Alcom-FT Technical Report Series
ALCOMFT-TR-01-79 (revised July 2, 2001)

Duality Between Prefetching and Queued Writing with Parallel Disks

David A. Hutchinson[1]*, Peter Sanders[2]**, Jeffrey Scott Vitter[1]***

[1] Department of Computer Science, Duke University, Durham, NC 27708–0129
{hutchins,jsv}@cs.duke.edu
[2] Max-Planck-Institute for Computer Science, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, sanders@mpi-sb.mpg.de

Abstract. Parallel disks promise to be a cost effective means for achieving high bandwidth in applications involving massive data sets, but algorithms for parallel disks can be difficult to devise. To combat this problem, we define a useful and natural duality between writing to parallel disks and the seemingly more difficult problem of prefetching. We first explore this duality for applications involving read-once accesses using parallel disks. We get a simple linear time algorithm for computing optimal prefetch schedules and analyze the efficiency of the resulting schedules for randomly placed data and for arbitrary interleaved accesses to striped sequences. Duality also provides an optimal schedule for the integrated caching and prefetching problem, in which blocks can be accessed multiple times. Another application of this duality gives us the first parallel disk sorting algorithms that are provably optimal up to lower order terms. One of these algorithms is a simple and practical variant of multiway merge sort, addressing a question that has been open for some time.

1 Introduction

External memory (EM) algorithms are designed to be efficient when the problem data do not fit into the high-speed random access memory (RAM) of a computer and therefore must reside on external devices such as disk drives [17]. In order to cope with the high latency of accessing data on such devices, efficient EM algorithms exploit locality in their design. They access a large *block* of B contiguous data elements at a time and perform the necessary algorithmic operations on the elements in the block while in the high-speed memory. The speedup can be significant. However, even with blocked access, a single disk provides much less bandwidth than the internal memory. This problem can be mitigated by using multiple disks in parallel. For each input/output operation, one block is

* Supported in part by the NSF through research grant CCR–0082986.
** Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT)
*** Supported in part by the NSF through research grants CCR–9877133 and EIA–9870724 and by the ARO through MURI grant DAAH04–96–1–0013

transferred between memory and each of the D disks. The algorithm therefore transfers D blocks at the cost of a single-disk access delay.

A simple approach to algorithm design for parallel disks is to employ large logical blocks, or *superblocks* of size $B \cdot D$ in the algorithm. A superblock is split into D physical blocks—one on each disk. We refer to this as *superblock striping.* Unfortunately, this approach is suboptimal for EM algorithms like sorting that deal with many blocks at the same time. An optimal algorithm for sorting and many related EM problems requires *independent access* to the D disks, in which each of the D blocks in a parallel I/O operation can reside at a different position on its disk [19, 17]. Designing algorithms for independent parallel disks has been surprisingly difficult [19, 14, 15, 3, 8, 9, 17, 16, 18].

In this paper we consider parallel disk output and input separately, in particular as the *output scheduling problem problem* and the *prefetch scheduling problem* respectively. The (online) *output scheduling (or queued writing) problem* takes as input a fixed size pool of m (initially empty) memory buffers for storing blocks, and the sequence $\langle w_0, w_1, \ldots, w_{L-1} \rangle$ of block *write requests* as they are issued. Each write request is labeled with the disk it will use. The resulting schedule specifies when the blocks are output. The buffer pool can be used to reorder the outputs with respect to the logical writing order given by Σ so that the total number of output steps is minimized.

The (offline) *prefetch scheduling problem* takes as input a fixed size pool of m (empty) memory buffers for storing blocks, and the sequence $\langle r_0, r_1, \ldots, r_{L-1} \rangle$ of distinct block *read requests* that will be issued. Each read request is labeled with the disk it will use. The resulting *prefetch schedule* specifies when the blocks should be fetched so that they can be consumed by the application in the right order.

The central theme in this paper is the newly discovered duality between these two problems. Roughly speaking, an output schedule corresponds to a prefetch schedule with reversed time axis and vice versa. We illustrate how computations in one domain can be analyzed via duality with computations in the other domain.

Sect. 2 introduces the duality principle formally for the case of distinct blocks to be written or read (*write-once* and *read-once* scheduling). Then Sect. 3 derives an optimal write-once output scheduling algorithm and applies the duality principle to obtain an optimal read-once prefetch scheduling algorithm.

