead mechanism used by disk drive controllers to load data into their track buffers. A simple high-level approach that may result in improved performance is if we split the set of disks into two sets and then used one set to store input runs and the other for output runs, swapping their role after each merge pass; this approach ensures that writes and reads do not interfere during the external merging process.

6 Other Applications

In this section we briefly mention other situations in which the data structures and techniques we developed to implement SRM can be used fruitfully.

6.1 Distribution Sort and Multi-way Partitioning

Consider a distribution sort or the partitioning of a stream into several other streams on an I/O system with D disks. Such a distribution/partitioning type of computation may be required as part of some other database operation, for example in a hash join. We could envisage using striped I/O to ensure perfect parallelism on all D disks. However, use of striped I/O would mean that the number of streams into which an input stream can be distributed using internal memory size M would be $O(M/DB)$; when the number of partitions or buckets desired is large, many distribution passes would be required. In this situation, just like it was desirable to merge $O(M/B)$ runs at a time during external mergesort, it is desirable to be able to partition an input stream into $O(M/B)$ streams. A randomized striping of the R output streams and $2R$ internal memory blocks will help in implementing an R-way distribution. In such a scheme, data structures and mechanisms similar to the ones we developed in this paper can be used: For instance, blocks destined to go to disk d can be queued up in queue OQ_d, and a *Parallel_Write* analogous to *Parallel_Read* can write the block corresponding to the head of the OQ_d queue appropriately to disk d.

6.2 Streaming through Multimedia Files

The problem of external merging of streams that are striped across disks is similar in terms of access patterns to a video server that has to stream through multiple streams that are striped across disks. The nonuniformity of the rates at which runs get depleted is similar to the nonuniformity of streaming rates owing to different compression rates for different frames in the stream. In both cases though, the rates of streaming required are partially predictable, albeit to a limited extent. While merging, the forecasting keys predict the participation order of blocks of a merge. Consider using a file of timestamps, where each timestamp corresponds to the time at which a block from that stream must be in internal memory. A file of such timestamps corresponding to a video stream is analogous to the forecasting data run corresponding to a run. Hence, the techniques we developed to implement the lookahead mechanism and the forecast and flush buffer management and prefetching scheme can now be analogously implemented using the timestamps to predict the time at which a block must be in memory. Our data structures and techniques may thus have applications to video servers, although substantial modifications may be needed to implement the real time aspect of video servers.

7 Conclusions and Future Work

In this paper we considered the important problem of external sorting in a synchronous parallel disk setting. We have proposed simple and elegant data structures and techniques to implement the SRM mergesort algorithm for parallel disks. To our knowledge, this is the first practical implementation of a parallel disk sorting algorithm that performs a provably optimal number of passes. Our simplified implementation of SRM includes a novel technique to implement a lookahead mechanism using forecasting keys. Our implementation significantly outperforms the popular double-buffered disk-striped mergesort (DSM) technique. Although each merging pass of DSM occurs a little faster than that of SRM, the smaller number of passes required by SRM makes SRM's overall performance better than that of DSM. Our techniques are also applicable to other streaming operations in databases.

In future work, we hope to improve the implementation of parallel I/O operations in TPIE and allow asynchronous I/O, thus getting an improvement in the elapsed time performance of SRM. We also plan to implement a parallel disk distribution sort in which the distribution is done by the approach of Section 6.1, analogous to the merge process of SRM executed in "reverse". In the envisioned parallel external distribution operation, the input stream is implemented using striped I/O, whereas the output stream requires parallel independent disk accesses. Comparing the performance of such a sort with SRM should be particularly interesting because disk drives may be somewhat better at performing the kind of I/O needed for distribution compared with merging.

Acknowledgments

The authors wish to thank Lars Arge and the anonymous referees for helpful discussions and suggestions.

References

[ACG+98] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *1998 Usenix Technical Conference*, pages 143–154, June 1998.

[ADADC+97] Andrea C. Arpaci-Dussea, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 243–254, 1997.

[AP94] 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, pages 659–668, 1994.

[AV88] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[BGV97] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[BSG+99] 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*, pages 83–92, Atlanta, GA, May 1999.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 6, pages 139–149, January 1995.

[CH96] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the parallel disk model with the vic* implementation. Technical Report PCS-TR96-293, Dept. of Computer Science, Dartmouth College, August 1996.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[ECW94] Vladimir Estivill-Castro and Derrick Wood. Foundations of external merging. In *Foundations of Software Technology and Theoretical Computer Science*, volume 14, pages 414–425, 1994.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GTVV93] 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, pages 714–723, Palo Alto, November 1993.

[GVW96] 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.

[IBM90] IBM database 2, administration guide for common servers, June 1990.

[Knu98] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 2nd edition, 1998.

[PSV94] V. S. Pai, A. A. Schaffer, and P. J. Varman. Markov analysis of multiple-disk prefetching strategies for external merging. *Theoretical Computer Science*, 128(1–2):211–239, June 1994.

[Sal89] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[Sea] Seagate Technology st-34501w/wc ultra-SCSI wide (Cheetah 4lp) data sheet. ftp://ftp.seagate.com/techsuppt/scsi/st34501w.txt.

[TPI99] TPIE user manual and reference, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[Ven94] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proceedings of the DAGS Symposium on Parallel Computation*, volume 3, pages 117–134, July 1994.

[Vit01] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001. Available electronically via the author's web page `http://www.cs.duke.edu/~jsv/`.

[VN93] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

[VS94] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[ZL96] L. Q. Zheng and P.-A. Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.

[ZL98] Weiye Zhang and P.-A. Larson. Buffering and read-ahead strategies for external mergesort. *Proceedings of the International Conference on Very Large Databases*, 24:523–532, 1998.