Even an optimal schedule might use parallel disks very inefficiently because for difficult inputs most disks might be idle most of the time. In Sect. 4 we therefore give performance guarantees for randomly placed data and for arbitrarily interleaved accesses to a number of data streams. In particular, we discuss the following allocation strategies:

Fully Randomized (FR): Each block is allocated to a random disk.
Striping (S): Consecutive blocks of a stream are allocated to consecutive disks in a simple, round-robin manner.
Simple Randomized (SR): Striping where the disk selected for the first block of each stream is chosen randomly.

Randomized Cycling (RC): Each stream i chooses a random permutation π_i of disk numbers and allocates the j-th block of stream i on disk $\pi_i(j \bmod D)$.

In Sect. 5 we relax the restriction that blocks are accessed only once and allow repetitions (*write-many* and *read-many* scheduling). Again we derive a simple optimal algorithm for the writing case and obtain an optimal algorithm for the reading case using the duality principle. A similar result has recently been obtained by Kallahalla and Varman [11] using more complicated arguments.

Finally, in Sect. 6 we apply the results from Sects. 3 and 4 to parallel disk sorting. Results on online writing translate into improved sorting algorithms using the distribution paradigm. Results on offline reading translate into improved sorting algorithms based on multi-way merging. By appending a 'D' for distribution sort or an 'M' for mergesort to an allocation strategy (FR, S, SR, RC) we obtain a descriptor for a sorting algorithm (FRD, FRM, SD, SM, SRD, SRM, RCD, RCM). This notation is an extension of the notation used in [18]. RCD and RCM turn out to be particularly efficient. Let

$$\mathrm{Sort}(N) = \frac{N}{DB}\left(1 + \log_{\frac{M}{B}} \frac{N}{M}\right)$$

and note that $2 \cdot \mathrm{Sort}(N)$ appears to be the lower bound for sorting N elements on D disks [1]. Our versions of RCD and RCM are the first algorithms that provably match this bound up to a lower order term $\mathcal{O}(BD/M)\,\mathrm{Sort}(N)$. The good performance of RCM is particularly interesting. The question of whether there is a simple variant of mergesort that is asymptotically optimal has been open for some time.

Related Work

Prefetching and caching has been intensively studied and can be a quite difficult problem. Belady [5] solves the caching problem for a single disk using our machine model. Cao et al. [7] propose a model that additionally allows overlapping of I/O and computation. Albers et al. [2] were the first to find an optimal polynomial time offline algorithm for the single-disk case in this model but it does not generalize well to multiple disks. Kimbrel and Karlin [12] devised a simple algorithm called *reverse aggressive* that obtains good approximations in the parallel disk case if the buffer pool is large and the failure penalty F is small. However, in our model, which corresponds to $F \to \infty$, the approximation ratio that they show goes to infinity. Reverse aggressive is very similar to our algorithm so that it is quite astonishing that the algorithm is optimal in our model. Kallahalla and Varman [10] studied online prefetching of read-once sequences for our model. They showed that very large lookahead $L \gg mD$ is needed to obtain good competitiveness against an optimal offline algorithm. They proposed an $\mathcal{O}(L^2D)$ time algorithm with this property, and yielding optimal schedules for the offline case. A practical disadvantage of this algorithm is that some blocks may be fetched and discarded several times before they can be delivered to the application.

There is less work on performance guarantees. A (slightly) suboptimal writing algorithm is analyzed in [16] for FR allocation and extended to RC-allocation in [18]. These results are the basis for our results in Sect. 4. For reading there is an algorithm for SR allocation that is close to optimal if $m \gg D \log D$ [3].

There are asymptotically optimal deterministic algorithms for external sorting [15], but the constant factors involved make them unattractive in practice. Barve et al. [3] introduced a simple and efficient randomized sorting algorithm called *Simple Randomized Mergesort (SRM).* For each run, SRM allocates blocks to disks using the SR allocation discipline. SRM comes within $\gamma \cdot \mathrm{Sort}(N)$ of the apparent lower bound if $M/B = \Omega\left(D \log(D)/\gamma^2\right)$ but for $M = o(D \log D)$ the bound proven is not asymptotically optimal. It was an open problem whether SRM or another variant of striped mergesort could be asymptotically optimal for small internal memory. Knuth [13, Exercise 5.4.9-31] gives the question of a tight analysis of SR a difficulty of 48 on a scale between 0 and 50.

To overcome the apparent difficulty of analyzing SR, Vitter and Hutchinson [18] analyzed RC allocation, which provides more randomness but retains the advantages of striping. RCD is an asymptotically optimal distribution sort algorithm for multiple disks that allocates successive blocks of a bucket to the disks according to the RC discipline and adapts the approach and analysis of Sanders, Egner, and Korst [16] for write scheduling of blocks. However, the question remained whether such a result can be obtained for mergesort and how close one can come to the lower bound for small internal memory.

2 The Duality Principle

Duality is a quite simple concept once the model is properly defined. Therefore, we start with a more formal description of the model:

Our machine model is the parallel disk model of Vitter and Shriver [19] with a single[1] processor, D disks and an internal memory of size M. All blocks have the same size B. In one *I/O step*, one block on each disk can be accessed in a synchronized fashion. We consider either a queued writing or a buffered prefetching arrangement, where a pool of m block buffers is available to the algorithm (see Fig. 1).

A *write-once output scheduling problem* is defined by a sequence $\Sigma = \langle b_0, \ldots, b_{L-1} \rangle$ of distinct blocks. Let $\mathrm{disk}(b_i)$ denote the disk on which block b_i is located. An application process *writes* these blocks in the order specified by Σ. We use the term *write* for the logical process of moving a block from the responsibility of the application to the responsibility of the scheduling algorithm. The scheduling algorithm orchestrates the physical *output* of these blocks to disks. An output schedule is specified by giving a function $\mathrm{oStep} : \{b_0, \ldots, b_{L-1}\} \to \mathbb{N}$ that specifies for each disk block $b_i \in \Sigma$ the time step when it will be output. An output schedule is *correct* if the following conditions hold: (i) No disk is referenced more than once in a single time step, i.e., if $i \neq j$ and $\mathrm{disk}(b_i) = \mathrm{disk}(b_j)$

[1] A generalization our results to multiple processors is relatively easy as long as data exchange between processors is much faster than disk access.

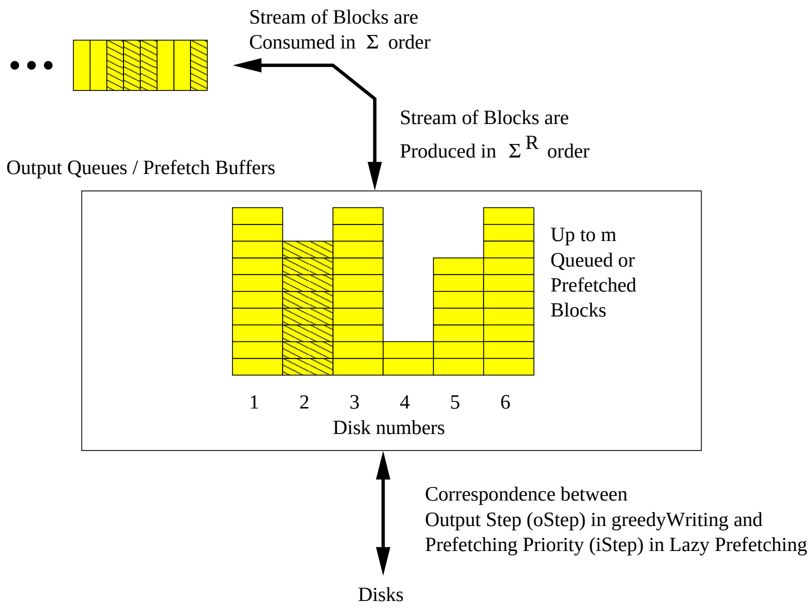

Fig. 1. Duality between the prefetching priority and the output step. The hashed blocks illustrate how the blocks of disk 2 might be distributed.

then $\mathrm{oStep}(b_i) \neq \mathrm{oStep}(b_j)$. (ii) The buffer pool is large enough to hold all the blocks b_j that are written before a block b_i but not output before b_i, i.e.,

$$\forall 0 \leq i < L : \mathrm{o}\mathcal{B}\mathrm{acklog}(b_i) := |\{j < i : \mathrm{oStep}(b_j) \geq \mathrm{oStep}(b_i)\}| < m \ .$$

The number of steps needed by an output schedule is $T = \max_{0 \leq i < L} \mathrm{oStep}(b_i)$. A schedule is *optimal* if it minimizes T among all correct schedules.

It will turn out that our write-once output scheduling algorithms even work if they are given the blocks *online*, i.e., one at a time without specifying Σ explicitly.

A *read-once prefetch scheduling problem* is defined analogously. Now *reading* means the logical process of moving a block from the responsibility of the scheduling algorithm to the application and *fetching* means the physical disk access. A prefetch schedule is defined using a function $\mathrm{iStep} : \{b_0, \ldots, b_{L-1}\} \rightarrow \mathbb{N}$. The limited buffer pool size requires the correctness condition

$$\forall 0 \leq i < L : \mathrm{i}\mathcal{B}\mathrm{acklog}(b_i) := |\{j > i : \mathrm{iStep}(b_j) \leq \mathrm{iStep}(b_i)\}| < m$$

(all blocks b_j that are fetched no later than a block b_i but consumed after b_i must be buffered).

It will turn out that our prefetch scheduling algorithms work *offline*, i.e., they need to know Σ in advance.

The following theorem shows that the reading and writing not only have similar models but are equivalent to each other in a quite interesting sense:

Theorem 1. **(Duality Principle)** *Consider any sequence $\Sigma = \langle b_0, \ldots, b_{L-1}\rangle$ of distinct write requests. Let* oStep *denote a correct output schedule for Σ that uses T output steps. Then we get a correct prefetch schedule* iStep *for $\Sigma^R = \langle b_{L-1}, \ldots, b_0\rangle$ that uses T fetch steps by setting* $\text{iStep}(b_i) = T - \text{oStep}(b_i) + 1$.

Vice versa, every correct prefetch schedule iStep *for Σ^R that uses T fetch steps yields a correct output schedule* $\text{oStep}(b_i) = T - \text{iStep}(b_i) + 1$ *for Σ, using T output steps.*

Proof. For the first part, consider $\text{iStep}(b_i) = T - \text{oStep}(b_i) + 1$. The resulting fetch steps are between 1 and T and all blocks on the same disk get different fetch steps. It remains to show that $\text{i}\mathcal{B}\text{acklog}(b_i) < m$ for $0 \le i < L$. With respect to Σ^R, we have

$$\begin{aligned}\text{i}\mathcal{B}\text{acklog}(b_i) &= |\{j < i : \text{iStep}(b_j) \le \text{iStep}(b_i)\}| \\ &= |\{j < i : T - \text{oStep}(b_j) + 1 \le T - \text{oStep}(b_i) + 1\}| \\ &= |\{j < i : \text{oStep}(b_j) \ge \text{oStep}(b_i)\}| \quad .\end{aligned}$$

the latter value is $\text{o}\mathcal{B}\text{acklog}(b_i)$ with respect to Σ and hence smaller than m.

The proof for the converse case is completely analogous. □

3 Optimal Write-Once and Read-Once Scheduling

We give an optimal algorithm for writing a write-once sequence, prove its optimality and then apply the duality principle to transform it into a read-once prefetching algorithm.

Consider the following algorithm *greedyWriting* for writing a sequence $\Sigma = \langle b_0, \ldots, b_{L-1}\rangle$ of distinct blocks. *Let Q denote the set of blocks in the buffer pool, so initially $Q = \emptyset$. Let $Q_d = \{b \in Q : \text{disk}(b) = d\}$. Write the blocks b_i in sequence as follows: (1) If $|Q| < m$ then simply insert b_i into Q. (2) Otherwise, each disk with $Q_d \ne \emptyset$ outputs the block in Q_d that appears first in Σ. The blocks output are then removed from Q and b_i is inserted into Q. (3) Once all blocks are written the queues are flushed, i.e., additional output steps are performed until Q is empty,*

Any schedule where blocks are output in arrival order on each disk, is called a *FIFO schedule*. The following lemma tells us that it is sufficient to consider FIFO schedules when we look for optimal schedules. The proof is based on transforming a non-FIFO schedule into a FIFO schedule by exchanging blocks in the schedule of a disk that are output out of order.

Lemma 1. *For any sequence of blocks Σ and every correct output schedule* oStep′ *there is a FIFO output schedule* oStep *consisting of at most the same number of output steps.*

Algorithm greedyWriting is one way to compute a FIFO schedule. The following lemma shows that greedyWriting outputs every block as early as possible.

Lemma 2. *For any sequence of blocks Σ and any FIFO output schedule* oStep', *let* oStep *denote the schedule produced by algorithm greedyWriting. Then for all* $b_i \in \Sigma$, *we have* $\mathrm{oStep}(b_i) \leq \mathrm{oStep}'(b_i)$.

Proof. (Outline) The proof is by induction over the number of blocks. There are two nontrivial cases. One case corresponds to the situation where the output step of a block immediately follows an output of a previous block on the same disk. The other case corresponds to the situation where no earlier step is possible because otherwise its $\mathrm{o}\mathcal{B}\mathrm{acklog}$ would be too large.

Combining Lemmas 1 and 2 we see that greedyWriting gives us optimal schedules for write-once sequences:

Theorem 2. *Algorithm greedyWriting gives a correct, minimum length output schedule for any write-once reference sequence Σ.*

Combining the duality principle and the optimality of greedyWriting, we get an optimal algorithm for read-once prefetching that we call *lazy prefetching*:

Corollary 1. *An optimal prefetch schedule* iStep *for a sequence Σ can be obtained by using greedyWriting to get an output schedule* oStep *for Σ^R and setting* $\mathrm{iStep}(b_i) = T - \mathrm{oStep}(b_i) + 1$.

Note that the schedule can be computed in time $\mathcal{O}(L + D)$ using very simple data structures.

4 How Good is Optimal?

When we are processing several streams concurrently, the knowledge that we have an optimal prefetching algorithm is often of little help. We also want to know "how good is optimal?" In the worst case, all requests may go to the same disk and no prefetching algorithm can cure the dreadful performance caused by this bottleneck. However, the situation is different if blocks are allocated to disks using striping, randomization[2] or both.

Theorem 3. *Consider a sequence of L block requests. and a buffer pool of size $m \geq D$ blocks. The number of I/O steps needed by greedyWriting or lazy prefetching is given by the following bounds, depending on the allocation discipline. For striping and randomized cycling, an arbitrary interleaving of sequential accesses to S sequences is allowed.*

$$\begin{aligned} \textit{Striping: } & \tfrac{L}{D} + S, \quad \textit{if } m > S(D-1); \\ \textit{Fully Random (FR): } & \left(1 + \mathcal{O}\left(\tfrac{D}{m}\right)\right)\tfrac{L}{D} + \mathcal{O}\left(\tfrac{m}{D}\log m\right) \quad \textit{(expected)}; \\ \textit{Randomized Cycling (RC): } & \left(1 + \mathcal{O}\left(\tfrac{D}{m}\right)\right)\tfrac{L}{D} + \min\left\{S + \tfrac{L}{D}, \tfrac{m}{D}\log m\right\} \textit{ (expected)} \end{aligned}$$

[2] In practice, this will be done using simple hash functions. However, for the analysis we assume that we have a perfect source of randomness.

Proof. (Outline) The bound for striped writing is based on the observation that $L/D + S$ is the maximum number of blocks to be handled by any disk and that the o$\mathcal{B}$acklog of any block can never exceed m if $m > S(D-1)$.

For fully random placement the key idea is that greedyWriting dominates the "throttled" algorithm of [16], which admits only $(1 - \Theta(D/M))D$ blocks per output step into the queues.

The bound for RC is a combination of the two previous bounds. The bound for FR applies to RC writing using the observation of [18] that the throttled algorithm of [16] performs at least as well for RC as for FR.

The results for writing transfer to offline prefetching via duality. For the RC case we also need the observation that the reverse of a sequence using RC is indistinguishable from a nonreversed sequence. □

For writing, the trailing additive term for each case enumerated in Theorem 3 can be dropped if the final contents of the buffer pool is not flushed.

5 Integrated Caching and Prefetching

We now relax the condition that the read requests in Σ are for distinct blocks, permitting the possibility of saving disk accesses by keeping previously accessed blocks in memory. For this *read-many* problem, we get a tradeoff for the use of the buffer pool because it has to serve the double purposes of keeping blocks that are accessed multiple times, and decoupling physical and logical accesses to equalize transient load imbalance of the disks. We define the *write-many* problem in such a way that the duality principle from Theorem 1 transfers: *The latest instance of each block must be kept either on its assigned disk, or in the buffer pool. The final instance of each block must be written to its assigned disk.*[3]

We prove that the following offline algorithm *manyWriting* minimizes the number of output operations for the write-many problem: *Let Q and Q_d be defined as for greedyWriting. To write block b_i, if $b_i \in Q$, the old version is overwritten in its existing buffer. Otherwise, if $|Q| < m$, b_i is inserted into Q. If this also fails, an output step is performed before b_i is inserted into Q. The output analogue of Belady's* MIN *rule [5] is used on each disk, i.e., each disk with $Q_d \neq \emptyset$ outputs the block in Q_d that is accessed again farthest in the future.*

Applying duality, we also get an optimal algorithm for integrated prefetching and caching of a sequence Σ: using the same construction as in Cor. 1 we get an optimal prefetching and caching schedule. It remains to prove the following theorem:

Theorem 4. *Algorithm manyWriting solves the* write-many problem *with the fewest number of output steps.*

[3] The requirement that the latest versions have to be kept might seem odd in an offline setting. However, this makes sense if there is a possibility that there are reads at unknown times that need an up-to-date version of a block.

Proof. We generalize the optimality proof of Belady's algorithm by Borodin and El-Yaniv [6] to the case of writing and multiple disks. Let $\Sigma = \langle b_0, \ldots, b_{L-1} \rangle$ be any sequence of blocks to be written. The proof is based on the following claim.

Claim: Let ALG be any algorithm for cached writing. Let d denote a fixed disk. For any $0 \leq i < L$ it is possible to construct an offline algorithm ALG_i that satisfies the following properties: (i) ALG_i processes the first $i-1$ writes exactly as ALG does. (ii) ALG_i takes no more steps than ALG. (iii) If block b_i is the first block written after output step s, then immediately before output step s we had $b_i \notin Q$ and $|Q| = m$. (iv) If b_i is is the first block written after output step s, then ALG_i performs this output according to the MIN rule on disk d.

Once this claim is established, the theorem can be proven as follows: Starting with an optimal offline algorithm OPT, we apply the claim with $i = 0$ and $d = 0$ to obtain another optimal algorithm OPT_0, then apply the claim with $i = 1$ and $d = 0$ to obtain OPT_1 and so on. By induction over i, it can be seen that OPT_{L-1} never leaves unused buffer frames before an output step and uses MIN for deciding which blocks to output on disk 0. We repeat this game for each disk and finally obtain an optimal algorithm that works like manyWriting on all disks.

It remains to prove the claim. We first set $\text{ALG}_i = \text{ALG}$. Now properties (i) and (ii) hold whereas properties (iii) and (iv) are violated. We now apply further transformations that preserve the smaller numbered properties until all prperties hold.

If property (iii) is violated by the current ALG_i, then b_i is the first block the new ALG_i writes after step s. In the new ALG_i, b_i is the last block written before step s. Note that this maintains the order in which block are written and properties (i) and (ii) are maintained. If property (iii) is still violated, this transformation is repeated until propterty (iii) holds. This process must terminate at the latest when b_i is written before the first output step.

If properties (i)–(iii) hold for ALG_i but property (iv) is violated, we define a new version ALG'_i that fulfills property (iv) as follows. We use $X + b$ as a shorthand for $X + \{b\}$ for a set of blocks X and a block b. ALG'_i works identically to ALG_i until the first $i - 1$ blocks are written. In the subsequent output step s, ALG'_i follows the MIN rule for disk d. It remains to define the behavior of ALG'_i after step s so that property (ii) is also maintained.

If during output step s ALG_i outputs block b' on disk d and ALG_i outputs block b on disk d then $b \neq b'$ and the buffer pool of ALG_i can be written as $Q = X + b$ whereas ALG'_i has $Q = X + b'$. Until b is written again (or until the end if b is not written again), ALG'_i mimics ALG_i except for outputting b' if ALG_i should output b. If the latter happens, both algorithms have the same state of Q again and subsequently, ALG'_i can completely follow ALG_i to maintain property (iv).

By definition of the MIN strategy, b' will be written again before b. Consider the state of Q of ALG_i and ALG'_i after such a write of b'. The pool of ALG_i can be written as $Q = Y + b + b'$ and the pool of ALG'_i as $Q = Y + b' + \varepsilon$ for some set of blocks Y and an unused buffer slot ε. ALG'_i does *not* exploit the free slot

ε to get ahead of ALG. Rather, ALG'_i preserves ε until b is written again. Now, ε can be used to buffer b without an additional output. After this, both ALG_i and ALG'_i are in the same state again.

Finally it might be that in step s ALG_i does not output anything on disk d yet ALG'_i outputs block b according to the MIN strategy. Then the buffer pools of ALG_i can be written $Q = X + b$ whereas for ALG'_i we have $Q = X + \varepsilon$. As before, this free slot is only used to unify the states of ALG_i and ALG'_i after a possible later access to b. □

6 Application to Sorting

Optimal algorithms for read-once prefetching or write-once output scheduling can be used to analyze or improve a number of interesting parallel disk sorting algorithms. We start by discussing multiway mergesort using randomized cycling allocation (RCM) in some detail and then survey a number of additional results.

Multiway mergesort is a frequently used external sorting algorithm. We describe a variant that is similar to the SRM algorithm in [3]. Originally the N input elements are stored as a single data stream using any kind of striping. During *run formation* the input is read in chunks of size M, that are sorted internally and then written out in runs allocated using RC allocation. Neglecting trivial rounding issues, run formation is easy to do using $2N/(DB)$ I/O steps. By investing another $\mathcal{O}(N/(DB^2))$ I/O steps we can keep *triggers*, the largest keys of each block, in a separate file. Then we set aside a buffer pool of size $m = cD$ for some constant c and perform $\lceil \log_{M/B-\mathcal{O}(D)} \frac{N}{M} \rceil$ *merge phases*. In a merge phase, groups of $k = \frac{M}{B} - \mathcal{O}(D)$ runs are merged into new sorted runs, i.e., after the last merge phase, only one sorted run is left. Merging k runs of size sB can be performed using s block reads by keeping one block of each run in the internal memory of the sorting application. The order of these reads for an entire phase can be exactly predicted using the trigger information and $\mathcal{O}(N/B^2)$ I/Os for merging trigger files [3]. Hence, if we use optimal prefetching, Theorem 3 gives an upper bound of $(1 + \mathcal{O}(1/c))\frac{N}{BD} + \cdots$ for the number of fetch steps of a phase. The number of output steps for a phase is $N/(BD)$ if we have an additional output buffer of D blocks. The final result is written using any striped allocation strategy, i.e., the application calling the sorting routing need not be able to handle RC allocation. We can write the resulting total number of I/O steps as $\text{Sort}_{m+D}^{2+\mathcal{O}(1/c),\min(\frac{N}{BD},\frac{\log D}{\mathcal{O}(\gamma)})}(N)$ where

$$\text{Sort}_m^{a,f}(N) = \frac{2N}{DB} + a \cdot \frac{N}{DB} \cdot \left\lceil \log_{\frac{M}{B}-m} \frac{N}{M} \right\rceil + f + o\left(\frac{N}{DB}\right) .$$

Table 1 compares a selection of sorting algorithms using this generalized form of the I/O bound for parallel disk sorting. (In the full paper we present additional results for example for FR allocation.) The term $\frac{2N}{DB}$ represents the reading and writing of the input and the final output respectively. The factor a is a constant that dominates the I/O complexity for large inputs. Note that for $a = 2$ and

$f = m = 0$ this expression is the apparent lower bound for sorting. The additive offset f may dominate for small inputs. The reduction of the memory by m blocks in the base of the logarithm is due to memory that is used for output or prefetching buffer pools outside the merging or distribution routines, and hence reduces the number of data streams that can be handled concurrently. One way to interpret m is to view it as the amount of *additional* memory needed to match the performance of the algorithm on the multihead I/O model [1] (where load balancing disk accesses is not an issue).[4]

Table 1. Summary of Main Results for I/O Complexity of Parallel Disk Sorting Algorithms. Algorithms with **boldface** names are asymptotically optimal: M/D = Merge /Distribution sort. SM/SD = merge / distribution sort with any striping (S) allocation. SRM and SRD use Simple Randomized striping (SR). RCD, RCD+ and RCM use Randomized Cycling (RC) allocation.

a	$\mathrm{Sort}_m^{a,f}(N)$ f	I/Os $\Theta(m)$	Algorithm	Source
Deterministic algorithms				
$2+\gamma$	0	$(2D)^{1+\frac{2}{\gamma}}$	M, superblock striping	
$2+\gamma$	0	$(2D)^{1+\frac{2}{\gamma}}$	SM	here
$2+\gamma$	0	$(2D)^{1+\frac{2}{\gamma}}$	SD	here
Randomized algorithms				
$2+\gamma$	0	$D\log(D)/\gamma^2$	SRM	[3]
$2+\gamma$	0	$D\log(D)/\gamma^2$	SRD	here
$3+\gamma$	0	D/γ	**RCD**	[18]
$2+\gamma$	$\min(\frac{N}{BD}, \log(D)/\mathcal{O}(\gamma))$	D/γ	**RCM**	here
$2+\gamma$	0	D/γ	**RCD+**	here

Even without any randomization, Theorem 3 shows that mergesort with deterministic striping and optimal prefetching (SM) is at least as efficient as the common practice of using superblock striping. However, both algorithms achieve good performance only if a lot of internal memory is available.

Using previous work on distribution sort and the duality between prefetching and writing, all results obtained for mergesort can be extended to distribution sort (e.g., SD, SRD, FRD, RCD+). There are several sorting algorithms based on the distribution principle, e.g. radix sort. The bounds given here are based on a generalization of quicksort where $k-1$ splitter elements are chosen to split an unsorted input stream into k approximately equal sized output streams with

[4] If we assume a fixed memory size we cannot discriminate between some of the algorithms using the abstract I/O model. One algorithm may have a smaller factor a yet need an extra distribution or merging phase for some input sizes N. In practice, one could use a smaller block size for these input sizes. The abstract I/O model does not tell us how this affects the total I/O time needed.

disjoint ranges of keys. After $\lceil \log_{M/B-\mathcal{O}(D)} \frac{N}{M} \rceil$ splitting phases, the remaining streams can be sorted using internal sorting.

A simple variant of distribution sort with randomized cycling (RCD) was already analyzed in [18]. The new variant, RCD+, has some practical improvements (fewer tuning parameters, simpler application interface) and, it turns out that the additive term f can also be eliminated. Using a careful formulation of the algorithmic details it is never necessary to flush the write buffers. All in all, RCD+ is currently the parallel disk sorting algorithm with the best I/O performance bounds known.

Acknowledgments: We would like to thank Jeffrey Chase, Andreas Crauser, S. Mitra, Nitin Rajput, Erhard Rahm, and Berthold Vöcking for valuable discussions.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 454–462, New York, May 23–26 1998. ACM Press.
3. R. D. Barve and J. S. Vitter. A simple and efficient parallel disk mergesort. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 232–241, St. Malo, France, June 1999.
4. Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.
5. A. L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
6. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.
7. Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, Nov. 1996.
8. F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, June 1997.
9. F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. of the Intl. Parallel Processing Symmposium*, pages 14–20, April 1999.
10. M. Kallahalla and P. J.Varman. Optimal read-once parallel disk scheduling. In *IOPADS*, pages 68–77, 1999.
11. M. Kallahalla and P.J. Varman. Optimal prefetching and caching for parallel I/O systems. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, 2001. To appear.
12. Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29(4):1051–1082, 2000.
13. D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

14. M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, June–July 1993.
15. M. H. Nodine and J. S. Vitter. Greed Sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, July 1995.
16. P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, 2000.
17. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, in press. An earlier version entitled "External Memory Algorithms and Data Structures" appeared in *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, 1–38.
18. J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, Washington, January 2001.
19. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.